Texts in Computer Science

Quentin Charatan
Aaron Kans

Java in Two Semesters

Featuring JavaFX

Fourth Edition

Texts in Computer Science

Series editors

David Gries, Department of Computer Science, Cornell University, Ithaca, NY, USA
Orit Hazzan, Faculty of Education in Science and Technology, Technion—Israel Institute of Technology, Haifa, Israel

More information about this series at http://www.springer.com/series/3191

Quentin Charatan • Aaron Kans

Java in Two Semesters

Featuring JavaFX

Fourth Edition

Quentin Charatan
University of East London
London, UK

Aaron Kans
University of East London
London, UK

ISSN 1868-0941 ISSN 1868-095X (electronic)
Texts in Computer Science
ISBN 978-3-319-99419-2 ISBN 978-3-319-99420-8 (eBook)
https://doi.org/10.1007/978-3-319-99420-8

Library of Congress Control Number: 2018961214

1st edition: © The McGraw-Hill Companies 2002
2nd edition: © McGraw-Hill Education (UK) Limited 2006
3rd edition: © McGraw-Hill Education (UK) Limited 2009
4th edition: © Springer Nature Switzerland AG 2019
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Alexi

Quentin Charatan

To Wendy

Aaron Kans

Preface

Accompanying Web site: https://www.springer.com/book/9783319994192.

As with previous editions, this book is designed for university students taking a first module in software development or programming, followed by a second, more advanced module. This book uses Java as the vehicle for the teaching of programming concepts—design concepts are explained using the UML notation. The topic is taught from first principles and assumes no prior knowledge of the subject.

This book is organized so as to support two twelve-week, one-semester modules, which might typically comprise a two-hour lecture, a one-hour seminar, and a one- or two-hour laboratory session. The outcomes at the start of each chapter highlight its key learning objectives, the self-test questions at the end of each chapter ensure that the learning objectives for that chapter have been met, while the programming exercises that follow allow these learning objectives to be applied to complete programs. In addition to these exercises and questions, a case study is developed in each semester to illustrate the use of the techniques covered in the text to develop a non-trivial application. Lecturers who teach on modules that run for fewer than twelve weeks in a semester could treat these case studies as a self-directed student learning experience, rather than as taught topics.

The approach taken in this book is ideal for all students including those entering university with little or no background in the subject matter, perhaps coming from pre-degree courses in other disciplines, or perhaps returning to study after long periods away from formal education. It is the authors' experience that such students have enormous difficulties in grasping the fundamental programming concepts the first time round and therefore require a simpler and gentler introduction to the subject that is presented in most standard texts.

This book takes an integrated approach to software development by covering such topics as basic design principles and standards, testing methodologies, and the user interface, as well as looking at detailed implementation topics.

In the first semester, considerable time is spent concentrating on the fundamental programming concepts such as declarations of variables and basic control structures, methods and arrays, prior to introducing students to classes and objects, inheritance, graphics, and event-driven programming.

The second semester covers more advanced topics such as interfaces, exceptions, collection classes from the Java collections framework, advanced graphics,

file-handling techniques, packages, the implementation of multi-threaded programs, socket programming, and processing collections using streams.

The fourth edition achieves three main goals. Firstly, it incorporates all the very useful feedback on the third edition that we have received from students and lecturers since its publication. Secondly, it includes many new questions and programming exercises at the end of the chapters. Finally, it includes new material to bring it completely up to date with the current developments in the field—in particular a number of key developments that were introduced in Java 8 which, according to Oracle™, is "the most significant re-engineering of the language since its launch."

One key feature of this new edition is that all graphical user interface developments are based on *JavaFX*, rather than the Swing Technology used in previous editions. JavaFX allows for the creation of sophisticated modern graphical interfaces that can run on a variety of devices and is now Oracle's preferred technology for building such interfaces, having decided that Swing will no longer be developed. JavaFX therefore plays a very significant role throughout the new text, and three new chapters are devoted to it.

Other key developments arising from Java 8 that have been incorporated into the new text include *lambda expressions,* which allow us to simplify development considerably by passing functions as arguments to methods, and the new *Stream API*, a technology that allows us to process collections in a very concise, declarative style of programming.

In addition to the above key changes, we also introduce techniques to improve the robustness of code—in particular the *Optional* class for dealing with empty values and the *try-with-resources* construct to ensure resources such as files are safely closed before exiting methods.

As well as adding these new features, some existing chapters have undergone significant enhancements. The Java Collections Framework chapter, for example, has been expanded to include a comprehensive section on the *sort* methods available in various classes and interfaces in Java. The coverage of generics has also been considerably expanded and the packages chapter now introduces the *Hibernate ORM* technology for accessing remote databases.

The accompanying Web site (see URL above) contains all the codes from the textbook and a guide on how to install and use the NetBeans™ Java IDE, as well as a collection of other useful resources.

We would like to thank our publisher, Springer, for the encouragement and guidance that we have received throughout the production of this book. Additionally, we would especially like to thank the computing students of the University of East London for their thoughtful comments and feedback and Steven Martin for his help and advice. For support and inspiration, special thanks are due once again to our families and friends.

London, UK

Quentin Charatan
Aaron Kans

Contents

Part I Semester One

1 The First Step 3
- 1.1 Introduction 3
- 1.2 Software 4
- 1.3 Compiling Programs 4
- 1.4 Programming in Java 5
- 1.5 Integrated Development Environments (IDEs) 6
- 1.6 Java Applications 8
- 1.7 Your First Program 10
  - 1.7.1 Analysis of the "Hello World" Program 11
  - 1.7.2 Adding Comments to a Program 13
- 1.8 Output in Java 14
- 1.9 Self-test Questions 16
- 1.10 Programming Exercises 17

2 Building Blocks 19
- 2.1 Introduction 19
- 2.2 Simple Data Types in Java 19
- 2.3 Declaring Variables in Java 21
- 2.4 Assignments in Java 23
- 2.5 Creating Constants 25
- 2.6 Arithmetic Operators 25
- 2.7 Expressions in Java 27
- 2.8 More About Output 30
- 2.9 Input in Java: The *Scanner* Class 31
- 2.10 Program Design 35
- 2.11 Self-test Questions 36
- 2.12 Programming Exercises 38

3 Selection 41
- 3.1 Introduction 41
- 3.2 Making Choices 42

3.3 The 'if' Statement . . . 43
3.3.1 Comparison Operators . . . 46
3.3.2 Multiple Instructions Within an 'if' Statement . . . 47
3.4 The 'if…else' Statement . . . 49
3.5 Logical Operators . . . 51
3.6 Nested 'if…else' Statements . . . 53
3.7 The 'switch' Statement . . . 55
3.7.1 Grouping Case Statements . . . 56
3.7.2 Removing Break Statements . . . 57
3.8 Self-test Questions . . . 59
3.9 Programming Exercises . . . 61

4 Iteration . . . 65
4.1 Introduction . . . 65
4.2 The 'for' Loop . . . 67
4.2.1 Varying the Loop Counter . . . 70
4.2.2 The Body of the Loop . . . 72
4.2.3 Revisiting the Loop Counter . . . 76
4.3 The 'while' Loop . . . 77
4.4 The 'do…while' Loop . . . 79
4.5 Picking the Right Loop . . . 83
4.6 The 'break' Statement . . . 84
4.7 The 'continue' Statement . . . 86
4.8 Self-test Questions . . . 88
4.9 Programming Exercises . . . 91

5 Methods . . . 95
5.1 Introduction . . . 95
5.2 Declaring and Defining Methods . . . 96
5.3 Calling a Method . . . 98
5.4 Method Input and Output . . . 99
5.5 More Examples of Methods . . . 103
5.6 Variable Scope . . . 107
5.7 Method Overloading . . . 109
5.8 Using Methods in Menu-Driven Programs . . . 112
5.9 Self-test Questions . . . 115
5.10 Programming Exercises . . . 117

6 Arrays . . . 119
6.1 Introduction . . . 119
6.2 Creating an Array . . . 120
6.3 Accessing Array Elements . . . 124
6.4 Passing Arrays as Parameters . . . 129
6.5 Varargs . . . 131

6.6 Returning an Array from a Method 134
6.7 The Enhanced ‘for’ Loop 137
6.8 Some Useful Array Methods 139
6.8.1 Array Maximum 139
6.8.2 Array Summation 141
6.8.3 Array Membership 141
6.8.4 Array Search 142
6.8.5 The Final Program 143
6.9 Multi-dimensional Arrays 148
6.9.1 Creating a Two-Dimensional Array 148
6.9.2 Initializing Two-Dimensional Arrays 149
6.9.3 Processing Two-Dimensional Arrays 150
6.9.4 The *MonthlyTemperatures* Program 151
6.10 Ragged Arrays 155
6.11 Self-test Questions 158
6.12 Programming Exercises 161

7 Classes and Objects 163
7.1 Introduction 163
7.2 Classes as Data Types 163
7.3 Objects 165
7.4 The *Oblong* Class 166
7.5 The *OblongTester* Program 171
7.6 Strings 173
7.6.1 Obtaining Strings from the Keyboard 173
7.6.2 The Methods of the *String* Class 174
7.6.3 Comparing Strings 176
7.6.4 Entering Strings Containing Spaces 178
7.7 Our Own *Scanner* Class for Keyboard Input 179
7.8 The *Console* Class 181
7.9 The *BankAccount* Class 182
7.10 Arrays of Objects 185
7.11 The *ArrayList* Class 188
7.12 Self-test Questions 190
7.13 Programming Exercises 192

8 Implementing Classes 195
8.1 Introduction 195
8.2 Designing Classes in UML Notation 196
8.3 Implementing Classes in Java 198
8.3.1 The *Oblong* Class 198
8.3.2 The *BankAccount* Class 202
8.4 The *static* Keyword 205
8.5 Initializing Attributes 208

8.6 The *EasyScanner* Class . . . 209
8.7 Passing Objects as Parameters . . . 209
8.8 Collection Classes . . . 211
8.8.1 The *Bank* Class . . . 211
8.8.2 Testing the *Bank* Class . . . 217
8.9 The Benefits of Object-Oriented Programming . . . 223
8.10 Self-test Questions . . . 223
8.11 Programming Exercises . . . 227

9 Inheritance . . . 235
9.1 Introduction . . . 235
9.2 Defining Inheritance . . . 236
9.3 Implementing Inheritance in Java . . . 237
9.4 Extending the *Oblong* Class . . . 241
9.5 Method Overriding . . . 245
9.6 Abstract Classes . . . 250
9.7 Abstract Methods . . . 253
9.8 The `final` Modifier . . . 257
9.9 The *Object* Class . . . 257
9.10 The `toString` Method . . . 258
9.11 Wrapper Classes and Autoboxing . . . 259
9.12 Self-test Questions . . . 260
9.13 Programming Exercises . . . 264

10 Introducing JavaFX . . . 265
10.1 Introduction . . . 265
10.2 A Brief History of Java Graphics . . . 266
10.3 JavaFX: An Overview . . . 267
10.4 2D Graphics: The *SmileyFace* Class . . . 269
10.5 Event-Handling in JavaFX: The *ChangingFace* Class . . . 275
10.6 Some More 2D Shapes . . . 281
10.7 An Interactive Graphics Class . . . 282
10.8 A Graphical User Interface (GUI) for the *Oblong* Class . . . 285
10.9 Containers and Layouts . . . 288
10.9.1 More About *HBox* and *VBox* . . . 288
10.9.2 GridPane . . . 290
10.9.3 StackPane . . . 291
10.9.4 *FlowPane* and *BorderPane* . . . 292
10.10 Borders, Fonts and Colours . . . 293
10.10.1 Borders . . . 294
10.10.2 Fonts . . . 295
10.10.3 Colours . . . 296
10.11 Number Formatting . . . 297
10.12 A Metric Converter . . . 299

10.13 Self-test Questions 302
10.14 Programming Exercises 303

11 Case Study—Part 1 307
11.1 Introduction 307
11.2 The Requirements Specification 308
11.3 The Design 308
11.4 Implementing the *Payment* Class 310
11.5 The *PaymentList* Class 313
11.5.1 Javadoc 317
11.5.2 Code Layout 319
11.6 Testing the *PaymentList* Class 320
11.7 Implementing the *Tenant* Class 328
11.8 Implementing the *TenantList* Class 330
11.9 Self-test Questions 334
11.10 Programming Exercises 334

12 Case Study—Part 2 335
12.1 Introduction 335
12.2 Keeping Permanent Records 335
12.3 Design of the *Hostel* Class 336
12.4 Design of the GUI 338
12.5 Designing the Event-Handlers 341
12.6 Implementing the *Hostel* Class 344
12.7 Testing the System 350
12.8 What Next? 353
12.9 Self-test Questions 354
12.10 Programming Exercises 354

Part II Semester Two

13 Interfaces and Lambda Expressions 357
13.1 Introduction 357
13.2 An Example 358
13.3 Interfaces 359
13.4 Inner Classes 364
13.5 Anonymous Classes 364
13.6 Lambda Expressions 368
13.6.1 The Syntax of Lambda Expressions 369
13.6.2 Variable Scope 371
13.6.3 Example Programs 371
13.6.4 Method References—The Double Colon Operator 374
13.7 Generics 376

13.7.1 Bounded Type Parameters 379
13.7.2 Wildcards 382
13.8 Other Interfaces Provided with the Java Libraries 383
13.9 Polymorphism and Polymorphic Types 385
13.9.1 Operator Overloading 385
13.9.2 Method Overloading 385
13.9.3 Method Overriding 385
13.9.4 Type Polymorphism 386
13.10 Self-test Questions 386
13.11 Programming Exercises 391

14 Exceptions 393
14.1 Introduction 393
14.2 Pre-defined Exception Classes in Java 394
14.3 Handling Exceptions 395
14.3.1 Claiming an Exception 398
14.3.2 Catching an Exception 401
14.4 The ‘finally’ Clause 403
14.5 The ‘Try-with-Resources’ Construct 406
14.6 Null-Pointer Exceptions 408
14.7 The *Optional* Class 410
14.8 Exceptions in GUI Applications 413
14.9 Using Exceptions in Your Own Classes 416
14.9.1 Throwing Exceptions 417
14.9.2 Creating Your Own Exception Classes 419
14.10 Documenting Exceptions 421
14.11 Self-test Questions 422
14.12 Programming Exercises 423

15 The Java Collections Framework 427
15.1 Introduction 427
15.2 The *List* Interface and the *ArrayList* Class 428
15.2.1 Creating an *ArrayList* Collection Object 429
15.2.2 The Interface Type Versus the Implementation Type 430
15.2.3 *List* Methods 431
15.3 The Enhanced *for* Loop and Java Collections 434
15.4 The *forEach* Loop 435
15.5 The *Set* Interface and the *HashSet* Class 436
15.5.1 *Set* Methods 437
15.5.2 Iterating Through the Elements of a Set 438
15.5.3 *Iterator* Objects 440

15.6 The *Map* Interface and the *HashMap* Class . . . 443
15.6.1 *Map* Methods . . . 444
15.6.2 Iterating Through the Elements of a Map . . . 446
15.7 Using Your Own Classes with Java's Collection Classes . . . 447
15.7.1 The *Book* Class . . . 448
15.7.2 Defining an *equals* Method . . . 450
15.7.3 Defining a *hashCode* Method . . . 450
15.7.4 The Updated *Book* Class . . . 452
15.8 Developing a Collection Class for *Book* Objects . . . 453
15.9 Sorting Objects in a Collection . . . 456
15.9.1 The *Collections.sort* and *Arrays.sort* Methods . . . 456
15.9.2 The *Comparable<T>* Interface . . . 458
15.9.3 The *Comparator<T>* Interface . . . 459
15.10 Self-test Questions . . . 463
15.11 Programming Exercises . . . 465

16 Advanced JavaFX . . . 469
16.1 Introduction . . . 469
16.2 Input Events . . . 470
16.2.1 Mouse Events . . . 470
16.2.2 Key Events . . . 473
16.3 Binding Properties . . . 477
16.4 The *Slider* Class . . . 479
16.5 Multimedia Nodes . . . 482
16.5.1 Embedding Images . . . 483
16.5.2 Embedding Videos . . . 486
16.5.3 Embedding Web Pages . . . 489
16.6 Cascading Style Sheets . . . 491
16.7 Self-test Questions . . . 496
16.8 Programming Exercises . . . 497

17 JavaFX: Interacting with the User . . . 499
17.1 Introduction . . . 499
17.2 Drop-Down Menus . . . 500
17.3 Context (Pop-Up) Menus . . . 503
17.4 Combo Boxes . . . 507
17.5 Check Boxes and Radio Buttons . . . 509
17.6 A Card Menu . . . 513
17.7 The *Dialog* Class . . . 518
17.8 Self-test Questions . . . 524
17.9 Programming Exercises . . . 525

18 Working with Files . . . 527
18.1 Introduction . . . 527
18.2 Input and Output . . . 528
18.3 Input and Output Devices . . . 528
18.4 File-Handling . . . 530
18.4.1 Encoding . . . 530
18.4.2 Access . . . 531
18.5 Reading and Writing to Text Files . . . 531
18.6 Reading and Writing to Binary Files . . . 539
18.7 Reading a Text File Character by Character . . . 541
18.8 Object Serialization . . . 542
18.9 Random Access Files . . . 544
18.10 Self-test Questions . . . 549
18.11 Programming Exercises . . . 550

19 Packages . . . 553
19.1 Introduction . . . 553
19.2 Understanding Packages . . . 553
19.3 Accessing Classes in Packages . . . 555
19.4 Developing Your Own Packages . . . 558
19.5 Package Scope . . . 559
19.6 Running Applications from the Command Line . . . 560
19.7 Deploying Your Packages . . . 563
19.8 Adding External Libraries . . . 564
19.8.1 Accessing Databases Using JDBC . . . 564
19.8.2 Accessing Databases Using Hibernate . . . 568
19.9 Self-test Questions . . . 574
19.10 Programming Exercises . . . 575

20 Multi-threaded Programs . . . 577
20.1 Introduction . . . 577
20.2 Concurrent Processes . . . 578
20.3 Threads . . . 578
20.4 The *Thread* Class . . . 580
20.5 Thread Execution and Scheduling . . . 582
20.6 Synchronizing Threads . . . 584
20.7 Thread States . . . 585
20.8 Multithreading and JavaFX . . . 587
20.8.1 The `Task` Class . . . 587
20.8.2 The `Service` Class . . . 590
20.8.3 Automating the *ChangingFace* Application . . . 591
20.8.4 Running a Task in the Background . . . 594
20.8.5 Animation Using a Series of Images . . . 596

20.9 Self-test Questions 599
20.10 Programming Exercises 600

21 Advanced Case Study 603
21.1 Introduction 603
21.2 System Overview 604
21.3 Requirements Analysis and Specification 604
21.4 Design 606
21.5 Enumerated Types in UML 608
21.6 Implementation 609
21.6.1 Implementing Enumerated Types in Java 609
21.6.2 The *Runway* Class 611
21.6.3 The *Plane* Class 612
21.6.4 The *Airport* Class 616
21.7 Testing 624
21.8 Design of the JavaFX Interface 625
21.9 The *TabPane* Class 626
21.10 The *AirportFrame* Class 628
21.11 Self-test Questions 638
21.12 Programming Exercises 639

22 The Stream API 641
22.1 Introduction 641
22.2 Streams Versus Iterations: Example Program 643
22.3 Creating Streams 646
22.4 Intermediate Operations 648
22.5 Operations for Terminating Streams 652
22.5.1 More Examples 652
22.5.2 Collecting Results 654
22.6 Concatenating Streams 656
22.7 Infinite Streams 656
22.8 Stateless and Stateful Operations 657
22.9 Parallelism 658
22.10 Self-test Questions 659
22.11 Programming Exercises 659

23 Working with Sockets 661
23.1 Introduction 661
23.2 Sockets 662
23.3 A Simple Server Application 663
23.4 A Simple Client Application 668
23.5 Connections from Multiple Clients 673

23.6 A Client–Server Chat Application 676
23.7 Self-test Questions 686
23.8 Programming Exercises 686

24 Java in Context 689
24.1 Introduction 689
24.2 Language Size 690
24.2.1 Pointers 690
24.2.2 Multiple Inheritance 691
24.3 Language Reliability 694
24.3.1 Aliasing 695
24.3.2 Overriding the *clone* Method 697
24.3.3 Immutable Objects 700
24.3.4 Using the *clone* Method of the *Object* Class 701
24.3.5 Copy Constructors 703
24.3.6 Garbage Collection 704
24.4 The Role of Java 706
24.5 What Next? 706
24.6 Self-test Questions 707
24.7 Programming Exercises 708

Index 709

Part I
Semester One

Check for updates

1 The First Step

Outcomes:

By the end of this chapter you should be able to:

- *explain the meaning of the terms* ***software***, ***program***, ***source code, program code***;
- *distinguish between* ***application software*** *and* ***system software***;
- *explain how Java programs are compiled and run*;
- *provide examples of different types of java applications*;
- *write Java programs that display text on the screen*;
- *join messages in output commands by using the* ***concatenation*** *(+) operator*;
- *add comments to programs.*

1.1 Introduction

Like any student starting out on a first programming module, you will be itching to do just one thing—get started on your first program. We can well understand that, and you won't be disappointed, because you will be writing programs in this very first chapter. Designing and writing computer programs can be one of the most enjoyable and satisfying things you can do, although it can seem a little daunting at first because it is like nothing else you have ever done. But, with a bit of perseverance, you will not only start to get a real taste for it but you may well find yourself sitting up till two o'clock in the morning trying to solve a problem. And just when you have given up and you are dropping off to sleep, the answer pops into your head and you are at the computer again until you notice it is getting light outside! So if this is happening to you, then don't worry—it's normal!

However, before you start writing programs we need to make sure that you understand what we mean by important terms such as *program*, *software, code* and *programming languages*.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_1

1.2 Software

A computer is not very useful unless we give it some instructions that tell it what to do. This set of instructions is called a **program**. Programs that the computer can use can be stored on electronic chips that form part of the computer, or can be stored on devices like hard disks, CDs, DVDs, and USB drives (sometimes called memory sticks), and can often be downloaded via the Internet.

The word **software** is the name given to a single program or a set of programs. There are two main kinds of software:

- **Application software**. This is the name given to useful programs that a user might need; for example, word-processors, spreadsheets, accounts programs, games and so on. Such programs are often referred to simply as **applications**.
- **System software**. This is the name given to special programs that help the computer to do its job; for example, operating systems (such as UNIX™ or Windows™, which help us to use the computer) and network software (which helps computers to communicate with each other).

Of course software is not restricted simply to computers themselves. Many of today's devices—from mobile phones to microwave ovens to games consoles—rely on computer programs that are built into the device. Such software is referred to as **embedded software**.

Both application and system software are built by writing a set of instructions for the computer to obey. **Programming**, or **coding**, is the task of writing these instructions. These instructions have to be written in a language specially designed for this purpose. These **programming languages** include C++, Visual Basic, Python and many more. The language we are going to use in this book is Java. Java is an example of an **object-oriented** programming language. Right now, that phrase might not mean anything to you, but you will find out all about its meaning as we progress through this book.

1.3 Compiling Programs

Like most modern programming languages, the Java language consists of instructions that look a bit like English. For example, words such as **while** and **if** are part of the Java language. The set of instructions written in a programming language is called the **program code** or **source code**.

Ultimately these instructions have to be translated into a language that can be understood by the computer. The computer understands only **binary** instructions—that means instructions written as a series of 0s and 1s. So, for example, the machine might understand 01100111 to mean add. The language of the computer is often referred to as **machine code**. A special piece of system software called a **compiler** translates the instructions written in a programming language into

machine instructions consisting of 0s and 1s. This process is known as **compiling**. Figure 1.1 illustrates how this process works for many programming languages.

Programming languages have a very strict set of rules that you must follow. Just as with natural languages, this set of rules is called the **syntax** of the language. A program containing syntax errors will not compile. You will see when you start writing programs that the sort of things that can cause compiler errors are the incorrect use of special Java keywords, missing brackets or semi-colons, and many others. If, however, the source code is free of such errors the compiler will successfully produce a machine code program that can be run on a computer, as illustrated.

Once a program has been compiled and the machine code program saved, it can be run on the target machine as many times as necessary. When you buy a piece of software such as a game or a word processor, it is this machine code program that you are buying.

1.4 Programming in Java

Before the advent of Java, most programs were compiled as illustrated in Fig. 1.1. The only problem with this approach is that the final compiled program is suitable only for a particular type of computer. For example, a program that is compiled for a PC will not run on a Mac™ or a UNIX™ machine.

But this is not the case with Java. Java—and nowadays many other languages—is **platform-independent**. A Java program will run on any type of computer.

How is this achieved? The answer lies in the fact that any Java program requires the computer it is running on to also be running a special program called a **Java Virtual Machine**, or **JVM** for short. This JVM is able to run a Java program for the particular computer on which it is running.

For example, you can get a JVM for a PC running Windows™; there is a JVM for a MAC™, and one for a Unix™ or Linux™ box. There is a special kind of JVM for mobile phones; and there are JVMs built into machines where the embedded software is written in Java.

We saw earlier that conventional compilers translate our program code into machine code. This machine code would contain the particular instructions appropriate to the type of computer it was meant for. Java compilers do not translate the program into machine code—they translate it into special instructions called **Java byte code**. Java byte code, which, like machine code, consists of 0s and 1s, contains instructions that are exactly the same irrespective of the type of computer—it is *universal*, whereas machine code is specific to a particular type of computer. The job of the JVM is to translate each byte code instruction for the computer it is running on, before the instruction is performed. See Fig. 1.2.

There are various ways in which a JVM can be installed on a computer. In the case of some operating systems a JVM comes packaged with the system, along with the Java **libraries**, or **packages**, (pre-compiled Java modules that can be integrated

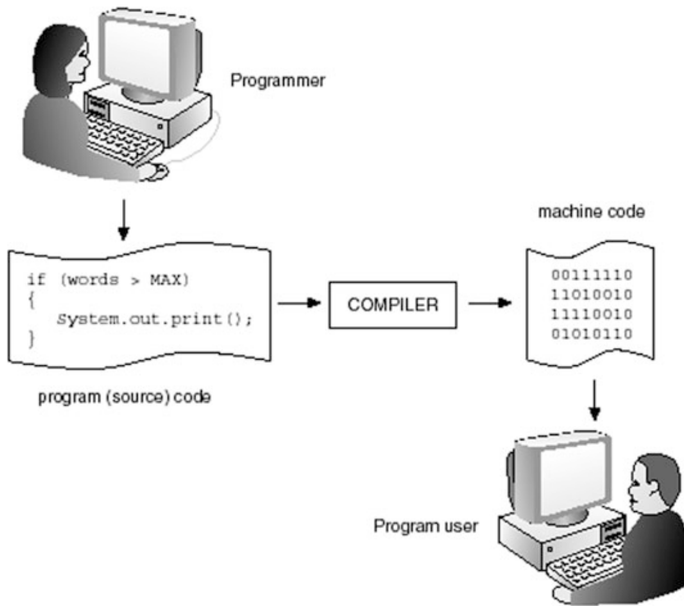


Fig. 1.1 The compilation process

with the programs you create) and a compiler. Together the JVM and the libraries are known as the **Java Runtime Environment (JRE)**. If you do not have a JRE on your computer (as will be the case with any Windows™ operating system), then the entire Java Development Kit (JDK), comprising the JRE, compiler and other tools, can be downloaded from Oracle™, the owners of the Java platform.[1]

1.5 Integrated Development Environments (IDEs)

It is very common to compile and run your programs by using a special program called an **Integrated Development Environment** or **IDE**. An IDE provides you with an easy-to-use window into which you can type your code; other windows will provide information about the files you are using; and a separate window will be provided to tell you of your errors.

Not only does an IDE do all these things, it also lets you run your programs as soon as you have compiled them. Depending on the IDE you are using, your screen will look something like that in Fig. 1.3.

[1]The original developers of Java were Sun Microsystems™. This company was acquired by Oracle™ in 2010.

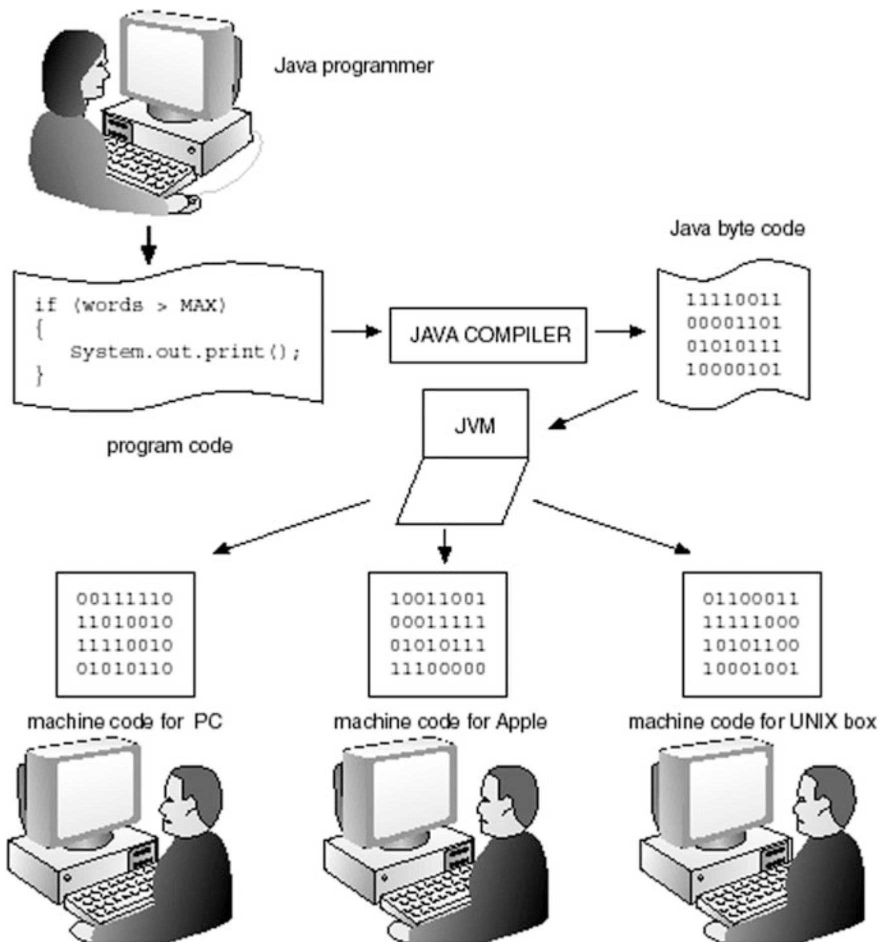


Fig. 1.2 Compiling Java programs

The IDE shown in Fig. 1.3 is NetBeans™, a very commonly used compiler for Java—another widely used IDE is Eclipse™. Instructions for installing and using an IDE are on the website (see preface for details).

It is perfectly possible to compile and run Java programs without the use of an IDE—but not nearly so convenient. You would do this from a command line in a console window. The source code that you write is saved in the form of a simple text file which has a `.java` extension. The compiler that comes as part of the JDK is called `javac.exe`, and to compile a file called, for example, `MyProgram.java`, you would write at the command prompt:

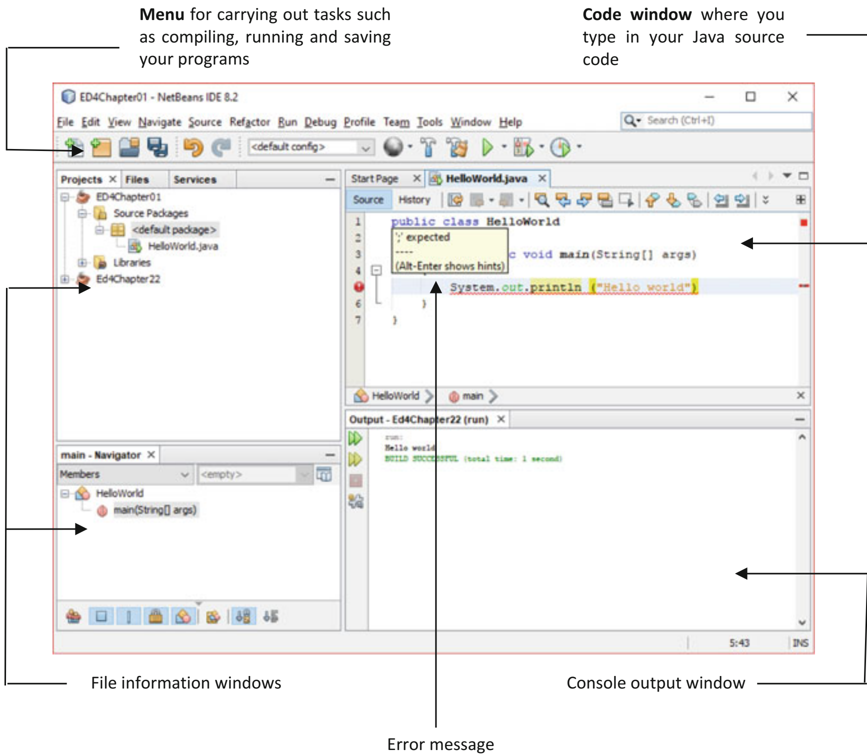


Fig. 1.3 A typical Java IDE screen

javac MyProgram.java

This would create a file called `MyProgram.class`, which is the compiled file in Java byte code. The name of the JVM is `java.exe` and to run the program you would type:

java MyProgram

To start off with however, we strongly recommend that you use an IDE such as NetBeans™ or Eclipse™.

1.6 Java Applications

As we explained in Sect. 1.2, Java applications can run on a computer, on such devices as mobile phones and games consoles, or sometimes can be embedded into an electronic device. In the last case you would probably be unaware of the fact that the software is running at all, whereas in the former cases you would be seeing

output from your program on a screen and providing information to your program via a keyboard and mouse, via a touch screen, or via a joystick or game controller.

The screen that provides output from your program, and prompts you to enter information, is known as the **user interface**. There are two principal types of user interface:

- **text** based;
- **graphics** based.

With text based user interfaces, information is displayed simply as text—with no pictures. Text based programs make use of the keyboard for user input. Text based programs are known as **console applications**. If you are using an IDE, the console window is usually integrated into the IDE as you saw in Fig. 1.3. However, if you are running a program from the command prompt you will see a window similar to that shown in Fig. 1.4.

You are probably more accustomed to running programs that have a **graphical user interface (GUI)**. Such interfaces allow for pictures and shapes to be drawn on the screen (such as text boxes and buttons) and make use of the mouse as well as the keyboard to collect user input. An example of a GUI is given in Fig. 1.5.

Eventually we want all your programs to have graphical interfaces, but these obviously require a lot more programming effort to create than simple console applications. So, for most of the first semester, while we are teaching you the fundamentals of programming in Java, we are going to concentrate on getting the program logic right and we will be sticking to console style applications. Once you have mastered these fundamentals, however, you will be ready to create attractive graphical interfaces before the end of this very first semester.

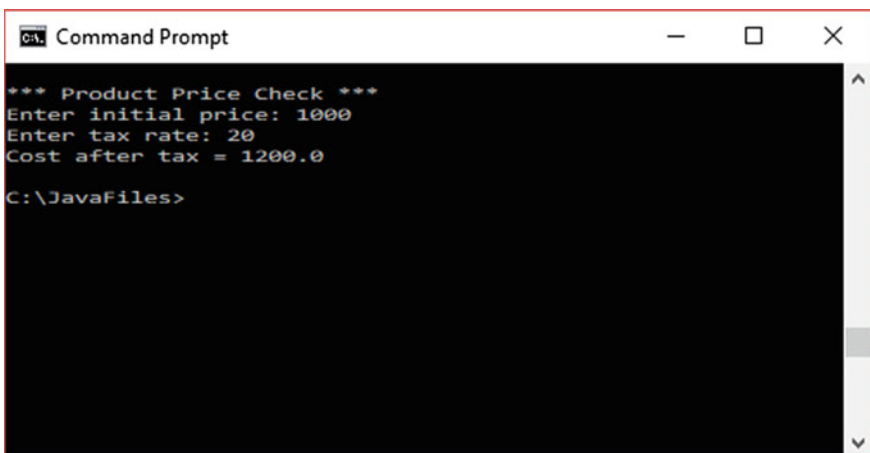


Fig. 1.4 A Java console application

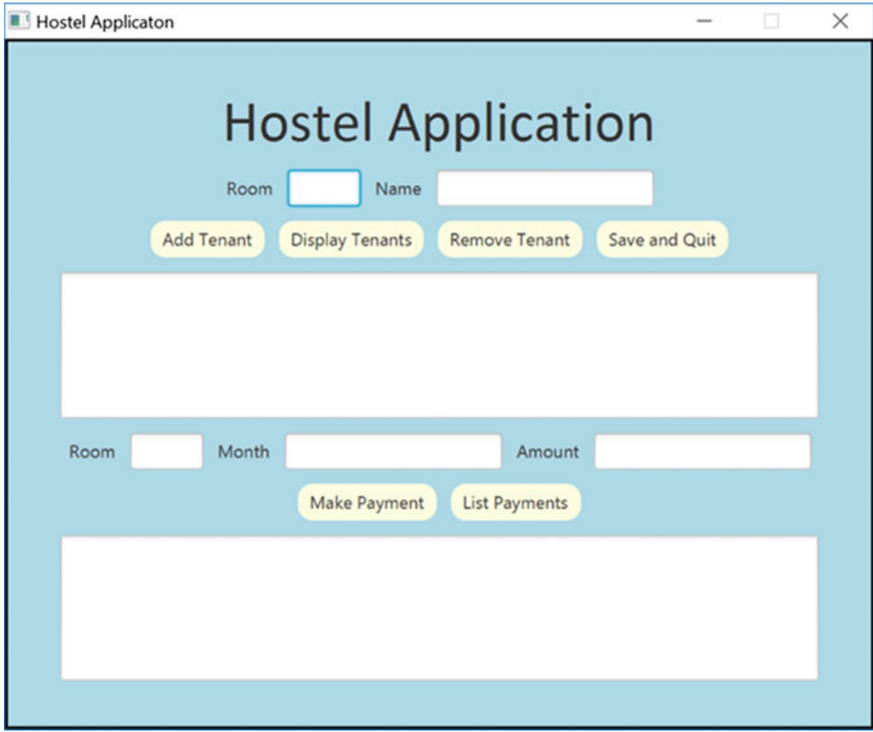


Fig. 1.5 A graphical application

1.7 Your First Program

Now it is time to write your first program. Anyone who knows anything about programming will tell you that the first program that you write in a new language has always got to be a program that displays the words "Hello world" on the screen; so we will stick with tradition, and your first program will do exactly that!

When your program runs you will see the words "Hello world" displayed. The type of window in which this is displayed will vary according to the particular operating system you are running, and the particular compiler you are using.

The code for the "Hello world" program is written out for you below.

HelloWorld

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println ("Hello world");
    }
}
```

1.7.1 Analysis of the "Hello World" Program

Let's start with the really important bit—the line of code that represents the instruction *display "Hello world" on the screen*. The line that does this looks like this:

```
System.out.println("Hello world");
```

This is the way we are always going to get stuff printed on a simple text screen; we use `System.out.println` (or sometimes `System.out.print`, as explained below) and put whatever we want to be displayed in the brackets. The `println` is short for "print line" by the way. You won't understand at this stage why it has to be in this precise form (with each word separated by a full stop, and the actual phrase in double quotes), but do make sure that you type it exactly as you see it here, with an upper case *S* at the beginning. Also, you should notice the semi-colon at the end of the statement. This is important; every Java instruction has to end with a semi-colon.

Now we can consider the meaning of the rest of the program. The first line, which we call the header, looks like this:

```
public class HelloWorld
```

The first, and most important, thing to pay attention to is the word **class**. We noted earlier that Java is referred to as an *object-oriented* programming language. Now, the true meaning of this will start to become clear in Chap. 7—but for the time being you just need to know that object-oriented languages require the program to be written in separate units called **classes**. The simple programs that we are starting off with will contain only one class (although they will interact with other classes from the "built-in" Java libraries). We always have to give a name to a class and in this case we have simply called our class `HelloWorld`.

When choosing a name for a class, you can choose any name as long as:

- the name is not already a keyword in the Java language (such as **`static, void`**);
- the name has no spaces in it;
- the name does not include operators or mathematical symbols such as + and −;
- the name starts either with a letter, an underscore (_), or a dollar sign ($).

So, the first line tells the Java compiler that we are writing a class with the name `HelloWorld`. However, you will also have noticed the word **`public`** in front of

the word **class**; placing this word here makes our class accessible to the outside world and to other classes—so, until we learn about specific ways of restricting access (in the second semester) we will always include this word in the header. A **public** class should always be saved in a file with the same name as the class itself—so in this case it should be saved as a file with the name HelloWorld.java.

Notice that everything in the class has to be contained between two curly brackets (known as **braces**) that look like this {}; these tell the compiler where the class begins and ends.

There is one important thing that we must emphasize here. Java is *case-sensitive* —in other words it interprets upper case and lower case characters as two completely different things—it is very important therefore to type the statements exactly as you see them here, paying attention to the case of the letters.

The next line that we come across (after the opening curly bracket) is this:

```
public static void main(String[] args)
```

This looks rather strange if you are not used to programming—but you will see that every application we write is going to contain one class with this line in it. In Chap. 7 you will find out that this basic unit called a class is made up of, among other things, a number of **methods**. You will find out a lot more about methods in Chap. 5, but for now it is good enough for you to know that a method contains a particular set of instructions that the computer must carry out. Our HelloWorld class contains just one method and this line introduces that method. In fact it is a very special method called a main method. Applications in Java must always contain a class with a method called main: this is where the program begins. A program starts with the first instruction of main, then obeys each instruction in sequence (unless the instruction itself tells it to jump to some other place in the program). The program terminates when it has finished obeying the final instruction of main.[2]

So this line that we see above introduces the main method; the program instructions are now written in a second set of curly brackets that show us where this main method begins and ends. At the moment we will not worry about the words **public static void** in front of main, and the bit in the brackets afterwards (String[] args)[3]—we will just accept that they always have to be there; you will begin to understand their significance as you learn more about

[2]In Chap. 10 you will learn to create graphics programs with a package called JavaFX, and in the case of JavaFX applications you will see that in some instances it is possible to run a JavaFX application without a main method.

[3]In fact, if you left out the words in brackets your program would still compile—but it wouldn't do what you wanted it to do!

programming concepts. The top line of a method is referred to as the method **header** and words such as **`public`** and **`static`**, that are part of the Java language, are referred to as **keywords**.[4]

As we have said, we place instructions inside a method by surrounding them with opening and closing curly brackets. In Java, curly brackets mark the beginning and end of a group of instructions. In this case we have only one instruction inside the curly brackets but, as you will soon see, we can have many instructions inside these braces.

By the way, you should be aware that the compiler is not concerned about the layout of your code, just that your code meets the rules of the language. So we could have typed the method header, the curly brackets and the `println` command all on one line if we wished! Obviously this would look very messy, and it is always important to lay out your code in a way that makes it easy to read and to follow. So throughout this book we will lay out our code in a neat easy-to-read format, lining up opening and closing braces.

1.7.2 Adding Comments to a Program

When we write program code, we will often want to include some comments to help remind us what we were doing when we look at our code a few weeks later, or to help other people to understand what we have done.

Of course, we want the compiler to ignore these comments when the code is being compiled. There are different ways of doing this. For short comments we place two slashes (//) at the beginning of the line—everything after these slashes, up to the end of the line, is then ignored by the compiler.

For longer comments (that is, ones that run over more than a single line) we usually use another approach. The comment is enclosed between two special symbols; the opening symbol is a slash followed by a star (/*) and the closing symbol is a star followed by a slash (*/). Everything between these two symbols is ignored by the compiler. The program below shows examples of both types of comment; when you compile and run this program you will see that the comments have no effect on the code, and the output is exactly the same as that of the original program.

[4] You will notice that we are using bold courier font for Java keywords.

HelloWorld – with comments

```
// this is a short comment, so we use the first method

public class HelloWorld
{
     public static void main(String[] args)
     {
          System.out.println("Hello world");
     }

/* this is the second method of including comments - it is more convenient to use this
method here, because the comment is longer and goes over more than one line */

}
```

In Chap. 11 you will learn about a special tool called **Javadoc** for documenting your programs. In that chapter you will see that in order to use this tool you must comment your classes in the Javadoc style—as you will see, Javadoc comments must begin with /** and end with */.

1.8 Output in Java

As you have already seen when writing your first program, to output a message on to the screen in Java we use the following command:

`System.out.println(`*`message to be printed on screen`*`);`

For example, we have already seen:

```
System.out.println("Hello world");
```

This prints the message "Hello world" onto the screen. There is in fact an alternative form of the `System.out.println` statement, which uses `System.out.`**`print`**. As we said before, `println` is short for *print line* and the effect of this statement is to start a new line after displaying whatever is in the brackets. You can see the effect of this below—we have adapted our program by adding an additional line.

HelloWorld – with an additional line

```
public class HelloWorld
{
     public static void main(String[] args)
     {
          System.out.println("Hello world"); // notice the use of println
          System.out.println("Hello world again!");
     }
}
```

When we run this program, the output looks like this:

Hello world
Hello world again!

Now let's change the first `System.out.println` to `System.out.print`:

HelloWorld – adapted to show the effect of using *print* instead of *println*

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.print("Hello world"); // notice the use of 'print'
        System.out.println("Hello world again!");
    }
}
```

Now our output looks like this:

`Hello worldHello world again!`

You can see that the output following the `System.out.print` statement doesn't start on a new line, but follows straight on from the previous line.

By the way, if you want a blank line in the program, then you can simply use `println` with empty brackets:

```
System.out.println();
```

Messages such as "Hello world" are in fact what we call **strings** (collections of characters). In Java, literal strings like this are always enclosed in speech marks. We shall explore strings in depth in Chap. 7. However, it is useful to know now how several strings can be printed on the screen using a single output command.

In Java, two strings can be joined together with the plus symbol (+). When using this symbol for this purpose it is known as the **concatenation operator**. For example, instead of printing the single string "Hello world", we could have joined two strings, "Hello" and "world", for output using the following command:

```
System.out.println("Hello " + "world");
```

Note that spaces are printed by including them within the speech marks ("`Hello `"), not by adding spaces around the concatenation operator (which has no effect at all).

1.9 Self-test Questions

1. Explain the meaning of the following terms:

   - program;
   - software;
   - application software;
   - system software;
   - machine code;
   - source code;.
   - embedded software;
   - compilation;
   - Java byte code;
   - Java virtual machine;
   - integrated development environment;

2. Explain how Java programs are compiled and run.
3. Describe two different ways of adding comments to a Java program.
4. What is the difference between using `System.out.println` and `System.out.print` to produce output in Java?
5. What, precisely, would be the output of the following programs?

(a)

```
public class Question5A
{
       public static void main(String[] args)
       {
           System.out.print("Hello, how are you? ");
           System.out.println("Fine thanks.");
       }
}
```

(b)

```
public class Question5B
{
       public static void main(String[] args)
       {
           System.out.println("Hello, how are you? ");
           System.out.println("Fine thanks.");
       }
}
```

(c)

```
public class Question5C
{
        public static void main(String[] args)
        {
                System.out.println("1 + 2 " + "+ 3" + " = 6");
        }
}
```

6. Identify the syntax errors in the following program:

```
public class
{
        public Static void main(String[] args)
        {
            system.out.println( I want this program to compile)
        }
```

1.10 Programming Exercises

1. If you do not have access to a Java IDE go to the accompanying website and follow the instructions for installing an IDE. You will also find instructions on the website for compiling and running programs.
2. Type and compile the *Hello World* program. If you make any syntax errors, the compiler will indicate where to find them. Correct them and re-compile your program. Keep doing this until you no longer have any errors. You can then run your program.
3. Make the changes to the *Hello World* program that are made in this chapter, then each time re-compile and run the program again.
4. Type and compile the program given in self test question 6 above. This program contained compiler errors that you should have identified in your answer to that question. Take a look at how the compiler reports on these errors then fix them so that the program can compile and run successfully.
5. Write a program that displays your name, address and telephone number, each on separate lines.
6. Adapt the above program to include a blank line between your address and telephone number.

7. Write a program that displays your initials in big letters made of asterisks. For example:

```
    *       *     *
   * *      *   *
  *****     * *
 *     *    *   *
*       *   *     *
```

   Do this by using a series of `println` commands, each printing one row of asterisks.

Check for updates

2 Building Blocks

Outcomes:

By the end of this chapter you should be able to:

- *distinguish between the eight built-in* ***primitive types*** *of Java*;
- ***declare*** *and* ***assign*** *values to* ***variables***;
- *create* ***constant*** *values with the keyword* **final**;
- *use the input methods of the* Scanner *class to get data from the keyboard*;
- *design the functionality of a method using* ***pseudocode***.

2.1 Introduction

The *Hello world* program that we developed in Chap. 1 is of course very simple indeed. One way in which this program is very limited is that it has no *data* to work on. All interesting programs will have to store data in order to give interesting results; what use would a calculator be without the numbers the user types into add and multiply? For this reason, one of the first questions you should ask when learning any programming language is "what types of data does this language allow me to store in my programs?"

2.2 Simple Data Types in Java

We begin this topic by taking a look at the basic types available in the Java language. The types of value used within a program are referred to as **data types**. If you wish to record the *price* of a cinema ticket in a program, for example, this value would probably need to be kept in the form of a **real number** (a number with a

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_2

Table 2.1 The primitive types of Java

Java type	Allows for	Range of values
byte	Very small integers	−128 to 127
short	Small integers	−32,768 to 32,767
int	Big integers	−2,147,483,648 to 2,147,483,647
long	Very big integers	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	Real numbers	$\pm 1.4 * 10^{-45}$ to $3.4 * 10^{38}$
double	Very big real numbers	$\pm 4.9 * 10^{-324}$ to $1.8 * 10^{308}$
char	Characters	Unicode character set
boolean	True or false	Not applicable

decimal point in it). However, if you wished to record *how many* tickets have been sold you would need to keep this in the form of an **integer** (whole number). It is necessary to know whether suitable types exist in the programming language to keep these items of data.

In Java there are a few simple data types that programmers can use. These simple types are often referred to as the **primitive types** of Java; they are also referred to as the **scalar types**, as they relate to a single piece of information (a single real number, a single character etc.).

Table 2.1 lists the names of these types in the Java language, the kinds of value they represent, and the exact range of these values.

As you can see, some kinds of data, namely integers and real numbers, can be kept as more than one Java type. For example, you can use the **byte** type, the **short** type or the **int** type to hold integers in Java. However, while each numeric Java type allows for both positive and negative numbers, *the maximum size of numbers that can be stored varies from type to type*.

For example, the type **byte** can represent integers ranging only from −128 to 127, whereas the type **short** can represent integers ranging from −32,768 to 32,767. Unlike some programming languages, these ranges are *fixed* no matter which Java compiler or operating system you are using.

The character type, **char**, is used to represent characters from a standard set of characters known as the **Unicode** character set. This contains nearly all the characters from most known languages. For the sake of simplicity, you can think of this type as representing any character that can be input from your keyboard.

Finally, the **boolean** type is used to keep only one of two possible values: **true** or **false**. This type can be useful when creating tests in programs. For example, the answer to the question "have I passed my exam?" will either be either *yes* or *no*. In Java a **boolean** type could be used to keep the answer to this question, with the value **true** being used to represent *yes* and the value **false** to represent *no*.

2.3 Declaring Variables in Java

The data types listed in Table 2.1 are used in programs to create named locations in the computer's memory that will contain values while a program is running. This process is known as **declaring**. These named locations are called **variables** because their values are allowed to *vary* over the life of the program.

For example, a program written to develop a computer game might need a piece of data to record the player's score as secret keys are found in a haunted house. The value held in this piece of data will vary as more keys are found. This piece of data would be referred to as a variable. To create a variable in your program you must:

- give that variable a name (of your choice);
- decide which data type in the language best reflects the kind of values you wish to store in the variable.

What name might you choose to record the score of the player in our computer game?

The rules for naming variables are the same as those we met when discussing the rules for naming classes in the previous chapter. However, the convention in Java programs is to begin the name of a variable with a *lower case* letter (whereas the convention is to start class names with an upper case letter). We could just pick a name like `x`, but it is best to pick a name that describes the purpose of the item of data; an ideal name would be `score`.

Which data type in Table 2.1 should you use if you wish to record a player's score? Well, since the score would always be a whole number, an integer type would be appropriate. There are four Java data types that can be used to hold integers (**`byte`**, **`short`**, **`int`** and **`long`**). As we said before, the only difference among these types is the range of values that they can keep. Unless there is specific reason to do otherwise, however, the **`int`** type is normally chosen to store integer values in Java programs. Similarly, when it comes to storing real numbers we will choose the **`double`** type rather than the **`float`** type.

Once the name and the type have been decided upon, the variable is **declared** as follows:

```
dataType variableName;
```

where `dataType` is the chosen primitive type and `variableName` is the chosen name of the variable. So, in the case of a player's score, the variable would be declared as follows:

```
int score;
```

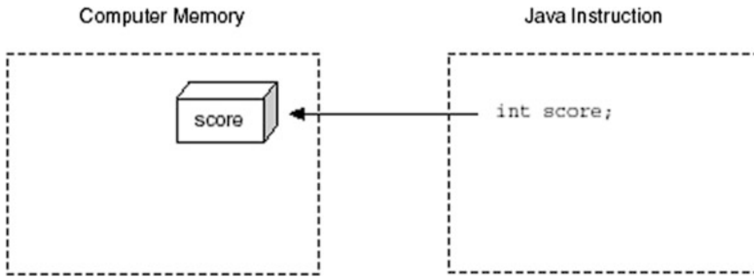


Fig. 2.1 The effect of declaring a variable in Java

Figure 2.1 illustrates the effect of this instruction on the computer's memory. As you can see, a small part of the computer's memory is set aside to store this item. You can think of this reserved space in memory as being a small box, big enough to hold an integer. The name of the box will be score.

In this way, many variables can be declared in your programs. Let's assume that the player of a game can choose a difficulty level (A, B, or C); another variable could be declared in a similar way.

What name might you give this variable? An obvious choice would be `difficulty level` but remember names cannot have spaces in them. You could use an underscore to remove the space (`difficulty_level`) or start the second word with a capital letter to distinguish the two words (`difficultyLevel`). Both are well-established naming conventions in Java. Alternatively you could just shorten the name to, say, *level*; that is what we will do here.

Now, what data type in Table 2.1 best represents the difficulty level? Since the levels are given as characters (A, B and C) the **`char`** type would be the obvious choice. At this point we have two variables declared: one to record the score and one to record the difficulty level.

```
int score;
char level;
```

Finally, several variables can be declared on a *single line* if they are *all of the same type*. For example, let's assume that there are ghosts in the house that hit out at the player; the number of times a player gets hit by a ghost can also be recorded. We can call this variable `hits`. Since the type of this variable is also an integer, it can be declared along with `score` in a single line as follows:

```
int score, hits; // two variables declared at once
char level ; // this has to be declared separately
```

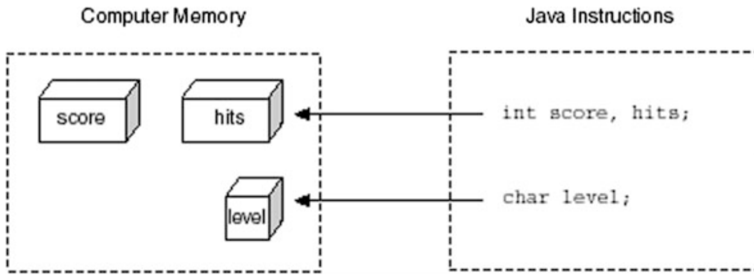


Fig. 2.2 The effect of declaring many variables in Java

Figure 2.2 illustrates the effect of these three declarations on the computer's memory.

Notice that the character box, *level*, is half the size of the integer boxes *score* and *hits*. That is because, in Java, the **`char`** type requires half the space of the **`int`** type. You should also be aware that the **`double`** type in Java requires twice the space of the **`int`** type.

You're probably wondering: if declaring a variable is like creating a box in memory, how do I put values into this box? The answer is with assignments.

2.4 Assignments in Java

Assignments allow values to be put into variables. They are written in Java with the use of the equality symbol (=). In Java this symbol is known as the **assignment operator**. Simple assignments take the following form:

```
variableName = value;
```

For example, to put the value zero into the variable score, the following assignment statement could be used:

```
score = 0;
```

This is to be read as "*set* the value of `score` *to* zero" or alternatively as "`score` *becomes equal to* zero". Effectively, this puts the number zero into the box in memory we called `score`. If you wish, you may combine the assignment statement with a variable declaration to put an initial value into a variable as follows:

```
int score = 0;
```

This is equivalent to the two statements below:

```
int score;
score = 0;
```

Although in some circumstances Java will automatically put initial values into variables when they are declared, this is not always the case and it is better explicitly to initialize variables that require an initial value.

Notice that the following declaration will not compile in Java:

```
int score = 2.5;
```

Can you think why?

The reason is that the right-hand side of the assignment (2.5) is a *real* number. This value could not be placed into a variable such as score, which is declared to hold only integers, without some information loss. In Java, such information loss is not permitted, and this statement would therefore cause a compiler error.

You may be wondering if it is possible to place a whole number into a variable declared to hold real numbers. The answer is yes. The following is perfectly legal:

```
double someNumber = 1000;
```

Although the value on the right-hand side (1000) appears to be an integer, it can be placed into a variable of type **double** because this would result in no information loss. Once this number is put into the variable of type **double**, it will be treated as the real number 1000.0.

Clearly, you need to think carefully about the best data type to choose for a particular variable. For instance, if a variable is going to be used to hold whole numbers *or* real numbers, use the **double** type as it can cope with both. If the variable is only ever going to be used to hold whole numbers, however, then although the **double** type might be adequate, use the **int** type as it is specifically designed to hold whole numbers.

When assigning a value to a character variable, you must enclose the value in single quotes. For example, to set the initial difficulty level to A, the following assignment statement could be used:

```
char level = 'A';
```

Remember: you need to declare a variable only once. You can then assign values to it as many times as you like. For example, later on in the program the difficulty level might be changed to a different value as follows:

```
char level = 'A'; // initial difficulty level
// other Java instructions
level = 'B'; // difficulty level changed
```

2.5 Creating Constants

There will be occasions where data items in a program have values *that do not change*. The following are examples of such items:

- the maximum score in an exam (100);
- the number of hours in a day (24);
- the mathematical value of π (approximately 3.1416).

In these cases the values of the items do not vary. Values that remain constant throughout a program (as opposed to variable) should be named and declared as **constants**.

Constants are declared much like variables in Java except that they are preceded by the keyword **`final`**. Once they are given a value, then that value is fixed and cannot later be changed. Normally we fix a value when we initialize the constant. For example:

```
final int HOURS = 24;
```

Notice that the standard Java convention has been used here of naming constants in upper case. Any attempt to change this value later in the program will result in a compiler error. For example:

```
final int HOURS = 24; // create constant
HOURS = 12; // will not compile!
```

2.6 Arithmetic Operators

Rather than just assign simple values (such as 24 and 2.5) to variables, it is often useful to carry out some kind of arithmetic in assignment statements. Java has the four familiar arithmetic operators, plus a remainder operator, for this purpose. These operators are listed in Table 2.2.

Table 2.2 The arithmetic operators of Java

Operation	Java operator
Addition	+
Subtraction	–
Multiplication	*
Division	/
Remainder	%

You can use these operators in assignment statements, much like you might use a calculator. For example, consider the following instructions:

```
int x;
x = 10 + 25;
```

After these instructions the variable x would contain the value 35: the result of adding 10 to 25. Terms on the right-hand side of assignment operators (like 10 + 25) that have to be *worked out* before they are assigned are referred to as **expressions**. These expressions can involve more than one operator.

Let's consider a calculation to work out the price of a product after a sales tax has been added. If the initial price of the product is 500 and the rate of sales tax is 17.5%, the following calculation could be used to calculate the total cost of the product:

```
double cost;
cost = 500 * (1 + 17.5/100);
```

After this calculation the final cost of the product would be 587.5.

By the way, in case you are wondering, the order in which expressions such as these are evaluated is the same as in arithmetic: terms in brackets are calculated first, followed by division and multiplication, then addition and subtraction. This means that the term in the bracket (1 + 17.5/100) evaluates to 1.175, not 0.185, as the division is calculated before the addition. The final operator (%) in Table 2.2 returns the remainder after *integer division* (this is often referred to as the **modulus**). Table 2.3 illustrates some examples of the use of this operator together with the values returned.

As an illustration of the use of both the division operator and the modulus operator, consider the following example.

Table 2.3 Examples of the modulus operator in Java

Expression	Value
29 % 9	2
6 % 8	6
40 % 40	0
10 % 2	0

A large party of 30 people is going to attend a school reunion. The function room will be furnished with a number of tables, each of which seats four people.

To calculate how many tables of four are required, and how many people will be left over, the division and modulus operators could be used as follows:

```
int tablesOfFour, peopleLeftOver;
tablesOfFour = 30/4; // number of tables
peopleLeftOver = 30%4; // number of people left over
```

After these instructions the value of `tablesOfFour` will be 7 (the result of dividing 30 by 4) and the value of `peopleLeftOver` will be 2 (the remainder after dividing 30 by 4). You may be wondering why the calculation for `tablesOfFour` (30/4) did not yield 7.5 but 7. The reason for this is that there are, in fact, two different in-built division routines in Java, one to calculate an integer answer and another to calculate the answer as a real number.

Rather than having two division operators, however, Java has a single division symbol (/) to represent *both* types of division. The division operator is said to be **overloaded**. This means that the same operator (in this case the division symbol) can behave in different ways. This makes life much easier for programmers as the decision about which routine to call is left to the Java language.

How does the Java compiler know which division routine we mean? Well, it looks at the values that are being divided. If *at least one value* is a real number (as in the product cost example), it assumes we mean the division routine that calculates an answer as a real number, otherwise it assumes we mean the division routine that calculates an answer as a whole number (as in the reunion example).[1]

2.7 Expressions in Java

So far, variable names have appeared only on the left-hand side of assignment statements. However, the expression on the right-hand side of an assignment statement can itself contain variable names. If this is the case then the name does not refer to *the location*, but to *the contents of the location*. For example, the assignment to calculate the cost of the product could have been re-written as follows:

```
double price, tax, cost; // declare three variables
price = 500; // set price
tax = 17.5; // set tax rate
cost = price * (1 + tax/100); // calculate cost
```

[1]To force the use of one division routine over another, a technique known as **type casting** can be used. We will return to this technique in later chapters.

Here, the variables price and tax that appear in the expression

```
price * (1 + tax/100)
```

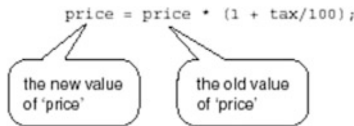
are taken to mean *the values contained in* price and tax respectively. This expression evaluates to 587.5 as before. Notice that although this price happens to be a whole number, it has been declared to be a **`double`** as generally prices are expressed as real numbers.

There is actually nothing to stop you using the name of the variable you are assigning to in the expression itself. This would just mean that the old value of the variable is being used to calculate its new value. Rather than creating a new variable, `cost`, to store the final cost of the product, the calculation could, for example, have updated *the original price* as follows:

```
price = price * (1 + tax/100);
```

Now only two variables are required, `price` and `tax`. Let's look at this assignment a bit more closely.

When reading this instruction, the price in the right-hand expression is to be read as the *old value* of price, whereas the price on the left-hand side is to be read as *the new value* of price.



You might be wondering what would happen if we used a variable in the right hand side of an expression before it had been given a value. For example, look at this fragment of code:

```
double price = 500;
double tax;
cost = price * (1 + tax/100);
```

The answer is that you would get a compiler error telling you that you were trying to use a variable before it has been initialized.

You will find that one very common thing that we have to do in our programs is to increase (or *increment*) an integer by 1. For example, if a variable `x` has been declared as an **`int`**, then the instruction for incrementing `x` would be:

```
x = x + 1;
```

In fact, this is so common that there is a special shorthand for this instruction, namely:

```
x++;
```

The '++' is therefore known as the increment operator. Similarly there exists a decrement operator, '− −'.

Thus:

```
x− −;
```

is shorthand for:

```
x = x − 1;
```

It is possible to use the increment and decrement operators in expressions. We will show you a couple of examples of this here, as you might easily come across them in other texts. However, we will not be using this technique in the remainder of this book, because we think it can sometimes be confusing for new programmers. If `x` and `y` are **`ints`**, the expression:

```
y = x++;
```

means assign the value of `x` to `y`, then increment `x` by 1.

However the expression:

```
y = ++x;
```

means increment `x` by 1, then assign this new value to `y`. The decrement operator can be used in the same way.

While we are on the subject of shortcuts, there is one more that you might come across in other places, but which, once again, we won't be using in this text:

```
y += x;
```

is shorthand for:

```
y = y + x;
```

The code fragments we have been writing so far in this chapter are, of course, not complete programs. As you already know, to create a program in Java you must write one or more classes. In the example that follows, we write a class, `Find-Cost`, where the main method calculates the price of the product.

FindCost

```
// a program to calculate the cost of a product after a sales tax has been added

public class FindCost
{
     public static void main(String[] args)
     {
          double price, tax;
          price = 500;
          tax = 17.5;
          price = price * (1 + tax/100);
     }
}
```

What would you see when you run this program? The answer is nothing! There is no instruction to display the result on to the screen. You have already seen how to display messages onto the screen. It is now time to take a closer look at the output command to see how you can also display results onto the screen.

2.8 More About Output

As well as displaying messages, Java also allows any values or expressions of the primitive types that we showed you in Table 2.1 to be printed on the screen using the same output commands. It does this by implicitly converting each value/expression to a string before displaying it on the screen. In this way numbers, the value of variables, or the value of expressions can be displayed on the screen. For example, the square of 10 can be displayed on the screen as follows:

```
System.out.print(10*10);
```

This instruction prints the number 100 on the screen. Since these values are converted into strings by Java they can be joined onto literal strings for output.

For example, let's return to the party of 30 people attending their school reunion that we discussed in Sect. 2.6. If each person is charged a fee of 7.50 for the evening, the total cost to the whole party could be displayed as follows:

```
System.out.print("cost = " + (30*7.5) );
```

Here the concatenation operator (+), is being used to join the string, "cost = ", onto the value of the expression, (30 * 7.5). Notice that when expressions like 30 * 7.5 are used in output statements it is best to enclose them in brackets. This would result in the following output:

```
cost = 225.0
```

Bear these ideas in mind and look at the next version of FindCost, which we have called FindCost2; the program has been re-written so that the output is visible.

FindCost2

```
// a program to calculate and display the cost of a product after sales tax has been added

public class FindCost2
{
     public static void main(String[] args)
     {
          double price, tax;
          price = 500;
          tax = 17.5;
          price = price * (1 + tax/100); // calculate cost
          // display results
          System.out.println("*** Product Price Check ***");
          System.out.println("Cost after tax = " + price);
     }
}
```

This program produces the following output:

```
*** Product Price Check ***
Cost after tax = 587.5
```

Although being able to see the result of the calculation is a definite improvement, this program is still very limited. The formatting of the output can certainly be improved, but we shall not deal with such issues until later on in the book. What does concern us now is that this program can only calculate the cost of products when the sales tax rate is 17.5% and the initial price is 500!

What is required is not to fix the rate of sales tax or the price of the product but, instead, to get the *user of your program* to *input* these values as the program runs.

2.9 Input in Java: The *Scanner* Class

Java provides a special class called Scanner, which makes it easy for us to write a program that obtains information that is typed in at the keyboard. Scanner is provided as part of what is known, in Java, as a **package**. A package is a collection of pre-compiled classes—lots more about that in the second semester! The Scanner class is part of a package called util. In order to access a package we use a command called **import**. So, to make the Scanner class accessible to the compiler we have to tell it to look in the util package, and we do this by placing the following line at the top of our program:

```
import java.util.Scanner;
```

Sometimes you might come across an **import** statement that looks like this:

```
import java.util.*;
```

This asterisk means that all the classes in the particular package are made available to the compiler. Although using the asterisk notation is perfectly acceptable, nowadays it is considered better practice to specify only those classes that we need, as in the first statement, as this clarifies precisely which classes are being used within a program—so that is what we will do in this text.

As long as the Scanner class is accessible, you can use all the input methods that have been defined in this class. We are going to show you how to do this now. Some of the code might look a bit mysterious to you at the moment, but don't worry about this right now. Just follow our instructions for the time being—after a few chapters, it will become clear to you exactly why we use the particular format and syntax that we are showing you.

Having imported the Scanner class, you will need to write the following instruction in your program:

```
Scanner keyboard = new Scanner(System.in);
```

What we are doing here is creating an object, keyboard, of the Scanner class. Once again, the true meaning of the phrase *creating an object* will become clear in the next few chapters, so don't worry too much about it now. However, you should know that, in Java, System.in represents the keyboard, and by associating our Scanner object with System.in, we are telling it to get the input from the keyboard as opposed to a file on disk or a modem for example. Just to note that, like a variable, you can choose any name for this object, but we have chosen the obvious name here—keyboard.

The Scanner class has several input methods, each one associated with a different input type, and once we have declared a Scanner object we can use these methods. Let's take some examples. Say we wanted a user to type in an integer at the keyboard, and we wanted this value to be assigned to an integer variable called x. We would use the Scanner method called nextInt; the instruction would look like this:

```
x = keyboard.nextInt();
```

In the case of a **double**, y, we would do this:

```
y = keyboard.nextDouble();
```

Notice that to access a method of a class you need to join the name of the method (getInt or getDouble) to the name of the object (keyboard) by using the full-stop. Also you must remember the brackets after the name of the method.

What about a character? Unfortunately this is a little bit more complicated, as there is no nextChar method provided. Assuming c had been declared as a character, we would have to do this:

```
c = keyboard.next().charAt(0);
```

You won't understand exactly why we use this format until Chap. 7—for now just accept it and use it when you need to.

Let us return to the haunted house game to illustrate this. Rather than *assigning* a difficulty level as follows:

```
char level;
level = 'A';
```

you could take a more flexible approach by asking the user of your program to *input* a difficulty level while the program runs. Since level is declared to be a character variable, then, after declaring a Scanner object, keyboard, you could write this line of code:

```
level = keyboard.next().charAt(0);
```

Some of you might be wondering how we would get the user to type in strings such as a name or an address. This is a bit more difficult, because a string is not a simple type like an **int** or a **char**, but contains many characters. In Java a String is not a simple data type but a class—so to do this you will have to wait until Chap. 7 where we will study classes and objects in depth.

Let us re-write our previous program that calculated the cost of an item after tax; this time the price of the product and the rate of sales tax are not fixed in the program, but are input from the keyboard. Since the type used to store the price and the tax is a **double**, the appropriate input method is nextDouble, as can be seen below.

FindCost3

```
import java.util.Scanner; // import the Scanner class from the util package

/* a program to input the initial price of a product and then calculate and display its cost after tax
has been added */

public class FindCost3
{
     public static void main(String[] args )
     {
          Scanner keyboard = new Scanner(System.in); // create Scanner object
          double price, tax;
          System.out.println("*** Product Price Check ***");
          System.out.print("Enter initial price: "); // prompt for input
          price = keyboard.nextDouble(); // input method called
          System.out.print("Enter tax rate: "); // prompt for input
          tax = keyboard.nextDouble(); // input method called
          price = price * (1 + tax/100); // perform the calculation
          System.out.println("Cost after tax = " + price);
     }
}
```

Note that, by looking at this program code alone, there is no way to determine what the final price of the product will be, as the initial price and the tax rate will be determined *only when the program is run*.

Let's assume that we run the program and the user interacts with it as follows[2]:

```
*** Product Price Check ***
Enter initial price: 1000
Enter tax rate: 12.5
Cost after tax = 1125.0
```

You should notice the following points from this test run:

- whatever the price of the computer product and the rate of tax, this program could have evaluated the final price;
- entering numeric values with additional formatting information, such as currency symbols or the percentage symbol, is not permitted;
- after an input method is called, the cursor always moves to the next line.

The programs we are looking at now involve input commands, output commands and assignments. Clearly, the order in which you write these instructions affects the results of your programs. For example, if the instructions to calculate the final price and then display the results were reversed as follows:

```
System.out.println("Cost after tax = " + price);
price = price * (1 + tax/100);
```

[2]We have used ***bold italic*** font to represent user input.

the price that would be displayed would not be the price *after* tax but the price *before* tax! In order to avoid such mistakes it makes sense *to design your code* by sketching out your instructions before you type them in.

2.10 Program Design

Designing a program is the task of considering exactly *how to build* the software, whereas writing the code (the task of *actually building* the software) is referred to as *implementation*. As programs get more complex, it is important to spend time on program design, before launching into program implementation.

As we have already said, Java programs consist of one or more classes, each with one or more methods. In later chapters we will introduce you to the use of diagrams to help design such classes. The programs we have considered so far, however, have only a single class and a single method (`main`), so a class diagram would not be very useful here. We will therefore return to this design technique as we develop larger programs involving many classes.

At a lower level, it is the instructions *within* a method that determine the *behaviour* of that method. If the behaviour of a method is complex, then it will also be worthwhile spending time on designing the instructions that make up the method. When you sketch out the code for your methods, you don't want to have to worry about the finer details of the Java compiler such as declaring variables, adding semi-colons and using the right brackets. Very often a general purpose "coding language" can be used for this purpose to convey the meaning of each instruction without worrying too much about a specific language syntax.

Code expressed in this way is often referred to as **pseudocode**. The following is an example of pseudocode that could have been developed for the main method of `FindCost3` program:

```
BEGIN
  DISPLAY program title
  DISPLAY prompt for price
  ENTER price
  DISPLAY prompt for tax
  ENTER tax
  SET price TO price * (1 + tax/100)
  DISPLAY new price
END
```

Note that these pseudocode instructions are not intended to be typed in and compiled as they do not meet the syntax rules of any particular programming language. So, exactly how you write these instructions is up to you: there is no fixed syntax for them. However, each instruction conveys a well-understood programming concept and can easily be translated into a given programming language. When you read these instructions you should be able to see how each line would be coded in Java.

Wouldn't it be much easier to write your main method if you have pseudocode like this to follow? In future, when we present complex methods to you we will do so by presenting their logic using pseudocode.

2.11 Self-test Questions

1. What would be the most appropriate Java data type to use for the following items of data?

   - the maximum number of people allowed on a bus;
   - the weight of a food item purchased in a supermarket;
   - the grade awarded to a student (for example 'A', 'B' or 'C').

2. Explain which, if any, of the following lines would result in a compiler error:

```
int x = 75.5;
double y = 75;
```

3. Which of the following would be valid names for a variable in Java?

   - `ticket`
   - `cinema ticket`
   - `cinemaTicket`
   - `cinema_ticket`
   - `void`
   - `Ticket`

4. Identify and correct the errors in the program below, which prompts for the user's age and then attempts to work out the year in which the user was born.

```
import java.util.Scanner;

public class SomeProg
{
     public static void main (String[] args)
     {
          Scanner keyboard = new Scanner(System.in);
          final int YEAR;
          int age, bornIn;
          System.out.print(How old are you this year? );
          age = keyboard.nextDouble();
          bornIn = YEAR - age;
          System.out.println("I think you were born in " + BornIn);
     }
}
```

5. What is the final value of z in the following program?

```
public class SomeProg
{
   public static void main (String[] args)
   {
      int x, y, z;
      x = 5;
      y = x + 2;
      x = 10;
      z = y * x;
   }
}
```

6. What would be the final output from the program below if the user entered the number 10?

```
import java.util.Scanner;

public class Calculate
{
     public static void main(String[] args )
     {
         Scanner keyboard = new Scanner(System.in);
         int num1, num2;
         num2 = 6;
         System.out.print("Enter value ");
         num1 = keyboard.nextInt();
         num1 = num1 + 2;
         num2 = num1 / num2;
         System.out.println("result = " + num2);
     }
}
```

7. Use pseudocode to design a program that asks the user to enter values for the length and height of a rectangle and then displays the area and perimeter of that rectangle.
8. The program below was written in an attempt to swap the value of two variables. However it does not give the desired result:

```
/* This program attempts to swap the value of two variables - it doesn't give the desired
result however! */

import java.util.Scanner;

public class SwapAttempt
{

      public static void main(String[] args)
      {
         // declare variables
         int x, y;
         // enter values
         System.out.print("Enter value for x ");
         x = keyboard.nextInt();
         System.out.print("Enter value for y ");
         y = keyboard.nextInt();

         // code attempting to swap two variables
         x = y;
         y = x;

         //display results
         System.out.println("x = " + x);
         System.out.println("y = " + y);
      }
}
```

(a) Can you see why the program doesn't do what we hoped?
(b) What would be the actual output of the program?
(c) How could you modify the program above so that the values of the two variables are swapped successfully?

2.12 Programming Exercises

1. Implement the `FindCost3` program from this chapter.

2. Implement the programs from self-test questions 4, 5, 6 and 8 above in order to verify your answers to those questions.

3. Implement the rectangle program that you designed in self-test question 7.

4. The following pseudocode has been arrived at for a program that converts pounds to kilos (1 kilo = 2.2 lb).

```
BEGIN
   PROMPT for value in pounds
   ENTER value in pounds
   SET value to old value ÷ 2.2
   DISPLAY value in kilos
END
```

   Implement this program, remembering to declare any variables that are necessary.

5. An individual's Body Mass Index (BMI) is a measure of a person's weight in relation to their height. It is calculated as follows:

   - divide a person's weight (in kg) by the square of their height (in meters)
     Design and implement a program to allow the user to enter their weight and height and then print out their BMI.

6. A group of students has been told to get into teams of a specific size for their coursework. Design and implement a program that prompts for the number of students in the group and the size of the teams to be formed, and displays how many teams can be formed and how many students are left without a team.

7. Design and implement a program that asks the user to enter a value for the radius of a circle, then displays the area and circumference of the circle.

   Note that the area is calculated by evaluating πr^2 and the circumference by evaluating $2\pi r$. You can take the value of π to be 3.1416—and ideally you should declare this as a constant at the start of the program.[3]

[3]Of course you will not be able to use the Greek letter π as a name for a variable or constant. You will need to give it a name like PI.

Selection

3

Outcomes:

By the end of this chapter you should be able to:

- *explain the difference between* ***sequence*** *and* ***selection***;
- *use an* **if** *statement to make a single choice in a program*;
- *use an* **if**…**else** *statement to make a choice between two options in a program*;
- *use nested* **if**…**else** *statements to make multiple choices in a program*;
- *use a* **switch** *statement to make multiple choices in a program.*

3.1 Introduction

One of the most rewarding aspects of writing and running a program is knowing that *you* are the one who has control over the computer. But looking back at the programs you have already written, just how much control do you actually have? Certainly, it was you who decided upon which instructions to include in your programs but *the order in which these instructions were executed* was *not* under your control. These instructions were always executed in **sequence**, that is one after the other, from the beginning to the end of the main method. You will soon find that there are numerous instances when this order of execution is too restrictive and you will want to have much more control over the order in which instructions are executed.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_3

3.2 Making Choices

Very often you will want your programs to make *choices* among different courses of action. For example, a program processing requests for airline tickets could have the following choices to make:

- display the price of the seats requested;
- display a list of alternative flights;
- display a message saying that no flights are available to that destination.

A program that can make choices can behave *differently* each time it is run, whereas programs in which instructions are just executed in sequence behave the *same way* each time they are run.

As we have already mentioned, unless you indicate otherwise, program instructions are always executed in sequence. **Selection**, however, is a method of program control in which a choice can be made about which instructions to execute.

For example, consider the following program, which welcomes customers queuing up for a roller-coaster ride:

RollerCoaster

```
import java.util.Scanner;

public class RollerCoaster
{
    public static void main(String[] args)
    {
         // declare variables
        int age;
        Scanner keyboard = new Scanner (System.in);

        // four instructions to process information
        System.out.println("How old are you?");
        age = keyboard.nextInt();
        System.out.println("Hello Junior!");
        System.out.println("Enjoy your ride");
    }
}
```

As you can see, following the variable declarations, there are *four* remaining instructions in this program. Remember that at the moment these instructions will be executed in sequence, from top to bottom. Consider the following interaction with this program:

```
How old are you?
10
Hello Junior!
Enjoy your ride
```

This looks fine but the message "Hello Junior!" is only meant for children. Now let's assume that someone older comes along and interacts with this program as follows:

How old are you?
45
Hello Junior!
Enjoy your ride

The message “Hello Junior!”, while flattering, might not be appropriate in this case! In other words, it is not always appropriate to execute the following instruction:

```
System.out.println("Hello Junior!");
```

What is required is a way of deciding (while the program is running) whether or not to execute this instruction. In effect, this instruction needs to be *guarded* so that it is only executed *when appropriate*. Assuming we define a child as someone under 13 years of age, we can represent this in pseudocode as follows:

```
DISPLAY "How old are you?"
ENTER age
IF age is under 13
BEGIN
   DISPLAY "Hello Junior!"
END
DISPLAY "Enjoy your ride"
```

In the above, we have emboldened the lines that have been added to guard the “Hello Junior!” instruction. The emboldened lines are not to be read as *additional* instructions; they are simply a means to *control the flow* of the *existing* instructions. The emboldened lines say, in effect, that the instruction to display the message “Hello Junior!” should only be executed if the age entered is under 13.

This, then, is an example of the form of control known as selection. Let’s now look at how to code this selection in Java.

3.3 The ‘if’ Statement

The particular form of selection discussed above is implemented by making use of Java’s **if** statement. The general form of an **if** statement is given as follows:

```
if ( /* a test goes here */ )
{
      // instruction(s) to be guarded go here
}
```

As you can see, the instructions to be guarded are placed inside the braces of the **if** statement. A **test** is associated with the **if** statement. A test is any expression that produces a result of **true** or **false**. For example **x** > **100** is a test as it is an expression that either gives an answer of **true** or **false** (depending upon the value of x). We call an expression that returns a value of **true** or **false** a **boolean** expression, as **true** and **false** are **boolean** values. Examples of tests in everyday language are:

- this password is valid;
- there is an empty seat on the plane;
- the temperature in the laboratory is too high.

The test must follow the **if** keyword and be placed in round brackets. When the test gives a result of **true** the instructions inside the braces of the **if** statement are executed. The program then continues by executing the instructions after the braces of the **if** statement as normal. If, however, the **if** test gives a result of **false** the instructions inside the **if** braces are *skipped* and not executed.

We can rewrite the RollerCoaster program by including an appropriate **if** statement around the "Hello Junior!" message with the test (age < 13) as follows:

RollerCoaster2

```
import java.util.Scanner;

// This program is an example of the use of selection in a Java program

public class RollerCoaster2
{
    public static void main(String[] args)
    {
      int age;
      Scanner keyboard = new Scanner (System.in);
      System.out.println("How old are you?");
      age = keyboard.nextInt();
      if (age < 13) // test controls if the next instruction is executed
      {
           System.out.println("Hello Junior!");
      }
      System.out.println("Enjoy your ride");
    }
}
```

Now the message "Hello Junior!" will only be executed if the test (age < 13) is **true**, otherwise it will be skipped (see Fig. 3.1).

Let's assume we run the above program with the same values entered as when running the previous version. First, the child approaches the ride:

```
How old are you?
10
Hello Junior!
Enjoy your ride
```

In this case, the **if** statement has allowed the "Hello Junior!" message to be displayed as the age entered is less than 13. Now the adult approaches the ride:

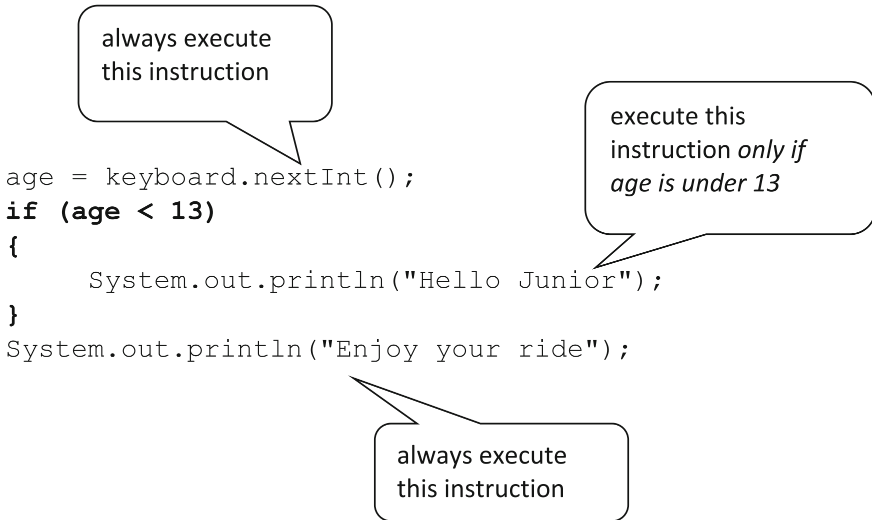


Fig. 3.1 The *if* statement allows a choice to be made in programs

```
How old are you?
45
Enjoy your ride
```

In this case the **`if`** statement has not allowed the given instruction to be executed as the associated test was not true. The message is skipped and the program continues with the following instruction to display “Enjoy your ride”.

In this program there was only a *single* instruction inside the **if** statement.

```
age = keyboard.nextInt();
if (age < 13)
{
    System.out.println("Hello Junior!"); // single instruction inside 'if'
}
System.out.println("Enjoy your ride");
```

When there is only a single instruction associated with an **if** statement, the braces can be omitted around this instruction, if so desired, as follows:

```
age = keyboard.nextInt();
if (age < 13)
System.out.println("Hello Junior!"); // braces can be omitted around this line
System.out.println("Enjoy your ride");
```

The compiler will always assume that the first line following the **`if`** test is the instruction being guarded. For clarity, however, we will always use braces around instructions.

3.3.1 Comparison Operators

In the example above, the “less than” operator (<) was used to check the value of the age variable. This operator is often referred to as a **comparison operator** as it is used to compare two values. Table 3.1 shows all the Java comparison operator symbols.

Since comparison operators give a **boolean** result of **true** or **false** they are often used in tests such as those we have been discussing. For example, consider a temperature variable being used to record the temperature for a particular day of the week. Assume that a temperature of 18° or above is considered to be a hot day. We could use the “greater than or equal to” operator ($\geq$) to check for this as follows:

```
if (temperature >= 18) // test to check for hot temperature
{
     // this line executed only when the test is true
     System.out.println("Today is a hot day!");
}
```

You can see from Table 3.1 that a double equals (= =) is used to check for equality in Java and not the single equals (=), which, as you know, is used for assignment. To use the single equals is a very common error! For example, to check whether an angle is a right angle the following test should be used:

```
if(angle == 90)// note the use of the double equals
{
     System.out.println("This IS a right angle");
}
```

To check if something is *not equal* to a particular value we use the exclamation mark followed by an equals sign (! =). So to test if an angle is *not* a right angle we can have the following:

```
if (angle != 90)
{
   System.out.println("This is NOT a right angle");
}
```

Table 3.1 The comparison operators of Java

Operator	Meaning
= =	Equal to
! =	Not equal to
<	Less than
>	Greater than
$\geq$	Greater than or equal to
$\leq$	Less than or equal to

3.3.2 Multiple Instructions Within an ‘if’ Statement

You have seen how an **if** statement guarding a single instruction may or may not be implemented with braces around the instruction. When *more than one* instruction is to be guarded by an **if** statement, however, the instructions *must* be placed in braces. As an example, consider once again the program we presented in the previous chapter that calculated the cost of a product.

FindCost3

```
import java.util.Scanner; // import the Scanner class from the util package

/* a program to input the initial price of a product and then calculate and display its cost after
tax has been added */

public class FindCost3
{
     public static void main(String[] args )
     {
          Scanner keyboard = new Scanner(System.in); // create Scanner object
          double price, tax;
          System.out.println("*** Product Price Check ***");
          System.out.print("Enter initial price: "); // prompt for input
          price = keyboard.nextDouble(); // input method called
          System.out.print("Enter tax rate: "); // prompt for input
          tax = keyboard.nextDouble(); // input method called
          price = price * (1 + tax/100); // perform the calculation
          System.out.println("Cost after tax = " + price);
     }
}
```

Now assume that a special promotion is in place for those products with an initial price over 100. For such products the company pays half the tax. The program below makes use of an **if** statement to apply this promotion, as well as informing the user that a tax discount has been applied. Take a look at it and then we will discuss it.

FindCostWithDiscount

```
import java.util.Scanner;

public class FindCostWithDiscount
{
    public static void main(String[] args )
    {
        double price, tax;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("*** Product Price Check ***");
        System.out.print("Enter initial price: ");
        price = keyboard.nextDouble();
        System.out.print("Enter tax rate: ");
        tax = keyboard.nextDouble();
        // the following 'if' statement allows a selection to take place
        if (price > 100) // test the price to see if a discount applies
        {
            // these two instructions executed only when the test is true
            System.out.println("Special Promotion: We pay half your tax!");
            tax = tax * 0.5;
        }
        // the remaining instructions are always executed
        price = price * (1 + tax/100);
        System.out.println("Cost after tax = " + price);
    }
}
```

Now, the user is still always prompted to enter the initial price and tax as before:

```
System.out.print("Enter initial price: ");
price = keyboard.nextDouble();
System.out.print("Enter tax rate: ");
tax = keyboard.nextDouble();
```

The next two instructions are then placed inside an **if** statement. This means they may not always be executed:

```
if (price > 100)
{
     System.out.println("Special Promotion: We pay half your tax!");
     tax = tax * 0.5;
}
```

Notice that if the braces were omitted in this case, only the *first* instruction would be taken to be inside the **`if`** statement—the second statement would not be guarded and so would *always* be executed!

With braces around both instructions, they will be executed only when the test (`price > 100`) returns a **`boolean`** result of **`true`**. So, for example, if the user had entered a price of 150 the discount would be applied; but if the user entered a price of 50 these instructions would not be executed and a discount would not be applied.

Regardless of whether or not the test was **`true`** and the instructions in the **`if`** statement executed, the program always continues with the remaining instructions:

```
price = price * (1 + tax/100);
System.out.println("Cost after tax = " + price);
```

Here is a sample program run when the test returns a result of **`false`** and the discount is not applied:

```
*** Product Price Check ***
Enter initial price: 20
Enter tax rate: 10
Cost after tax = 22.0
```

In this case the program appears to behave in exactly the same way as the original program. Here, however, is a program run when the test returns a result of **true** and a discount does apply:

```
*** Product Price Check ***
Enter initial price: 1000
Enter tax rate: 10
Special Promotion: We pay half your tax!
Cost after tax = 1050.0
```

3.4 The ‘if…else’ Statement

Using the **`if`** statement in the way that we have done so far has allowed us to build the idea of a choice into our programs. In fact, the **`if`** statement made one of two choices before continuing with the remaining instructions in the program:

- execute the conditional instructions, or
- do not execute the conditional instructions.

The second option amounts to “do nothing”. Rather than do nothing if the test is **`false`**, an extended version of an **`if`** statement exists in Java to state an *alternative* course of action. This extended form of selection is the **`if`**…**`else`** statement. As the name implies, the instructions to be executed if the test evaluates to **`false`** are preceded by the Java keyword **`else`** as follows:

```
if ( /* test goes here */ )
{
      // instruction(s) if test is true go here
}
else
{
     // instruction(s) if test is false go here
}
```

This is often referred to as a **double-branched** selection as there are two alternative groups of instructions, whereas a single **`if`** statement is often referred to as a **single-branched** selection. The program below, `DisplayResult`, illustrates the use of a double-branched selection.

DisplayResult

```
import java.util.Scanner;

public class DisplayResult
{
    public static void main(String[] args)
    {
        int mark;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What exam mark did you get? ");
        mark = keyboard.nextInt();
        if (mark >= 40)
        {
                // executed when test is true
                System.out.println("Congratulations, you passed");
        }
        else
        {
                // executed when test is false
                System.out.println("I'm sorry, but you failed");
        }
        System.out.println("Good luck with your other exams");
    }
}
```

This program checks a student’s exam mark and tells the student whether or not he or she has passed (gained a mark greater than or equal to 40), before displaying a good luck message on the screen. Let’s examine this program a bit more closely.

Prior to the **if**…**else** statement the following lines are executed in sequence:

```
int mark;
Scanner keyboard = new Scanner(System.in);
System.out.println("What exam mark did you get? ");
mark = keyboard.nextInt();
```

Then the following condition is tested as part of the **if**…**else** statement:

`(mark >= 40)`

When this test is **true** the following line is executed:

```
System.out.println("Congratulations, you passed");
```

When the test is **false**, however, the following line is executed *instead*:

```
System.out.println("I'm sorry, but you failed");
```

Finally, whichever path was chosen the program continues by executing the last line:

```
System.out.println("Good luck with your other exams");
```

The **if**…**else** form of control has allowed us to choose from *two* alternative courses of action. Here is a sample program run:

```
What exam mark did you get?
52
Congratulations, you passed
Good luck with your other exams
```

Here is another sample run where a different course of action is chosen.

```
What exam mark did you get?
35
I'm sorry, but you failed
Good luck with your other exams
```

3.5 Logical Operators

As we've already pointed out, the test in an **if** statement is an expression that produces a **boolean** result of **true** or **false**. Often it is necessary to join two or more tests together to create a single more complicated test.

As an example, consider a program that checks the temperature in a laboratory. Assume that, for the experiments in the laboratory to be successful, the temperature must remain between 5 and 12 °C. An **if** statement might be required as follows:

```
if ( /* test to check if temperature is safe */ )
{
     System.out.println ("TEMPERATURE IS SAFE!");
}
else
{
     System.out.println("UNSAFE: RAISE ALARM!!");
}
```

The test should check if the temperature is safe. This involves combining two tests together:

1. check that the temperature is greater than or equal to 5 (`temperature >= 5`)
2. check that the temperature is less than or equal to 12 (`temperature <= 12`)

Both of these tests need to evaluate to **true** in order for the temperature to be safe. When we require two tests to be true we use the following symbol to join the two tests:

```
&&
```

This symbol is read as "AND". So the correct test is:

```
if (temperature >= 5 && temperature <= 12)
```

Now, if the temperature were below 5 the first test (`temperature >= 5`) would evaluate to **false** giving a final result of **false**; the **if** statement would be skipped and the **else** statement would be executed:

```
UNSAFE: RAISE ALARM!!
```

If the temperature were greater than 12 the second part of the test (temperature <= 12) would evaluate to **false** also giving an overall result of **false** and again the **if** statement would be skipped and the **else** statement would be executed.

However, when the temperature is between 5 and 12 *both* tests would evaluate to **true** and the final result would be **true** as required; the **if** statement would then be executed instead:

Table 3.2 The logical operators of Java

Logical operator	Java counterpart
AND	&&
OR	||
NOT	!

```
TEMPERATURE IS SAFE!
```

Notice that the two tests must be *completely* specified as each needs to return a **boolean** value of **true** or **false**. It would be wrong to try something like the following:

```
// wrong! second test does not mention 'temperature'!
if (temperature >= 5 && <= 12)
```

This is wrong as the second test (<= 12) is not a legal **boolean** expression. Symbols that join tests together to form longer tests are known as **logical operators**. Table 3.2 lists the Java counterparts to the three common logical operators.

Both the AND and OR operators join two tests together to give a final result. While the AND operator requires both tests to be **true** to give a result of **true**, the OR operator requires only that *at least one* of the tests be **true** to give a result of **true**. The NOT operator flips a value of **true** to **false** and a value of **false** to **true**. Table 3.3 gives some examples of the use of these logical operators.

As an example of the use of the NOT operator (!), let us return to the temperature example. We said that we were going to assume that a temperature of greater than 18° was going to be considered a hot day. To check that the day is not a hot day we could use the NOT operator as follows:

```
if (!(temperature > 18) ) // test to check if temperature is not hot
{
     System.out.println("Today is not a hot day!");
}
```

Table 3.3 Logical operators: some examples

Expression	Result	Explanation
10 > 5 && 10 > 7	true	Both tests are true
10 > 5 && 10 > 20	false	The second test is false
10 > 15 && 10 > 20	false	Both tests are false
10 > 5 || 10 > 7	true	At least one test is true (in this case both tests are true)
10 > 5 || 10 > 20	true	At least one test is true (in this case just one test is true)
10 > 15 || 10 > 20	false	Both tests are false
! (10 > 5)	false	Original test is true
! (10 > 15)	true	Original test is false

Of course, if a temperature is not greater than 18° then it must be less than or equal to 18°. So, another way to check the test above would be as follows:

```
if (temperature <= 18) // this also checks if temperature is not hot
{
    System.out.println("Today is not a hot day!");
}
```

3.6 Nested 'if…else' Statements

Instructions within **`if`** and **`if`**…**`else`** statements can themselves be *any* legal Java commands. In particular they could contain other **`if`** or **`if`**…**`else`** statements. This form of control is referred to as **nesting**. Nesting allows multiple choices to be processed.

As an example, consider the following program, which asks a student to enter his or her tutorial group (A, B, or C) and then displays on the screen the time of the software lab.

Timetable

```
import java.util.Scanner;

public class Timetable
{
    public static void main(String[] args)
    {
        char group; // to store the tutorial group
        Scanner keyboard = new Scanner(System.in);
        System.out.println("***Lab Times***"); // display header
        System.out.println("Enter your group (A,B,C)");
        group = keyboard.next().charAt(0);

        // check tutorial group and display appropriate time
        if (group == 'A')
        {
            System.out.print("10.00 a.m"); // lab time for group A
        }
        else
        {
            if (group == 'B')
            {
                System.out.print("1.00 p.m"); // lab time for group B
            }
            else
            {
                if (group == 'C')
                {
                    System.out.print("11.00 a.m"); // lab time for group C
                }
                else
                {
                    System.out.print("No such group"); // invalid group
                }
            }
        }
    }
}
```

As you can see, nesting can result in code with many braces can become difficult to read, even with our careful use of tabs. Such code can be made easier to read by not including the braces associated with all the **else** branches, as in the second version of the timetable program shown below:

TimetableVersion2

```
import java.util.Scanner;

public class TimetableVersion2
{
    public static void main(String[] args)
    {
            char group; // to store the tutorial group
            Scanner keyboard = new Scanner(System.in);
            System.out.println("***Lab Times***"); // display header
            System.out.println("Enter your group (A,B,C)");
            group = keyboard.next().charAt(0);

            if (group == 'A')
            {
                  System.out.println("10.00 a.m");
            }
            else if (group == 'B')
            {
                  System.out.println("1.00 p.m");
            }
            else if(group == 'C')
            {
                  System.out.println("11.00 a.m");
            }
            else
            {
                  System.out.println("No such group");
            }
    }
}
```

This program is a little bit different from the ones before because it includes some basic **error checking**. That is, it does not *assume* that the user of this program will always type the *expected* values. If the wrong group (not A, B or C) is entered, an error message is displayed saying "*No such group*".

```
// valid groups checked above
else // if this 'else' is reached, group entered must be invalid
{
      System.out.println("No such group"); // error message
}
```

Error checking like this is a good habit to get into.

There is one other point to notice here. The program is set up so that only the upper case letters, 'A', 'B' and 'C' are accepted as valid. If the user were to enter 'a' for example then "No such group" would be displayed. Can you think how to fix this? You will have the opportunity to do that in the programming exercises at the end of the chapter.

This use of nested selections is okay up to a point, but when the number of options becomes large the program can again look very untidy. Fortunately, this type of selection can also be implemented in Java with another form of control: a **`switch`** statement.

3.7 The ‘switch’ Statement

Our next program, `TimetableWithSwitch`, behaves in exactly the same way as the previous program but using a **switch** instead of a series of nested **if**… **else** statements allows a neater implementation. Take a look at it and then we’ll discuss it.

TimetableWithSwitch

```
import java.util.Scanner;

public class TimetableWithSwitch
{
    public static void main(String[] args)
    {
        char group;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("***Lab Times***");
        System.out.println("Enter your group (A,B,C)");
        group = keyboard.next().charAt(0);
        switch(group) // beginning of switch
        {
            case 'A': System.out.println("10.00 a.m.");
                      break;
            case 'B': System.out.println("1.00 p.m.");
                      break;
            case 'C': System.out.println("11.00 a.m.");
                      break;
            default: System.out.println("No such group");
        } // end of switch
    }
}
```

As you can see, this looks a lot neater. The **switch** statement works in exactly the same way as a set of nested **if** statements, but is more compact and readable. A **switch** statement may be used when

- only one variable is being checked in each condition (in this case every condition involves checking the variable group);
- the check involves specific values of that variable (e.g. ‘A’, ‘B’) and not ranges (for example $\geq$ 40).

As can be seen from the example above, the keyword **case** is used to precede a possible value of the variable that is being checked. There may be many **case** statements in a single **switch** statement. The general form of a **switch** statement in Java is given as follows:

```
switch(someVariable)
{
    case value1: // instructions(s) to be executed
                 break;
    case value2: // instructions(s) to be executed
                 break;
    // more values to be tested can be added
    default: // instruction(s) for default case
}
```

where

- `someVariable` is the name of the variable being tested. This variable is most commonly of type **int** or **char**.
- `value1`, `value2`, etc. are the possible values of that variable.
- **break** is a command that forces the program to skip the rest of the **switch** statement.
- **default** is an optional (last) case that can be thought of as an "otherwise" statement. It allows you to code instructions that deal with the possibility of none of the cases above being **true**.

The **break** statement is important because it means that once a matching case is found, the program can skip the rest of the cases below. If it is not added, not only will the instructions associated with the matching case be executed, but also all the instructions associated with all the cases below it. Notice that the last set of instructions does not need a **break** statement as there are no other cases to skip.

3.7.1 Grouping Case Statements

There will be instances when a particular group of instructions is associated with more than one **case** option. As an example, consider the time table again. Let's assume that both groups 'A' and 'C' have a lab at 10.00 a.m. The following **switch** statement would process this without grouping case 'A' and 'C' together:

```
// groups A and C have labs at the same time
switch(group)
{
     case 'A':   System.out.println("10.00 a.m.");
                 break;
     case 'B':   System.out.println("1.00 p.m.");
                 break;
     case 'C':   System.out.println("10.00 a.m.");
                 break;
     default:    System.out.println("No such group");
}
```

While this will work, both **case** 'A' and **case** 'C' have the same instruction associated with them:

```
System.out.println("10.00 a.m.");
```

Rather than repeating this instruction, the two **case** statements can be combined into one as follows:

```
// groups A and C have been processed together
switch(group)
{
     case 'A': case 'C':  System.out.println("10.00 a.m.");
                          break;
     case 'B':   System.out.println("1.00 p.m.");
                 break;
   default:      System.out.println("No such group");
}
```

In the example above a time of 10.00 a.m. will be displayed when the group is either 'A' or 'C'. The example above combined two **case** statements, but there is no limit to how many such statements can be combined.

3.7.2 Removing Break Statements

In the examples above we have always used a **break** statement to avoid executing the code associated with more than one **case** statement. There may be situations where it is *not* appropriate to use a **break** statement and we *do* wish to execute the code associated with more than one case statement.

For example, let us assume that spies working for a secret agency are allocated different levels of security clearance, the lowest being 1 and the highest being 3. A spy with the highest clearance level of 3 can access all the secrets, whereas a spy with a clearance of level of 1 can see only secrets that have the lowest level of security. An administrator needs to be able to view the collection of secrets that a spy with a particular clearance level can see. We can implement this scenario by way of a **switch** statement in the below. Take a look at it and then we will discuss it.

SecretAgents

```
import java.util.Scanner;

public class SecretAgents
{
    public static void main(String[] args)
    {
        int security;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("***Secret Agents***");
        System.out.println("Enter security level (1,2,3)");
        security = keyboard.nextInt();
        switch(security) // check level of security
        {
           // level 3 security
          case 3: System.out.println("The code to access the safe is 007.");
          // level 2 security
           case 2: System.out.println("Jim Kitt is really a double agent.");
           // level 1 security
          case 1: System.out.println("Martinis in the hotel bar may be poisoned.");
                  break; // necessary to avoid error message below
           default: System.out.println("No such security level.");
        }
    }
}
```

You can see that there is just a single **break** statement at the end of **case** 1.

```
case 3: System.out.println("The code to access the safe is 007.");
case 2: System.out.println("Jim Kitt is really a double agent.");
case 1: System.out.println("Martinis in the hotel bar may be poisoned.");
break; // the only break statement
```

If the user entered a security level of 3 for example, the `println` instruction associated with this case would be executed:

```
case 3: System.out.println("The code to access the safe is 007.");
```

However, as there is no **`break`** statement at the end of this instruction, the instruction associated with the **`case`** below is then also executed:

```
System.out.println("Jim Kitt is really a double agent.");
```

We have still not reached a **`break`** statement so the instruction associated with the next **`case`** statement is then executed:

```
System.out.println("Martinis in the hotel bar may be poisoned.");
break; // the only break statement
```

Here we do reach a **`break`** statement so the **`switch`** terminates. Here is a sample test run:

```
***Secret Agents***
Enter security level (1, 2, 3)
3
The code to access the safe is 007.
Jim Kitt is really a double agent.
Martinis in the hotel bar may be poisoned.
```

Because the security level entered is 3 `all` secrets can be revealed. Here is another sample test run when security level 2 is entered:

```
***Secret Agents***
Enter security level (1, 2, 3)
2
Jim Kitt is really a double agent.
Martinis in the hotel bar may be poisoned.
```

Because the security level is 2 the first secret is not revealed.

The last **break** statement is necessary as we wish to avoid the final error message if a valid security level (1, 2 or 3) is entered. The error message is only displayed if an invalid security level is entered:

```
***Secret Agents***
Enter security level (1, 2, 3)
8
No such security level.
```

3.8 Self-test Questions

1. Explain the difference between *sequence* and *selection*.

2. When would it be appropriate to use

   - an **if** statement?
   - an **if**…**else** statement?
   - a **switch** statement?

3. Consider the following Java program, which is intended to display the cost of a cinema ticket. Part of the code has been replaced by a comment:

```
import java.util.Scanner;

public class CinemaTicket
{
     public static void main(String[] args)
     {
          double price = 10.00;
          int age;
          Scanner keyboard = new Scanner(System.in);
          System.out.print("Enter your age: ");
          age = keyboard.nextInt();
          // code to reduce ticket price for children goes here
          System.out.println("Ticket price = " + price);
       }
}
```

   Replace the comment so that children under the age of 14 get half price tickets.

4. Consider the following program:

```
import java.util.Scanner;

public class Colours
{
   public static void main(String[] args)
   {
      int x;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter a number: ");
      x = keyboard.nextInt();
      if (x > 10)
      {
         System.out.println("Green");
         System.out.println("Blue");
      }
      System.out.println("Red");
   }
}
```

What would be the output from this program if

(a) the user entered 10 when prompted?
(b) the user entered 20 when prompted?
(c) the braces used in the **if** statement are removed, and the user enters 10 when prompted?
(d) the braces used in the **if** statement are removed, and the user enters 20 when prompted?

5. Consider the following program:

```
import java.util.Scanner;

public class Colours2
{
    public static void main(String[] args)
    {
        int x;
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter a number: ");
        x = keyboard.nextInt();
        if (x > 10)
        {
            System.out.println("Green");
        }
        else
        {
            System.out.println("Blue");
        }
        System.out.println("Red");
    }
}
```

What would be the output from this program if

(a) the user entered 10 when prompted?
(b) the user entered 20 when prompted?

6. Consider the following program:

```
import java.util.Scanner;

public class Colours3

{
public static void main(String[] args)
    {
        int x;
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter a number: ");
        x = keyboard.nextInt();
        switch (x)
        {
            case 1: case 2: System.out.println("Green"); break;
            case 3: case 4: case 5: System.out.println("Blue"); break;
            default: System.out.println("numbers 1-5 only");
        }
        System.out.println("Red");
    }
}
```

What would be the output from this program if

(a) the user entered 1 when prompted?
(b) the user entered 2 when prompted?
(c) the user entered 3 when prompted?
(d) the user entered 10 when prompted?
(e) the **break** statements were removed from the **switch** statement and the user entered 3 when prompted?
(f) the **default** were removed from the **switch** statement and the user entered 10 when prompted?

3.9 Programming Exercises

1. Design and implement a program that asks the user to enter two numbers and then displays the message "NUMBERS ARE EQUAL", if the two numbers are equal and "NUMBERS ARE NOT EQUAL", if they are not equal.

   Hint: Don't forget to use the double equals (= =) to test for equality.

2. Adapt the program developed in the question above so that as well as checking if the two numbers are equal, the program will also display "FIRST NUMBER BIGGER" if the first number is bigger than the second number and display "SECOND NUMBER BIGGER" if the second number is bigger than the first.

3. Design and implement a program that asks the user to enter two numbers and then guess at the sum of those two numbers. If the user guesses correctly a congratulatory message is displayed, otherwise a commiseration message is displayed along with the correct answer.

4. Implement the `DisplayResult` program from Sect. 3.4 which processed an exam mark, and then adapt the program so that marks of 70 or above are awarded a distinction rather than a pass.

5. In programming Exercise 5 of the previous chapter you were asked to calculate the BMI of an individual. Adapt this program so that it also reports on whether the BMI is in a healthy range, or if it indicates the person is underweight or overweight, using the following table:

BMI	Classification
<18.5	Underweight
18.5—24.9	Healthy
>24.9	Overweight

6. Write a program to take an order for a new computer. The basic system costs 375.99. The user then has to choose from a 38 cm screen (costing 75.99) or a 43 cm screen (costing 99.99). The following extras are optional.

Item	Price
Antivirus software	65.99
Printer	125.00

The program should allow the user to select from these extras and then display the final cost of the order.

7. (a) Implement the `TimetableVersion2` program (Sect. 3.6) so that it accepts both upper case and lower case letters for the group.
   (b) Adapt the `TimetableWithSwitch` program (Sect. 3.7) in the same way.

8. Consider a bank that offers four different types of account ('A', 'B', 'C' and 'X'). The following table illustrates the annual rate of interest offered for each type of account.

Account	Annual rate of interest (%)
A	1.5
B	2
C	1.5
X	5

Design and implement a program that allows the user to enter an amount of money and a type of bank account, before displaying the amount of money that

can be earned in one year as interest on that money for the given type of bank account. You should use the **switch** statement when implementing this program.

Hint: be careful to consider the case of the letters representing the bank accounts. You might want to restrict this to, say, just upper case. Or you could enhance your program by allowing the user to enter either lower case or upper case letters.

9. Consider the bank accounts discussed in Exercise 8 again. Now assume that each type of bank account is associated with a minimum balance as given in the table below:

Account	Minimum balance
A	250
B	1000
C	250
X	5000

Adapt the **switch** statement of the program in Exercise 8 above so that the interest is applied only if the amount of money entered satisfies the minimum balance requirement for the given account. If the amount of money is below the minimum balance for the given account an error message should be displayed.

Check for updates

4 Iteration

Outcomes:

By the end of this chapter you should be able to:

- *explain the term* ***iteration***;
- *repeat a section of code with a* **for** *loop*;
- *repeat a section of code with a* **while** *loop*;
- *repeat a section of code with a* **do…while** *loop*;
- *select the most appropriate loop for a particular task*;
- *use a* **break** *statement to terminate a loop*;
- *use a* **continue** *statement to skip an iteration of a loop*;
- *explain the term* ***input validation*** *and write simple validation routines.*

4.1 Introduction

So far we have considered sequence and selection as forms of program control. One of the advantages of using computers rather than humans to carry out tasks is that they can repeat those tasks over and over again without ever getting tired. With a computer we do not have to worry about mistakes creeping in because of fatigue, whereas humans would need a break to stop them becoming sloppy or careless when carrying out repetitive tasks over a long period of time. Neither sequence nor selection allows us to carry out this kind of control in our programs. As an example, consider a program that needs to display a square of stars (five by five) on the screen as follows:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_4

This could be achieved with five output statements executed in sequence, as shown in below in a program which we have called `DisplayStars`:

DisplayStars

```
public class DisplayStars
{
      public static void main (String[] args)
      {
           System.out.println("*****"); // instruction to display one row
           System.out.println("*****"); // instruction to display one row
           System.out.println("*****"); // instruction to display one row
           System.out.println("*****"); // instruction to display one row
           System.out.println("*****"); // instruction to display one row
       }
}
```

While this produces the desired result, the program actually consists just of the following instruction to print out one row, but repeated 5 times:

```
System.out.println("*****"); // this instruction is written 5 times
```

Writing out the same line many times is somewhat wasteful of our precious time as programmers. Imagine what would happen if we wanted a square 40 by 40!

Rather than write out this instruction five times we would prefer to write it out once and get the program to *repeat that same line* another four times. Something like:

```
public class DisplayStars
{
   public static void main (String[] args)
   {
      // CARRY OUT THE FOLLOWING INSTRUCTION 5 TIMES
      System.out.println("*****");
   }
}
```

Iteration is the form of program control that allows us to instruct the computer to carry out a task several times by repeating a section of code. For this reason this form of control is often also referred to as **repetition**. The programming structure that is used to control this repetition is often called a **loop**; we say that the loop **iterates** a certain number of times. There are three types of loop in Java:

- **`for`** loop;
- **`while`** loop;
- **`do…while`** loop.

We will consider each of these in turn.

4.2 The 'for' Loop

If we wish to repeat a section of code a *fixed* number of times (five in the example above) we would use Java's **`for`** loop. For example, the program below re-writes `DisplayStars` by making use of a **`for`** loop. Take a look at it and then we will discuss it:

DisplayStars2

```
public class DisplayStars2
{
    public static void main (String[] args)
    {
        for(int i = 1; i <= 5; i++) // loop to repeat 5 times
        {
            System.out.println("*****"); // instruction to display one row
        }
    }
}
```

As you can see there are three bits of information in the header of the **`for`** loop, each bit separated by a semi-colon:

```
for(int i = 1; i <= 5; i++) // three bits of information in the brackets
{
    System.out.println("*****");
}
```

All three bits of information relate to a **counter**. A counter is just another variable (usually integer) that has to be created. We use it to keep track of how many times we have been through the loop so far. In this case we have called our counter `i`, but we could give it any variable name—often though, simple names like `i` and `j` are chosen.

Let's look carefully at how this **`for`** loop works. First the counter is initialized to some value. We have decided to initialize it to 1:

```
for(int i = 1; i <= 5; i++) // counter initialized to 1
{
   System.out.println("*****");
}
```

Notice that the loop counter `i` is *declared* as well as initialized in the header of the loop. Although it is possible to declare the counter variable prior to the loop, declaring it within the header restricts the use of this variable to the loop itself. This is often preferable.

The second bit of information in the header is a test, much like a test when carrying out selection. When the test returns a **`boolean`** value of **`true`** the loop repeats; when it returns a **`boolean`** value of **`false`** the loop ends. In this case the

counter is tested to see if it is less than or equal to 5 (as we wish to repeat this loop 5 times):

```
for(int i = 1; i <= 5; i++) // counter tested
{
   System.out.println("*****");
}
```

Since the counter was set to 1, this test is **true** and the loop is entered. We sometimes refer to the instructions inside the loop as the **body** of the loop. As with **if** statements, the braces of the **for** loop can be omitted when only a single instruction is required in the body of the loop—but for clarity we will always use braces with our loops. When the body of the loop is entered, all the instructions within the braces of the loop are executed. In this case there is only one instruction to execute:

```
for(int i = 1; i <= 5; i++)
{
   System.out.println("*****"); // this line is executed
}
```

This line prints a row of stars on the screen. Once the instructions inside the braces are complete, the loop *returns to the beginning* where the third bit of information in the header of the **for** loop is executed. The third bit of information *changes* the value of the counter so that eventually the loop test will be **false**. If we want the loop to repeat 5 times and we have started the counter off at 1, we should *add* 1 to the counter each time we go around the loop:

```
for(int i = 1; i <= 5; i++) // counter is changed
{
   System.out.println("*****");
}
```

After the first increment, the counter now has the value of 2. Once the counter has been changed the test is examined again to see if the loop should repeat:

```
for(int i = 1; i <= 5; i++) // counter tested again
{
   System.out.println("*****");
}
```

This test is still **true** as the counter is still not greater than 5. Since the test is **true** the body of the loop is entered again and another row of stars printed. This process of checking the test, entering the loop and changing the counter repeats

until five rows of stars have been printed. At this point the counter is incremented as usual:

```
for(int i = 1; i <= 5; i++) // counter eventually equals 6
{
   System.out.println("*****");
}
```

Now when the test is checked it is **false** as the counter is greater than five:

```
for(int i = 1; i <= 5; i++) // now the test is false
{
   System.out.println("*****");
}
```

When the test of the **for** loop is **false** the loop stops. The instructions inside the loop are skipped and the program continues with any instructions after the loop.

Now that you have seen one example of the use of a **for** loop, the general form of a **for** loop can be given as follows:

```
for( /* start counter */ ; /* test counter */ ; /* change counter */)
{
      // instruction(s) to be repeated go here
}
```

Be very careful that the loop counter and the test achieve the desired result. For example, consider the following test:

```
for(int i = 1; i >= 10; i++) // something wrong with this test!
{
   // instruction(s) to be repeated go here
}
```

Can you see what is wrong here?

The test to continue with the loop is that the counter be *greater* than or equal to 10 (`i >= 10`). However, the counter starts at 1 so this test is immediately **false**! Because this test would be **false** immediately, the loop does not repeat at all and it is skipped altogether!

Now consider this test:

```
for(int i = 1; i >= 1; i++) // something wrong with this test again!
{
   // instruction(s) to be repeated go here
}
```

Can you see what is wrong here?

The test to continue with the loop is that the counter be greater than or equal to 1 (`i >= 1`). However, the counter starts at 1 and increases by 1 each time, so this test will always be **`true`**! Because this test would be **`true`** always, the loop will never stop repeating when it is executed!

As long as you are careful with your counter and your test, however, it is a very easy matter to set your **`for`** loop to repeat a certain number of times. If, for example, we start the counter at 1 and increment it by 1 each time, and we need to carry out some instructions 70 times, we could have the following test in the **`for`** loop:

```
for(int i = 1; i <= 70; i++) // this loop carries out the instructions 70 times
{
   // instruction(s) to be repeated goes here
}
```

4.2.1 Varying the Loop Counter

The `DisplayStars2` program illustrated a common way of using a **`for`** loop; start the counter at 1 and add 1 to the counter each time the loop repeats. However, you may start your counter at *any* value and change the counter in any way you choose when constructing your **`for`** loops.

For example, we could have re-written the **`for`** loop of the above program so that the counter starts at 0 instead of 1. In that case, if we wish the **`for`** loop to still execute the instructions 5 times the counter should reach 4 and not 5:

```
// this counter starts at 0 and goes up to 4 so the loop still executes 5 times
for(int i = 0; i <= 4; i++)
{
   System.out.println("*****");
}
```

Another way to ensure that the counter does not reach a value greater than 4 is to insist that the counter stays below 5. In this case we need to use the "less than" operator (<) instead of the "less than or equal to" operator (<=):

```
// this loop still executes 5 times
for(int i = 0; i < 5; i++)
{
   System.out.println("*****");
}
```

We can also change the way we modify the counter after each iteration. Returning to the original **for** loop, we would increment the counter by 2 each time instead of 1. If we still wish the loop to repeat 5 times we could start at 2 and get the counter to go up to 10:

```
// this loop still executes 5 times
for(int i = 2; i <= 10; i = i+2) // the counter moves up in steps of 2
{
   System.out.println("*****");
}
```

Finally, counters can move down as well as up. As an example, look at the following program that prints out a countdown of the numbers from 10 down to 1.

Countdown

```
public class Countdown
{
    public static void main(String[] args)
    {
        System.out.println("*** Numbers from 10 to 1 ***");
        System.out.println();
        for (int i=10; i >= 1; i--) // counter moving down from 10 to 1
        {
            System.out.println(i);
        }
    }
}
```

Here the counter starts at 10 and is reduced by 1 each time. The loop stops when the counter falls below the value of 1. Note the use of the loop counter *inside* the loop:

```
System.out.println(i); // value of counter 'i' used here
```

This is perfectly acceptable as the loop counter is just another variable. However, when you do this, be careful not to inadvertently *change* the loop counter within the loop body as this can throw the test of your **for** loop off track! Running the `Countdown` program gives us the following result:

```
*** Numbers from 10 to 1 ***

10
9

8

7

6
```

```
5
4
3
2
1
```

4.2.2 The Body of the Loop

The body of the loop can contain any number and type of instructions, including variable declarations, **`if`** statements, **`switch`** statements, or even another loop! For example, the `DisplayEven` program below modifies our `Countdown` by including an **`if`** statement inside the **`for`** loop so that only the *even* numbers from 10 to 1 are displayed:

DisplayEven

```
public class DisplayEven
{
    public static void main(String[] args)
    {
        System.out.println("*** Even numbers from 10 to 1 ***");
        System.out.println();
        for(int i=10; i >= 1; i--) // loop through the numbers 10 down to 1
        {
            // body of the loop contains in 'if' statement
            if (i%2 == 0) // check if number is even
            {
                System.out.println(i); // number displayed only when it is checked to be even
            }
        }
    }
}
```

You can see that the body of the **`for`** loop contains within it an **`if`** statement. The test of the **`if`** statement checks the current value of the loop counter '`i`' to see if it is an even number:

```
for(int i=10; i >= 1; i--)
{
    if (i%2 == 0) // use the modulus operator to check the value of the loop counter
    {
        System.out.println(i);
    }
}
```

An even number is a number that leaves no remainder when divided by 2, so we use the modulus operator (`%`) to check this. Now the loop counter is displayed only if it is an even number. Running the program gives us the obvious results:

```
*** Even numbers from 10 to 1 ***
10
8
6
4
2
```

In this example we included an **`if`** statement inside the **`for`** loop. It is also possible to have one **`for`** loop inside another. When we have one loop inside another we refer to these loops as **nested** loops. As an example of this consider the program `DisplayStars3` below, which displays a square of stars as before, but this time uses a pair of nested loops to achieve this:

DisplayStars3

```
public class DisplayStars3
{
   public static void main (String[] args)
   {
      for(int i = 1; i <= 5; i++) // outer loop as before
      {
         for (int j = 1; j <= 5; j++) // inner loop to display one row of stars
         {
            System.out.print("*");
         }  // inner loop ends here
         System.out.println(); // necessary to start next row on a new line
      } // outer loop ends here
   }
}
```

You can see that the outer **`for`** loop is the same as the one used previously in `DisplayStars2`. Whereas in the original program we had a single instruction to display a single row of stars inside our loop:

```
System.out.println("*****"); // original instruction inside the 'for' loop
```

in `DisplayStars3` we have replaced this instruction with *another* **`for`** loop, followed by a blank `println` instruction:

```
// new instructions inside the original 'for' loop to print a single row of stars
for (int j = 1; j <= 5; j++) // new name for this loop counter
{
   System.out.print("*");
}
System.out.println();
```

Notice that when we place one loop inside another, we need a fresh name for the loop counter in the nested loop. In this case we have called the counter ‘`j`’. These instructions together allow us to display a single row of 5 stars and move to a new line, ready to print the next row.

Let's look at how the control in this program flows. First the outer loop counter is set to 1:

```
for(int i = 1; i <= 5; i++) // outer loop counter initialized
{
    for (int j = 1; j <= 5; j++)
    {
        System.out.print("*");
    }
    System.out.println();
}
```

The test of the outer loop is then checked:

```
for(int i = 1; i <= 5; i++) // outer loop counter tested
{
    for (int j = 1; j <= 5; j++)
    {
        System.out.print("*");
    }
    System.out.println();
}
```

This test is found to be **`true`** so the body of the outer loop is executed. First the inner loop repeats five times:

```
for(int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++) // this loop repeats 5 times
    {
        System.out.print("*");
    }
    System.out.println();
}
```

The inner loop prints five stars on the screen as follows:

```
*****
```

After the inner loop stops, there is one more instruction to complete: the command to move the cursor to a new line:

```
for(int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++)
    {
        System.out.print("*");
    }
    System.out.println(); // last instruction of outer loop
}
```

This completes one cycle of the outer loop, so the program returns to the beginning of this loop and increments its counter:

```
for(int i = 1; i <= 5; i++)  // counter moves to 2
{
    for (int j = 1; j <= 5; j++)
    {
        System.out.print("*");
    }
    System.out.println();
}
```

The test of the outer loop is then checked and found to be **`true`** and the whole process repeats, printing out a square of five stars as before.

`DisplayStars3` displayed a five by five square of stars. Now take a look at the next program and see if you can work out what it does. Look particularly at the header of the inner loop:

DisplayShape

```
public class DisplayShape
{
   public static void main (String[] args)
   {
       for(int i = 1; i <= 5; i++) // outer loop controlling the number of rows
       {
         for (int j = 1; j <= i; j++) // inner loop controlling the number of stars in one row
         {
                 System.out.print("*");
         }
         System.out.println();
       }
   }
}
```

You can see this is very similar to the previous program, except that in that program the inner loop displayed 5 stars each time. In this case the number of stars is not fixed to a number, but to the value of the outer loop counter `i`:

```
for(int i = 1; i <= 5; i++) // outer loops controls the number of rows
{
    // inner loop determines how many stars in each row
    for (int j = 1; j <= i; j++) // inner loop displays 'i' number of stars
    {
        System.out.print("*");
    }
    System.out.println();
}
```

The first time around this loop the inner loop will display only 1 star in the row as the `i` counter starts at 1. The second time around this loop it will display 2 stars as the `i` counter is incremented, then 3 stars. Eventually it will display 5 stars the last time around the loop when the outer `i` counter reaches 5. Effectively this means the program will display a *triangle* of stars as follows:

```
*
* *
* * *
* * * *
* * * * *
```

4.2.3 Revisiting the Loop Counter

Before we move on to look at other kinds of loops in Java it is important to understand that, although a **`for`** loop is used to repeat something a fixed number of times, you don't necessarily need to know this fixed number when you are writing the program. This fixed number could be a value given to you by the user of your program, for example. This number could then be used to test against your loop counter. The program below modifies `DisplayStars3` by asking the user to determine the size of the square of stars.

DisplayStars4

```
import java.util.Scanner;

public class DisplayStars4
{
    public static void main(String[] args)
    {
        int num; // to hold user response
        Scanner keyboard = new Scanner(System.in);
        // prompt and get user response
        System.out.println("Size of square?");
        num = keyboard.nextInt();
        // display square
        for(int i = 1; i <= num; i++) // number of rows fixed to 'num'
        {
            for (int j = 1; j <= num; j++) // number of stars in a row fixed to 'num'
            {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

In this program you cannot tell from the code exactly how many times the loops will iterate, but you can say that they will iterate `num` number of times—whatever the user may have entered for `num`. So in this sense the loop is still fixed. Here is a sample run of `DisplayStars4`:

```
Size of square?
7
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
```

Here is another sample run:

```
Size of square?
3
* * *
* * *
* * *
```

4.3 The 'while' Loop

Much of the power of computers comes from the ability to ask them to carry out repetitive tasks, so iteration is a very important form of program control. The **`for`** loop is an often used construct to implement fixed repetitions.

Sometimes, however, a repetition is required that is *not fixed* and a **`for`** loop is not the best one to use in such a case. Consider the following scenarios, for example:

- a racing game that repeatedly moves a car around a track until the car crashes;
- a ticket issuing program that repeatedly offers tickets for sale until the user chooses to quit the program;
- a password checking program that does not let a user into an application until he or she enters the right password.

Each of the above cases involves repetition; however, the number of repetitions is not fixed but depends upon some condition. The **`while`** loop offers one type of non-fixed iteration. The syntax for constructing this loop in Java is as follows:

```
while ( /* test goes here */ )
{
   // instruction(s) to be repeated go here
}
```

As you can see, this loop is much simpler to construct than a **`for`** loop; as this loop is not repeating a fixed number of times, there is no need to create a counter to keep track of the number of repetitions.

When might this kind of loop be useful? The first example we will explore is the use of the **`while`** loop to check data that is input by the user. Checking input data for errors is referred to as **input validation**.

For example, look back at the program `DisplayResult` in the last chapter, which asked the user to enter an exam mark:

```
System.out.println("What exam mark did you get?");
mark = keyboard.nextInt();
if (mark >= 40)
// rest of code goes here
```

The mark that is entered should never be greater than 100 or less than 0. At the time we assumed that the user would enter the mark correctly. However, good programmers never make this assumption!

Before accepting the mark that is entered and moving on to the next stage of the program, it is good practice to check that the mark entered is indeed a valid one. If it is not, then the user will be allowed to enter the mark again. This will go on until the user enters a valid mark.

We can express this using pseudocode as follows:

```
PROMPT for mark
ENTER mark
KEEP REPEATING WHILE mark < 0 OR mark > 100
BEGIN
  DISPLAY error message to user
  ENTER mark
END
// REST OF PROGRAM HERE
```

The design makes clear that an error message is to be displayed every time the user enters an invalid mark. The user may enter an invalid mark many times so an iteration is required here.

However, the number of iterations is not fixed as it is impossible to say how many, if any, mistakes the user will make.

This sounds like a job for the **`while`** loop.

```
System.out.println("What exam mark did you get?");
mark = keyboard.nextInt();
while (mark < 0 || mark > 100) // check for invalid input
{
    // display error message and allow for re-input
    System.out.println("Invalid mark: Re-enter!");
    mark = keyboard.nextInt();
}
if (mark >= 40)
// rest of code goes here
```

The program below shows the whole of the `DisplayResult` rewritten to include the input validation. Notice how this works–we ask the user for the mark; if it is within the acceptable range, the **`while`** loop is not entered and we move past it to the other instructions. But if the mark entered is less than zero or greater than 100 we enter the loop, display an error message and ask the user to input the mark again. This continues until the mark is within the required range.

DisplayResult2

```
import java.util.Scanner;

public class DisplayResult2
{
    public static void main(String[] args)
    {
        int mark;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What exam mark did you get?");
        mark = keyboard.nextInt();
        // input validation
        while (mark < 0 || mark > 100) // check if mark is invalid
        {
            // display error message
            System.out.println("Invalid mark: please re-enter");
            // mark must be re-entered
            mark = keyboard.nextInt();
        }
        // by this point loop is finished and mark will be valid
        if (mark >= 40)
        {
            System.out.println("Congratulations, you passed");
        }
        else
        {
            System.out.println("I'm sorry, but you failed");
        }
        System.out.println("Good luck with your other exams");
    }
}
```

Here is a sample program run:

```
What exam mark did you get?
101
Invalid mark: please re-enter
−10
Invalid mark: please re-enter
10
I'm sorry, but you failed
Good luck with your other exams
```

4.4 The 'do…while' Loop

There is one more loop construct in Java that we need to tell you about: the **do… while** loop.

The **do…while** loop is another variable loop construct, but, unlike the **while** loop, the **do…while** loop has its test at the *end* of the loop rather than at the *beginning*.

The syntax of a **do…while** loop is given below:

```
do
{
   // instruction(s) to be repeated go here
} while ( /* test goes here */ ); // note the semi-colon at the end
```

You are probably wondering what difference it makes if the test is at the end or the beginning of the loop. Well, there is one subtle difference. If the test is at the end of the loop, the loop will iterate *at least once*. If the test is at the beginning of the loop, however, there is a possibility that the condition will be **`false`** to begin with, and the loop is never executed. A **`while`** loop therefore executes *zero or more times* whereas a **`do...while`** loop executes *one or more times.*

To make this a little clearer, look back at the **`while`** loop we just showed you for validating exam marks. If the user entered a valid mark initially (such as 66), the test to trap an invalid mark (`mark < 0 || mark > 100`) would be **`false`** and the loop would be skipped altogether. A **`do...while`** loop would not be appropriate here as the possibility of never getting into the loop should be left open.

When would a **`do`**…**`while`** loop be suitable? Well, any time you wish to code a non-fixed loop that must execute at least once. Usually, this would be the case when the test can take place only *after* the loop has been entered.

To illustrate this, think about all the programs you have written so far. Once the program has done its job it terminates—if you want it to perform the same task again you have to go through the whole procedure of running the program again.

In many cases a better solution would be to put your whole program in a loop that keeps repeating until the user chooses to quit your program. This would involve asking the user each time if he or she would like to continue repeating the program, or to stop.

A **`for`** loop would not be the best loop to choose here as this is more useful when the number of repetitions can be predicted. A **`while`** loop would be difficult to use, as the test that checks the user's response to the question cannot be carried out at the beginning of the loop. The answer is to move the test to the end of the loop and use a **`do...while`** loop as follows:

```
char response; // variable to hold user response
do // place code in loop
{
     // program instructions go here
     System.out.println("another go (y/n)?");
     response = keyboard.next().charAt(0); // get user reply
} while (response == 'y' || response == 'Y'); // test must be at the end of the loop
```

Notice the test of the **`do...while`** loop allows the user to enter either a lower case or an upper case 'Y' to continue running the program:

```
while (response == 'y' || response == 'Y');
```

As an example of this application of the **do…while** loop, the program below amends the `FindCost3` program of Chap. 2, which calculated the cost of a product, by allowing the user to repeat the program as often as he or she chooses.

FindCost4

```
import java.util.Scanner;

public class FindCost4
{
    public static void main(String[] args)
    {
        double price, tax;
        char reply;
        Scanner keyboard = new Scanner(System.in);
        do
        {
            // these instructions as before
            System.out.println("*** Product Price Check ***");
            System.out.print("Enter initial price: ");
            price = keyboard.nextDouble();
            System.out.print("Enter tax rate: ");
            tax = keyboard.nextDouble();
            price = price * (1 + tax/100);
            System.out.println("Cost after tax = " + price);

            // now see if user wants another go
            System.out.println();
            System.out.print("Would you like to enter another product(y/n)?: ");
            reply = keyboard.next().charAt(0);
            System.out.println();
        } while (reply == 'y' || reply == 'Y');
    }
}
```

Here is sample program run:

```
*** Product Price Check ***
Enter initial price: 50
Enter tax rate: 10
Cost after tax = 55.0
Would you like to enter another product (y/n)?: y

*** Product Price Check ***
Enter initial price: 70
Enter tax rate: 5
Cost after tax = 73.5

Would you like to enter another product (y/n)?: Y

*** Product Price Check ***
Enter initial price: 200
Enter tax rate: 15
```

Cost after tax = 230.0

Would you like to enter another product (y/n)?: ***n***

Another way to allow a program to be run repeatedly using a **do…while** loop is to include a *menu* of options within the loop (this was very common in the days before windows and mice). The options themselves are processed by a **switch** statement. One of the options in the menu list would be the option to quit and this option is checked in the **while** condition of the loop. The program below is a reworking of the time table program of the previous chapter using this technique.

TimetableWithLoop

```
import java.util.Scanner;

public class TimetableWithLoop
{
    public static void main(String[] args)
    {
        char group, response;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("***Lab Times***");
        do // put code in loop
        {
            // offer menu of options
            System.out.println(); // create a blank line
            System.out.println("[1] TIME FOR GROUP A");
            System.out.println("[2] TIME FOR GROUP B");
            System.out.println("[3] TIME FOR GROUP C");
            System.out.println("[4] QUIT PROGRAM");
            System.out.print("enter choice [1,2,3,4]: ");
            response = keyboard.next().charAt(0); // get response
            System.out.println(); // create a blank line
            switch(response) // process response
            {
                case '1':  System.out.println("10.00 a.m ");
                           break;
                case '2':  System.out.println("1.00 p.m ");
                           break;
                case '3':  System.out.println("11.00 a.m ");
                           break;
                case '4':  System.out.println("Goodbye ");
                           break;
                default:   System.out.println("Options 1-4 only!");
            }
        } while (response != '4'); // test for Quit option
    }
}
```

Notice that the menu option is treated as a character here, rather than an integer. So option 1 would be interpreted as the character ‘1’ rather than the number 1, for example. The advantage of treating the menu option as a character rather than a number is that an incorrect menu entry would not result in a program crash if the value entered was non-numeric. Here is a sample run of this program:

****Lab Times****

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM

```
enter choice [1,2,3,4]: 2

1.00 p.m

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM
enter choice [1,2,3,4]: 5

Options 1-4 only!

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM
enter choice [1,2,3,4]: 1

10.00 a.m

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM
enter choice [1,2,3,4]: 3

11.00 a.m

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM enter choice [1,2,3,4]: 4

Goodbye
```

4.5 Picking the Right Loop

With three types of loop to choose from in Java, it can sometimes be difficult to decide upon the best one to use in each case, especially as it is technically possible to pick *any type of loop* to implement *any type of repetition*! For example, **`while`** and **`do…while`** loops *can* be used for fixed repetitions by introducing your own counter and checking this counter in the test of the loop. However, it is always best

to pick the most appropriate loop construct to use in each case, as this will simplify the logic of your code. Here are some general guidelines that should help you:

- if the number of repetitions required can be determined prior to entering the loop—use a **for** loop;
- if the number of repetitions required cannot be determined prior to entering the loop, and you wish to allow for the possibility of zero iterations—use a **while** loop;
- if the number of repetitions required cannot be determined before the loop, and you require at least one iteration of the loop—use a **do…while** loop.

4.6 The 'break' Statement

In the previous chapter we met the **break** statement when looking at **switch** statements. Here for example is a **switch** statement from the previous chapter that processed a student's timetable:

```
switch(group)
{
   case 'A': System.out.print("10.00 a.m ");
             break; // terminates switch
   case 'B': System.out.print("1.00 p.m ");
             break; // terminates switch
   case 'C': System.out.print("11.00 a.m ");
             break; // terminates switch
   default: System.out.print("No such group");
}
```

Here the **break** statement allowed the **switch** to terminate without processing the remaining **cases**. The **break** statement can also be used with Java's loops to terminate a loop before it reaches its natural end. For example, consider a program that allows the user a maximum of three attempts to guess a secret number. This is an example of a non-fixed iteration—but the iteration does have a fixed upper limit of three.

We could use any of the loop types to implement this. If we wished to use a **for** loop, however, we would need to make use of the **break** statement. Take a look at the following program that does this for a secret number of 27:

SecretNumber

```
import java.util.Scanner;

// This program demonstrates the use of the 'break' statement inside a 'for' loop

public class SecretNumber
{

   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner (System.in);
      final int SECRET = 27; // secret number
      int num; // to hold user's guess
      boolean guessed = false; // so far number not guessed

      System.out.println("You have 3 goes to guess the secret number");
      System.out.println("HINT: It is a number less than 50!");
      System.out.println();

      // look carefully at this loop
      for (int i= 1; i <= 3; i++) // loop repeats 3 times
      {
         System.out.print("Enter guess: ");
         num = keyboard.nextInt();
         // check guess
         if (num == SECRET) // check if number guessed correctly
         {
            guessed = true; // record number has been guessed correctly
            break; // exit loop
         }
      }

      // now check to see if the number was guessed correctly or not
      if (guessed)
      {
         System.out.println("Number guessed correctly");
      }
      else
      {
         System.out.println("Number NOT guessed");
      }
   }
}
```

The important part of this program is the **for** loop. You can see that it has been written to repeat three times:

```
for (int i= 1; i <= 3; i++) // loop executes 3 times
{
   System.out.print("Enter guess: ");
   num = keyboard.nextInt();
   // code here to check the guess
}
```

Each time around the loop the user gets to have a guess at the secret number. We need to do two things if we determine that the guess is correct. Firstly, set a **boolean** variable to **true** to indicate a correct guess. Then, secondly, we need to terminate the loop, even if this is before we reach the third iteration. We do so by using a **break** statement if the guess is correct:

```
for (int i= 1; i <= 3; i++)
{
   System.out.print("Enter guess: ");
   num = keyboard.nextInt();
   if (num == SECRET) // check if number guessed correctly
   {
      guessed = true; // record number has been guessed correctly
      break; // exit loop even if it has not yet finished three iterations
   }
}
```

Here is a sample program run:

You have 3 goes to guess the secret number
HINT: It is a number less than 50!
*Enter guess: **49***
*Enter guess: **27***
Number guessed correctly

Here the user guessed the number after two attempts and the loop terminated early due to the **break** statement. Here is another program run where the user fails to guess the secret number:

You have 3 goes to guess the secret number
HINT: It is a number less than 50!

*Enter guess: **33***
*Enter guess: **22***
*Enter guess: **11***
Number NOT guessed

Here the **break** statement is never reached so the loop iterates three times without terminating early.

4.7 The 'continue' Statement

Whereas the **break** statement forces a loop to terminate, a **continue** statement forces a loop to skip the remaining instructions in the body of the loop and to *continue* to the next iteration. As an example of this here is a reminder of the earlier program that displayed the even numbers from 10 down to 1:

DisplayEven – a reminder

```
public class DisplayEven
{
   public static void main(String[] args)
   {
      System.out.println("*** Even numbers from 10 to 1 ***");
      System.out.println();
      for(int i=10; i >= 1; i--) // loop through the numbers 10 down to 1
      {
         // body of the loop contains in 'if' statement
         if (i%2 == 0) // check if number is even
         {
            System.out.println(i); // number displayed only when it is checked to be even
         }
      }
   }
}
```

Here the body of the loop displayed the loop counter if it was an even number. An alternative approach would have been to skip a number if it was odd and move on to the next iteration of the loop. If the number is not skipped then it must be even, so can be displayed. This is what we have done in the following program:

DisplayEven2

```
public class DisplayEven2
{
   public static void main(String[] args)
   {
      System.out.println("*** Even numbers from 10 to 1 ***");
      System.out.println();
      for(int i=10; i>=1; i--)
      {
         if (i%2 != 0) // check if number is NOT even
         {
            continue; // skips the rest of this iteration and moves to the next iteration
         }
         System.out.println(i); // even number only displayed if we have not skipped this iteration
      }
   }
}
```

The **`if`** statement checks to see if the number is odd (not even). If this is the case the rest of the instructions in the loop can be skipped with a **`continue`** statement, so the loop moves to the next iteration:

```
if (i%2 != 0) // check if number is NOT even
{
   continue; // skips the rest of the loop body and moves to the next iteration
}
System.out.println(i); // this line only executed if this iteration is not skipped
```

The last `println` instruction is only executed if the number is even and the iteration has not been skipped. Of course, the result of running this program will be the same as the result of running the original program.

4.8 Self-test Questions

1. Consider the following program:

```
public class IterationQ1
{
   public static void main(String[] args)
   {
      for(int i = 1; i <= 4; i++)
      {
         System.out.println("YES");
      }
      System.out.println("OK");
   }
}
```

(a) How many times does this **for** loop repeat?
(b) What would be the output of this program?

2. Consider the following program:

```
public class IterationQ2
{
   public static void main(String[] args)
   {
      for(int i = 1; i < 4; i++)
      {
         System.out.println("YES");
         System.out.println("NO");
      }
      System.out.println("OK");
   }
}
```

(a) How many times does this **for** loop repeat?
(b) What would be the output of this program?

3. Consider the following program:

```
import java.util.Scanner;

public class IterationQ3
{
   public static void main(String[] args)
   {
      int num;
      Scanner keyboard = new Scanner(System.in);

      System.out.print("Enter a number: ");
      num = keyboard.nextInt();

      for(int i= 1; i< num; i++)
      {
         System.out.println("YES");
         System.out.println("NO");
      }
      System.out.println("OK");
   }
}
```

(a) What would be the output of this program if the user entered 5 when prompted?

(b) What would be the output of this program if the user entered 0 when prompted?

4. Consider the following program

```
public class IterationQ4
{
    public static void main(String[] args)
    {
        for(int i=1; i<=15; i= i +2)
        {
            System.out.println(i);
        }
    }
}
```

(a) How many times does this **for** loop repeat?

(b) What would be the output of this program?

(c) What would be the consequence of changing the test of the loop to (`i >= 15`)?

5. Consider the following program:

```
public class IterationQ5
{
    public static void main(String[] args)
    {
        for(int i=5; i>=2; i--)
        {
            switch (i)
            {
                case 1: case 3: System.out.println("YES"); break;
                case 2: case 4: case 5: System.out.println("NO");
            }
        }
        System.out.println("OK");
    }
}
```

(a) How many times does this **for** loop repeat?

(b) What would be the output of this program?

(c) What would be the consequence of changing the loop counter to (`i++`) instead of (`i--`)

6. What would be the output from the following program?

```
public class IterationQ6
{
    public static void main(String[] args)
    {
        for(int i=1; i <= 3; i++)
        {
                for(int j=1; j <= 7; j++)
                {
                        System.out.print("*");
                }
                System.out.println();
        }
    }
}
```

7. Examine the program below that aims to allow a user to guess the square of a number that is entered. Part of the code has been replaced by a comment:

```
import java.util.Scanner;

public class IterationQ7
{
   public static void main(String[] args)
   {
      int num, square;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter a number ");
      num = keyboard.nextInt();
      System.out.print("Enter the square of this number ");
      square = keyboard.nextInt();
      // loop to check answer
      while (/* test to be completed */)
      {
         System.out.println("Wrong answer, try again");
         square = keyboard.nextInt();
      }
      System.out.println("Well done, right answer");
   }
}
```

(a) Why is a **while** loop preferable to a **for** loop or a **do…while** loop here?

(b) Replace the comment with an appropriate test for this loop.

8. What would be the output of the following program?

```
public class IterationQ8
{
   public static void main(String[] args)
   {
      for(int i=1; i<=10; i++)
      {
         if (i > 5)
         {
            break;
         }
         System.out.println(i);
      }
   }
}
```

9. What would be the output of the following program?

```
public class IterationQ9
{
   public static void main(String[] args)
   {
      for(int i=1; i<=10; i++)
      {
         if (i <= 5)
         {
               continue;
         }
         System.out.println(i);
      }
   }
}
```

4.9 Programming Exercises

1. Implement a few of the programs from this chapter, and then implement the programs from the self-test questions above in order to verify your answers to those questions.

2. (a) Modify the `DisplayEven` program from Sect. 4.2.2 so that the program displays the even numbers from 1 to 20 instead of from 10 down to 1.
   (b) Modify the program further so that the user enters a number and the program displays all the even numbers from 1 up to the number entered by the user.
   (c) Modify the program again so that it identifies which of these numbers are odd and which are even. For example, if the user entered 5 the program should display something like the following:

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

3. Write a program that makes use of nested **`for`** loops to display the following shapes:

(a)
```
* * * * * *
* * * * * *
* * * * * *
```

(b)
```
* * *
* * *
* * * * * * * * *
* * * * * * * * *
* * *
* * *
```

Hint: make use of an ***`if...else`*** *statement inside your for loops.*

(c)
```
* * * *
* * *
* *
*
```

6. (a) Using a **for** loop, write a program that displays a "6 times" multiplication table; the output should look like this:

```
1 × 6 = 6
2 × 6 = 12
3 × 6 = 18
4 × 6 = 24
5 × 6 = 30
6 × 6 = 36
7 × 6 = 42
8 × 6 = 48
9 × 6 = 54
10 × 6 = 60
11 × 6 = 66
12 × 6 = 72
```

   (b) Adapt the program so that instead of a "6 times" table, the user chooses which table is displayed
   (c) Modify the program further by making use of a **while** loop to carry out some *input validation* that ensures that the user enters a number that is never less than 2. If a number less than 2 is entered an error message should be displayed and the user is asked to enter another number.
   (d) Finally, make use of a **do…while** loop so that the user is asked to enter 'y' or 'n' to indicate if they wish to run the program again. Ideally the program should run again if the user enters 'y' or 'Y'.

7. Implement the program `DisplayStars4` from Sect. 4.2.3 (which allows the user to determine the size of a square of stars) and then

   (a) adapt it so that the user is allowed to enter a size only between 2 and 20;
   (b) adapt the program further so that the user can choose whether or not to have another go.

8. Modify programming Exercise 7, from Sect. 2.12, that caries out some calculations related to a circle as follows:

   (a) Add input validation to ensure that the radius entered is always non-negative
   (b) Provide a menu interface for this program. For example:

```
[1] Set radius
[2] Display radius
[3] Display area
[4] Display perimeter
[5] Quit
```

9. Consider a vending machine that offers the following options:

```
[1] Get gum
[2] Get chocolate
[3] Get popcorn
[4] Get juice
[5] Display total sold so far
[6] Quit
```

Design and implement a program that continuously allows users to select from these options. When options 1–4 are selected an appropriate message is to be displayed acknowledging their choice. For example, when option 3 is selected the following message could be displayed:

```
Here is your popcorn
```

When option 5 is selected, the number of each type of item sold is displayed. For example:

```
3 items of gum sold
2 items of chocolate sold
6 items of popcorn sold
9 items of juice sold
```

When option 6 is chosen the program terminates. If an option other than 1–6 is entered an appropriate error message should be displayed, such as:

```
Error, options 1-6 only!
```

Methods

5

Outcomes:

By the end of this chapter you should be able to:

- *explain the meaning of the term **method***;
- *declare and define methods*;
- ***call** a method*;
- *explain the meaning of the terms **actual parameters** and **formal parameters***;
- *devise simple **algorithms** with the help of pseudocode*;
- *identify the **scope** of a particular variable*;
- *explain the meaning of the term **polymorphism***;
- *declare and use **overloaded** methods.*

5.1 Introduction

As early as Chap. 1 we were using the term **method**. There you found out that a method is a part of a class, and contains a particular set of instructions. So far, all the classes you have written have contained just one method, the `main` method. In this chapter you will see how a class can contain not just a `main` method, but many other methods as well.

Normally a method will perform a single well-defined task. Examples of the many sorts of task that a method could perform are calculating the area of a circle, displaying a particular message on the screen, converting a temperature from Fahrenheit to Celsius, and many more. In this chapter you will see how we can collect the instructions for performing these sorts of tasks together in a method.

You will also see how, once we have written a method, we can get it to perform its task within a program. When we do this we say that we are **calling** the method. When we call a method, what we are actually doing is telling the program to jump

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_5

to a new place (where the method instructions are stored), carry out the set of instructions that it finds there, and, when it has finished (that is, when the method has terminated), return and carry on where it left off.

So in this chapter you will learn how to write a method within a program, how to call a method from another part of the program and how to send information into a method and get information back.

5.2 Declaring and Defining Methods

Let's illustrate the idea of a method by thinking about a simple little program. The program prompts the user to enter his or her year of birth, month of birth and day of birth; each time the prompt is displayed, it is followed by a message, consisting of a couple of lines, explaining that the information entered is confidential. This is shown in the program below. The program would obviously then go on to do other things with the information that has been entered, but we are not interested in that, so we have just replaced all the rest of the program with a comment.

DataEntry

```
import java.util.Scanner;

public class DataEntry
{
       public static void main(String[] args)
       {
              Scanner keyboard = new Scanner(System.in);

              int year, month, day;

              // prompt for year of birth
              System.out.println("Please enter the year of your birth");

              // display confidentiality message
              System.out.println("Please note that all information supplied is confidential");
              System.out.println("No personal details will be shared with any third party");

              // get year from user
              year = keyboard.nextInt();

              // prompt for month of birth
              System.out.println("Please enter the month of your birth as a number from 1 to 12");

              // display confidentiality message
              System.out.println("Please note that all information supplied is confidential");
              System.out.println("No personal details will be shared with any third party");

              // get month from user
              month = keyboard.nextInt();

              // prompt for day of birth
              System.out.println("Please enter the day of your birth as a number from 1 to 31");

              // display confidentiality message
              System.out.println("Please note that all information supplied is confidential");
              System.out.println("No personal details will be shared with any third party");

              // get day from user
              day = keyboard.nextInt();

              // more code here
       }
}
```

You can see from the above program that we have had to type out the two lines that display the confidentiality message three times. It would be far less time-consuming if we could do this just once, then send the program off to wherever these instructions are stored, and then come back and carry on with what it was doing. You will probably have realized by now that we can indeed do this—by writing a *method*. The job of this particular method will be simply to display the confidentiality message on the screen. We need to give our method a name so that we can refer to it when required, so let's call it `displayMessage`. Here is how it is going to look:

```
static void displayMessage()
{
   System.out.println("Please note that all information supplied is confidential");
   System.out.println("No personal details will be shared with any third party");
}
```

The body of this method, which is contained between the two curly brackets, contains the instructions that we want the method to perform, namely to display two lines of text on the screen. The first line, which declares the method, is called the method **header**, and consists of three words—let's look into each of these a bit more closely:

`static`

You have seen this word in front of the `main` method many times now. However, we won't be explaining its meaning to you properly until Chap. 8. For now, all you need to know is that methods that have been declared as **`static`** (such as `main`) can only call other methods in the class if they too are **`static`**. So, if we did not declare `displayMessage` as **`static`** and tried to call it from the `main` method then our program would not compile.

`void`

In the next section you will see that it is possible for a method to *send back* or *return* some information once it terminates. This particular method simply displays a message on the screen, so we don't require it to send back any information when it terminates. The word **`void`** indicates that the method does not return any information.

`displayMessage()`

This is the name that we have chosen to give our method. You can see that the name is followed by a pair of empty brackets. Very soon you will learn that it is possible to send some information *into* a method—for example, some values that the method needs in order to perform a calculation. When we need to do that we list, in these brackets, the types of data that we are going to send in; here, however, as the method is doing nothing more that displaying a message on the screen we do not have to send in any data, and the brackets are left empty.

5.3 Calling a Method

Now that we have declared and defined our method, we can make use of it. The idea is that we get the method to perform its instructions as and when we need it to do so—you have seen that this process is referred to as *calling* the method. To call a method in Java, we simply use its name, along with the following brackets, which in this case are empty. So in this case our method call, which will be placed at the point in the program where we need it, looks like this:

```
displayMessage();
```

Now we can rewrite our `DataEntry` program, replacing the appropriate lines of code with the simple method call. The whole program is shown below:

DataEntry2

```
import java.util.Scanner;

public class DataEntry2
{
       public static void main(String[] args)
       {
              Scanner keyboard = new Scanner(System.in);

              int year, month, day;

              System.out.println("Please enter the year of your birth");
              displayMessage(); // call displayMessage method
              year = keyboard.nextInt();

              System.out.println("Please enter the month of your birth as a number from 1 to 12");
              displayMessage(); // call displayMessage method
              month = keyboard.nextInt();

              System.out.println("Please enter the day of your birth as a number from 1 to 31");
              displayMessage(); // call displayMessage method
              day = keyboard.nextInt();

              // more code here
       }

       // the code for displayMessage method
       static void displayMessage()
       {
              System.out.println("Please note that all information supplied is confidential");
              System.out.println("No personal details will be shared with any third party");
       }
}
```

You can see that the method itself is defined separately after the `main` method—although it could have come before it, since the order in which methods are presented doesn't matter to the compiler. When the program is run, however, it always starts with `main`. The overall result is of course that this program runs in exactly the same way as the previous one.

We should emphasize again here that when one method calls another method, the first method effectively pauses at that point, and the program then carries out the instructions in the called method; when it has finished doing this, it returns to the original method, which then resumes. In most of the programs in this chapter it will

be the `main` method that calls the other method. This doesn't have to be the case, however, and it is perfectly possible for any method to call another method—indeed, the called method could in turn call yet another method. This would result in a number of methods being "chained". When each method terminates, the control of the program would return to the method that called it.

You can see an example of a method being called by a method other than `main` in Sect. 5.6.

5.4 Method Input and Output

We have already told you that it is possible to send some data into a method, and that a method can send data back to the method that called it. Now we will look into this in more detail.

In order to do this we will use as an example a program that we wrote in Chap. 2. Here is a reminder of that program:

FindCost3 **- a reminder**

```
import java.util.Scanner;

/* a program to input the initial price of a product and then calculate and display its cost after tax
has been added */

public class FindCost3
{
	public static void main(String[] args)
	{
		Scanner keyboard = new Scanner(System.in);
		double price, tax;
		System.out.println("*** Product Price Check ***");
		System.out.print("Enter initial price: ");
		price = keyboard.nextDouble();
		System.out.print("Enter tax rate: ");
		tax = keyboard.nextDouble();
		price = price * (1 + tax/100);
		System.out.println("Cost after tax = " + price);
	}
}
```

The line that calculates the new price, with the sales tax added, is this one:

```
price = price * (1 + tax/100);
```

Let's create a method that performs this calculation—in a real application this would be very useful, because we might need to call this method at various points within the program, and, as you will see, each time we do so we could get it to carry out the calculation for different values of the price and the tax. We will need a way to send in these values to the method. But on top of that, we need to arrange for the method to tell us the result of adding the new tax—if it didn't do that, it wouldn't be much use!

The method is going to look like this:

```
static double addTax(double priceIn, double taxIn)
{
      return priceIn * (1 + taxIn/100);
}
```

First, take a careful look at the header. You are familiar with the first word, **static**, but look at the next one; this time, where we previously saw the word **void**, we now have the word **double**. As we have said, this method must send back—or **return**—a result, the new price of the item. So the type of data that the method is to return in this case is a **double**. In fact what we are doing here is declaring a method of *type* **double**. Thus, the *type* of a method refers to its *return* type. It is possible to declare methods of any type—**int**, **boolean**, **char** and so on.

After the type declaration, we have the name of the method, in this case addTax —and this time the brackets aren't empty. You can see that within these brackets we are declaring two variables, both of type **double**. The variables declared in this way are known as the **formal parameters** of the method. Formal parameters are variables that are created exclusively to hold values sent in from the calling method. They are going to hold, respectively, the values of the price and the tax that are going to be sent in from the calling method (you will see how this is done in a moment). Of course, these variables could be given any name we choose, but we have called them priceIn and taxIn respectively. We will use this convention of adding the suffix In to variable names in the formal parameter list throughout this book.

Now we can turn our attention to the body of the method, which as you can see, in this case, consists of a single line:

```
return priceIn * (1 + taxIn/100);
```

The word **return** in a method serves two very important functions. First it ends the method—as soon as the program encounters this word, the method terminates, and control of the program jumps back to the calling method. The second function is that it sends back a value. In this case it sends back the result of the calculation:

```
priceIn * (1 + taxIn/100)
```

You should note that if the method is of type **void**, then there is no need to include a **return** instruction—the method simply terminates once the last instruction is executed.

Now we can discuss how we actually call this method and use its return value. The whole program appears below:

FindCost4

```
public class FindCost4
{
    public static void main(String[] args )
    {
        Scanner keyboard = new Scanner(System.in);
        double price, tax;

        System.out.println("*** Product Price Check ***");

        System.out.print("Enter initial price: ");
        price = keyboard.nextDouble();
        System.out.print("Enter tax rate: ");
        tax = keyboard.nextDouble();

        price = addTax(price, tax); // call the addTax method
        System.out.println("Cost after tax = " + price);
    }

    static double addTax(double priceIn, double taxIn)
    {
        return priceIn * (1 + taxIn/100);
    }
}
```

The line in `main` that calls the method is this one:

```
price = addTax(price, tax);
```

First, we will consider the items in brackets after the method name. As you might have expected, there are two items in the brackets—these are the *actual* values that we are sending into our method. They are therefore referred to as the **actual parameters** of the method. Their values are copied onto the formal parameters in the called method. This process, which is referred to as **passing** parameters, is illustrated in Fig. 5.1.

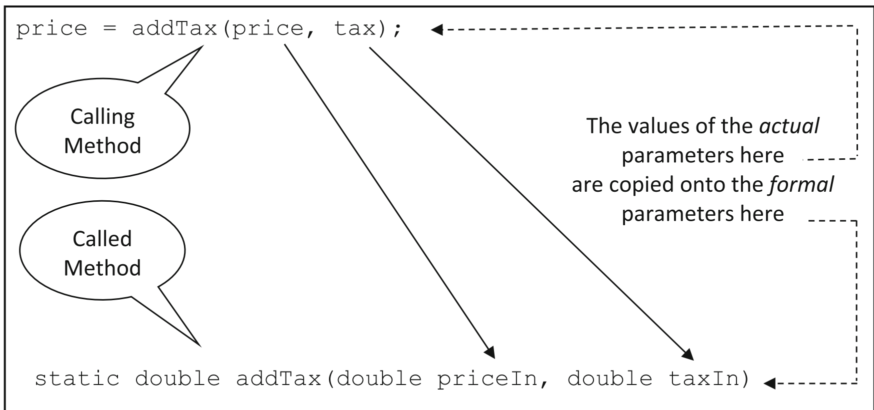


Fig. 5.1 Passing parameters from one method to another method

You might have been wondering how the program knows which values in the actual parameter list are copied onto which variables in the formal parameter list. The answer to this is that it is the *order* that is important—you can see this from Fig. 5.1—the value of `price` is copied onto `priceIn`; the value of `tax` is copied onto `taxIn`. Although the variable names have been conveniently chosen, the names themselves have nothing to do with which value is copied to which variable.

You might also be wondering what would happen if you tried to call the method with the wrong number of variables. For example:

```
price = addTax(price);
```

The answer is that you would get a compiler error, because there is no method called `addTax` that requires just one single variable to be passed into it.

You can see then that the actual parameter list must match the formal parameter list exactly. Now this is important not just in terms of the number of variables, but also in terms of the *types*. For example, using actual values this time instead of variable names, the following method call would be perfectly acceptable:

```
price = addTax(187.65, 17.5);
```

However, the one below would cause a compiler error:

```
price = addTax(187.65, 'c');
```

The reason, of course, is that `addTax` requires two **`double`**s, not a **`double`** and a **`char`**.

We can now move on to looking at how we make use of the return value of a method.

The `addTax` method returns the result that we are interested in, namely the new price of the item. What we need to do is to assign this value to the variable `price`. As you have already seen we have done this in the same line in which we called the method:

```
price = addTax(price, tax);
```

A method that returns a value can in fact be used just as if it were a variable of the same type as the return value! Here we have used it in an assignment statement—but in fact we could have simply dropped it into the `println` statement, just as we would have done with a simple variable of type **`double`**:

```
System.out.println("Cost after tax = " + addTax(price, tax));
```

5.5 More Examples of Methods

Just to make sure you have got the idea, let's define a few more methods. To start with, we will create a very simple method, one that calculates the square of a number. When we are going to write a method, there are four things to consider:

- the name that we will give to the method;
- the inputs to the method (the formal parameters);
- the output of the method (the return value);
- the body of the method (the instructions that do the job required).

In the case of the method in question, the name `square` would seem like a sensible choice. We will define the method so that it will calculate the square of any number, so it should accept a single value of type **`double`**. Similarly, it will return a **`double`**.

The instructions will be very simple—just return the result of multiplying the number by itself. So here is our method:

```
static double square(double numberIn)
{
      return numberIn * numberIn;
}
```

Remember that we can choose any names we want for the input parameters; here we have stuck with our convention of using the suffix `In` for formal parameters.

To use this method in another part of the program, such as the `main` method, is now very easy. Say, for example, we had declared and initialized two variables as follows:

```
double a = 2.5;
double b = 9.0;
```

Let's say we wanted to assign the square of a to a **double** variable x and the square of b to a **double** variable y. We could do this as follows:

```
x = square(a);
y = square(b);
```

After these instructions, x would hold the value 6.25 and y would hold the value 81.0.

For our next illustration we will choose a slightly more complicated example. We will define a method that we will call max; it will accept two integer values, and will return the bigger value of the two (of course, if they are equal, it can return the value of either one). It should be pretty clear that we will require two integer parameters, and that the method will return an integer. As far as the instructions are concerned, it should be clear that an **if...else** statement should do the job—if the first number is greater than the second, return the first number, if not return the second. Here is our method:

```
static int max(int firstIn, int secondIn)
{
       if(firstIn > secondIn)
       {
              return firstIn;
       }
       else
       {
              return secondIn;
       }
}
```

You should note that in this example we have two **return** statements, each potentially returning a different value—the value that is actually returned is decided at run-time by the values of the variables firstIn and secondIn. Remember, as soon as a **return** statement is reached the method terminates.

Working out how to write the instructions for this method was not too hard a job. In fact it was so simple, we didn't bother to design it with pseudocode. However, there will be many occasions in the future when the method has to carry out a much more complex task, and you will need to think through how to perform this task. A set of instructions for performing a job is known as an **algorithm**—common examples of algorithms in everyday life are recipes and DIY (Do-It-Yourself) instructions. Much of a programmer's time is spent devising algorithms for particular tasks, and, as you saw in Chap. 2, we can use pseudocode to help us design our algorithms. We will look at further examples as we progress through this chapter.

Let's develop one more method. There are many instances in our programming lives where we might need to test whether a number is even or odd. Let's provide a method that does this job for us. We will call our method isEven, and it will

report on whether or not a particular number is an even number. The test will be performed on integers, so we will need a single parameter of type **int**. The return value is interesting—the method will tell us whether or not a number is even, so it will need to return a value of **true** if the number is even or **false** if it is not. So our return type is going to be **boolean**. The instructions are quite simple to devise—again, an **if…else** statement should certainly do the job. But how can we test whether a number is even or not? Well, an even number will give a remainder of zero when divided by 2. An odd number will not. So we can use the modulus operator here. Here is our method:

```
static boolean isEven(int numberIn)
{
        if(numberIn % 2 == 0)
        {
                return true;
        }
        else
        {
                return false;
        }
}
```

Actually there is a slightly neater way we could have written this method. The expression:

```
numberIn % 2 == 0
```

will evaluate to either **true** or **false**—and we could therefore simply have returned the value of this expression and written our method like this:

```
static boolean isEven(int numberIn)
{
        return (numberIn % 2 == 0);
}
```

It is interesting to note that the calling method couldn't care less how the called method is coded—all it needs is for it to do the calculation correctly and return the desired value. This is something that will become very significant when we look at methods that call methods of other classes later in this semester.

A method that returns a **boolean** value can be referred to as a **boolean** method. In Chap. 3 you came across **boolean** *expressions* (expressions that evaluate to **true** or **false**) such as:

```
temperature > 10
```

or

```
y == 180
```

Because **boolean** methods also evaluate to **true** or **false** (that is, they return a value of **true** or **false**) they can—as with **boolean** expressions—be used as the test in a selection or loop.

For example, assuming that a variable called number had been declared as an **int**, we could write something like:

```
if(isEven(number))
{
       // code here
}
```

or

```
while(isEven(number))
{
       // code here
}
```

To test for a **false** value we simply negate the expression with the *not* operator (!):

```
if(!isEven(number))
{
       // code here
}
```

Before we leave this section, there is one thing we should make absolutely clear—a method cannot change the *original* value of a variable that was passed to it as a parameter. The reason for this is that all that is being passed to the method is a *copy* of whatever this variable contains. In other words, just a *value*. The method does not have access to the original variable. Whatever value is passed is copied to the parameter in the called method. We will illustrate this with a very simple program indeed:

ParameterDemo

```
public class ParameterDemo
{
    public static void main(String[] args)
    {
        int x = 10;
        demoMethod(x);
        System.out.println(x);
    }

    static void demoMethod(int xIn)
    {
        xIn = 25;
        System.out.println(xIn);
    }
}
```

You can see that in the main method we declare an integer, `x`, which is initialized to 10. We then call a method called `demoMethod`, with `x` as a parameter. The formal parameter of this method—`xIn`—will now of course hold the value 10. But the method then assigns the value of 25 to the parameter—it then displays the value on the screen.

The method ends there, and control returns to the main method. The final line of this method displays the value of `x`.

The output from this method is as follows:

25
10

This shows that the original value of `x` has not in any way been affected by what happened to `xIn` in `demoMethod`.

5.6 Variable Scope

Looking back at the `FindCost4` program in Sect. 5.4, it is possible that some of you asked yourselves the following questions: Why do we need to bother with all this stuff in the brackets? We've already declared a couple of variables called `price` and `tax`—why can't we just use them in the body of the method? Well, go ahead and try it—you will see that you get a compiler error telling you that these variables are not recognized!

How can this be? They have certainly been declared. The answer lies in the matter of *where* exactly these variables have been declared. In actual fact variables are only "visible" within the pair of curly brackets in which they have been declared—this means that if they are referred to in a part of the program outside these brackets, then you will get a compiler error. Variables that have been declared inside the brackets of a particular method are called **local** variables—so the variables price and tax are said to be *local* to the main method. We say that variables have a **scope**—this means that their visibility is limited to a particular part of the program. If `price` or `tax` were referred to in the `addTax` method, they would be out of scope.

Let's take another, rather simple, example:

ScopeTest

```
public class ScopeTest
{
    public static void main(String[] args)
    {
        int x = 1;      // x is local to main
        int y = 2;      // y is local to main
        method1(x, y); // call method1
    }

    static void method1(int xIn, int yIn)
    {
        int z; // z is local to method1
        z = xIn + yIn;
        System.out.println(z);
    }
}
```

In this program the variables `x` and `y` are local to `main`. The variable `z` is local to `method1`. The variables `xIn` and `yIn` are the formal parameters of `method1`. This program will compile and run without a problem, because none of the variables is referred to in the wrong place.

Imagine, however, that we were to rewrite this program as we have done below:

ScopeTest2

```
// this program will give rise to two compiler errors

public class ScopeTest2
{
       public static void main(String[] args)
       {
              int x = 1; // x is local to main
              int y = 2; // y is local to main
              method1(x, y); // call method1
              System.out.println(z); // this line will cause a compiler error as z is local to method1
       }

       static void method1(int xIn, int yIn)
       {
              int z;     // z is local to method1
              z = x + y; // this line will cause a compiler error as x and y are local to main
              System.out.println(z);
       }
}
```

As the comments indicate, the lines in bold will give rise to compiler errors, as the variables referred to are out of scope.

It is interesting to note that, since a method is completely unaware of what has been declared inside any other method, you could declare variables with the same name inside different methods. The compiler would regard each variable as being completely different from any other variable in another method which simply had the same name. So, for example, if we had declared a *local* variable called `x` in `method1`, this would be perfectly ok—it would exist completely independently from the variable named `x` in `main`.

To understand why this is so, it helps to know a little about what goes on when the program is running. A part of the computer's memory called the **stack** is reserved for use by running programs. When a method is called, some space on the stack is used to store the values for that method's formal parameters and its local variables. That is why, whatever names we give them, they are local to their particular method. Once the method terminates, this part of the stack is no longer accessible, and the variables effectively no longer exist. And this might help you to understand even more clearly why the value of a variable passed as a parameter to a method cannot be changed by that method.

Before we move on, it will be helpful if we list the kinds of variables that a method can access:

- a method can access variables that have been declared as formal parameters;
- a method can access variables that have been declared locally—in other words that have been declared within the curly brackets of the method;

- as you will learn in Chap. 8, a method has access to variables declared as *attributes of the class* (don't worry—you will understand what this means in good time!).

A method cannot access any other variables.

5.7 Method Overloading

You have already encountered the term *overloading* in previous chapters, in connection with operators. You found out, for example, that the division operator (/) can be used for two distinct purposes—for division of integers, and for division of real numbers. The + operator is not only used for addition, but also for concatenating two strings. So the same operator can behave differently depending on what it is operating on—operators can be overloaded.

Methods too can be overloaded. To illustrate, let's return to the `max` method of Sect. 5.5. Here it is again:

```
static int max(int firstIn, int secondIn)
{
       if(firstIn > secondIn)
       {
              return firstIn;
       }
       else
       {
              return secondIn;
       }
}
```

As you will recall, this method accepts two integers and returns the greater of the two. But what if we wanted to find the greatest of three integers? We would have to write a new method, which we have shown below. We are just showing you the header here—we will think about the actual instructions in a moment:

```
static int max(int firstIn, int secondIn, int thirdIn)
{
       // code goes here
}
```

You can see that we have given this method the same name as before—but this time it has *three* parameters instead of two. And the really clever thing is that we can declare and call both methods within the same class. Both methods have the same name but the parameter list is different—and each one will *behave* differently. In our example, the original method compares two integers and returns the greater of the two; the second one, once we have worked out the algorithm for doing this,

will examine three integers and return the value of the one that is the greatest of the three. When two or more methods, distinguished by their parameter lists, have the same name but perform different functions we say that they are **overloaded**. Method overloading is actually one example of what is known as **polymorphism**. Polymorphism literally means *having many forms*, and it is an important feature of object-oriented programming languages. It refers, in general, to the phenomenon of having methods and operators with the same name performing different functions. You will come across other examples of polymorphism in later chapters.

Now, you might be asking yourself how, when we call an overloaded method, the program knows which one we mean. The answer of course depends on the actual parameters that accompany the method call—they are matched with the formal parameter list, and the appropriate method will be called. So, if we made this call somewhere in a program:

```
int x = max(3, 5);
```

then the first version of max would be called—the version that returns the bigger of two integers. This, of course, is because the method is being called with two integer parameters, matching this header:

```
static int max(int firstIn, int secondIn)
```

However, if this call, with three integer parameters, were made:

```
int x = max(3, 5, 10);
```

then it would be the second version that was called:

```
static int max(int firstIn, int secondIn, int thirdIn)
```

One very important thing we have still to do is to devise the *algorithm* for this second version. Can you think of a way to do it? Have go at it before reading on.

One way to do it is to declare an integer variable, which we could call `result`, and start off by assigning to it the value of the first number. Then we can consider the next number. Is it greater than the current value of result? If it is, then we should assign this value to result instead of the original value. Now we can consider the third number—if this is larger than the current value of result, we assign its value to result. You should be able to see that `result` will end up having the value of the greatest of the three integers. It is helpful to express this as pseudocode:

```
SET result TO first number
IF second number > result
BEGIN
   SET result TO second number
END
IF third number > result
BEGIN
   SET result TO third number
END
RETURN result
```

Here is the code:

```java
static int max(int firstIn, int secondIn, int thirdIn)
{
       int result;
       result = firstIn;
       if(secondIn > result)
       {
              result = secondIn;
       }
       if(thirdIn > result)
       {
              result = thirdIn;
       }
       return result;
```

The following program illustrates how both versions of our max method can be used in the same program:

OverloadingDemo

```java
public class OverloadingDemo
{
       public static void main(String[] args)
       {
              int maxOfTwo, maxOfThree;
              maxOfTwo = max(2, 10);      // call the first version of max
              maxOfThree = max(-5, 5, 3); // call the second version of max
              System.out.println(maxOfTwo);
              System.out.println(maxOfThree);
       }

       // this version of max accepts two integers and returns the greater of the two
       static int max(int firstIn, int secondIn)
       {
              if(firstIn > secondIn)
              {
                     return firstIn;
              }
              else
              {
                     return secondIn;
              }
       }

       // this version of max accepts three integers and returns the greatest of the three
       static int max(int firstIn, int secondIn, int thirdIn)
       {
              int result;
              result = firstIn;
              if(secondIn > result)
              {
                     result = secondIn;
              }
              if(thirdIn > result)
              {
                     result = thirdIn;
              }
              return result;
       }
}
```

As the first call to max in the main method has two parameters, it will call the first version of max; the second call, with its three parameters, will call the second version. Not surprisingly then the output from this program looks like this:

10
5

It might have occurred to you that we could have implemented the second version of max (that is the one that takes three parameters) in a different way. We could have started off by finding the maximum of the first two integers (using the first version of max), and then doing the same thing again, comparing the result of this with the third number.

```
static int max(int firstIn, int secondIn, int thirdIn)
{
       int step1, result;
       step1 = max(firstIn, secondIn); // call the first version of max
       result = max(step1, thirdIn); // call the first version of max again
       return result;
}
```

This version is an example of how we can call a method not from the main method, but from another method.

Some of you might be thinking that if we wanted similar methods to deal with lists of four, five, six, or even more numbers, it would be an awful lot of work to write a separate method for each one—and indeed it would! But don't worry—in the next chapter you will find that there is a much easier way to deal with situations like this.

5.8 Using Methods in Menu-Driven Programs

In Chap. 4 we developed a program that presented the user with a menu of choices; we pointed out that this was a very common interface for programs before the days of graphics. Until we start working with graphics later in the semester, we will use this approach with some of our more complex programs.

Here is a reminder of that program:

***TimetableWithLoop* - a reminder**

```
import java.util.Scanner;

public class TimetableWithLoop
{
    public static void main(String[] args)
    {
        char group, response;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("***Lab Times***");
        do // put code in loop
        {
            // offer menu of options
            System.out.println(); // create a blank line
            System.out.println("[1] TIME FOR GROUP A");
            System.out.println("[2] TIME FOR GROUP B");
            System.out.println("[3] TIME FOR GROUP C");
            System.out.println("[4] QUIT PROGRAM");
            System.out.print("enter choice [1,2,3,4]: ");
            response = keyboard.next().charAt(0); // get response
            System.out.println(); // create a blank line
            switch(response) // process response
            {
                case '1':  System.out.println("10.00 a.m ");
                           break;
                case '2':  System.out.println("1.00 p.m ");
                           break;
                case '3':  System.out.println("11.00 a.m ");
                           break;
                case '4':  System.out.println("Goodbye ");
                           break;
                default: System.out.println("Options 1-4 only!");
            }
        } while (response != '4'); // test for Quit option
    }
}
```

In this program, each **case** statement consisted of a single instruction (apart from the **break**), which simply displayed one line of text. Imagine, though, that we were to develop a more complex program in which each menu choice involved a lot of processing. The **switch** statement would start to get very messy, and the program could easily become very unwieldy. In this situation, confining each menu option to a particular method will make our program far more manageable.

The next program, `CircleCalulation` is an example of such a program. The program allows a user enter the radius of a circle and then enables the area and circumference of the circle to be calculated and displayed. Four menu options are offered. The first allows the user to enter the radius. The second displays the area of the circle, and the third the circumference. The final option allows the user to quit the program.

Study it carefully, and then we will point out some of the interesting features.

CircleCalculation

```
import java.util.Scanner;

/* This program demonstrates how methods can be used in a menu-driven program */

public class CircleCalculation
{
  public static void main(String[] args)
  {
       Scanner keyboard = new Scanner(System.in);

       /* The variable below is local to the main method; if the value is needed by another method,
          it must be passed in as a parameter */
       double radius = -999; // initialize with a dummy value to show that nothing has been entered
       char choice; // to store menu choice
       do
       {

              System.out.println();
              System.out.println("*** CIRCLE CALCULATIONS ***");
              System.out.println();
              System.out.println("1. Enter the radius of the circle");
              System.out.println("2. Display the area of the circle");
              System.out.println("3. Display the circumference of the circle");
              System.out.println("4. Quit");
              System.out.println();
              System.out.println("Enter a number from 1 - 4");
              System.out.println();
              choice = keyboard.next().charAt(0);
              switch(choice)
              {
                     case '1' :   radius = option1(); // call method option1
                                  break;
                     case '2' :   option2(radius); // call method option2
                                  break;
                     case '3' :   option3(radius); // call method option3
                                  break;
                     case '4' :   break;
                     default :    System.out.println("Enter only numbers from 1 - 4");
                                  System.out.println();
              }
       } while(choice != '4');
   }

   // option1 gets the user to enter the radius of the circle
   static double option1()
   {
       double myRadius; // local variable
       Scanner keyboard = new Scanner(System.in);
       System.out.print("Enter the radius of the circle: ");
       myRadius = keyboard.nextDouble();
       return myRadius;
   }

   // option2 calculates and displays the area of the circle
   static void option2(double radiusIn)
   {
       if(radiusIn == -999)
       {
              System.out.println("Radius has not been entered");
       }
       else
       {
              double area; // local variable
              area = 3.1416 * radiusIn * radiusIn; // calculate the area
              System.out.println("The area of the circle is: " + area);
       }
   }


   // option3 calculates and displays the circumference of the circle
   static void option3(double radiusIn)
   {
       if(radiusIn == -999)
       {
              System.out.println("Radius has not been entered");
       }
       else
       {
              double circumference; // local variable
              circumference = 2 * 3.1416 * radiusIn; // calculate the circumference
              System.out.println("The circumference of the circle is: " + circumference);
       }
   }
}
```

There are no new programming techniques in this program; it is the *design* that is interesting. The comments are self-explanatory; so we draw your attention only to a few important points:

- The radius is initialized with a "dummy" value of −999. This allows us to check if the radius has been entered before attempting to perform a calculation.
- Choosing menu option 1 causes the method `option1` to be called—the value of the radius entered by the user is returned.
- Choosing menu option 2 causes the method `option2` to be called. The radius of the circle is sent in as a parameter. After using the dummy value to check that a value for the radius has been entered, the area is then calculated and displayed.
- Choosing menu option 3 causes the method `option3` to be called. This is similar to `option2`, but for the circumference instead of the area.
- Choosing option 4 causes the program to terminate—this happens because the body of the **`while`** loop executes only while `choice` is not equal to 4. If it is equal to 4, the loop is not executed and the program ends. The associated **`case`** statement consists simply of the instruction **`break`**, thus causing the program to jump out of the **`switch`** statement.
- You can see that we have had to declare a new `Scanner` object in each method where it is needed—now that you understand the notion of variable *scope*, you should understand why we have had to do this.

5.9 Self-test Questions

1. Explain the meaning of the term *method*.
2. What would be the output of the following program?

```
public class SomeApp
{
    public static void main(String[] args)
    {
       method1();
       System.out.println("England");
       method2();
       System.out.println("Ireland");
    }

    static void method1()
    {
       System.out.println("Wales");
    }

    static void method2()
    {
       System.out.println("Scotland");
       method1();
    }
}
```

3. Consider the following program:

```
public class MethodsQ3
{
        public static void main(String[] args)
        {
              System.out.println(myMethod(3, 5));
              System.out.println(myMethod(3, 5, 10));
        }

        static int myMethod(int firstIn, int secondIn, int thirdIn)
        {
              return firstIn + secondIn + thirdIn;
        }

        static int myMethod(int firstIn, int secondIn)
        {
              return firstIn - secondIn;
        }
}
```

(a) By referring to this program:

(i) Distinguish between the terms *actual parameters* and *formal parameters*.

(ii) Explain what is meant by a method's *return type*.

(iii) Explain the meaning of the terms *polymorphism* and *method overloading*.

(b) What would be displayed on the screen when this program was run?

(c) Explain, giving reasons, the effect of adding either of the following lines into the `main` method:

(i) `System.out.println(myMethod(3));`

(ii) `System.out.println(myMethod(3, 5.7, 10));`

4. What would be displayed on the screen as a result of running the following program?

```
public class MethodsQ4
{
        public static void main(String[] args)
        {
              int x = 3;
              int y = 4;
              System.out.println(myMethod(x, y));
              System.out.println(y);
        }

        static int myMethod(int firstIn, int secondIn)
        {
              int x = 10;
              int y;
              y = x + firstIn + secondIn;
              return y;
        }
}
```

5. What would be displayed on the screen as a result of running the following program?

```
public class MethodsQ5
{
  public static void main(String[] args)
  {
        int x = 2;
        int y = 7;
        System.out.println(myMethod(x, y));
        System.out.println(y);
  }

  static int myMethod(int a, int x)
  {
        int y = 20;
        return y - a - x;
  }
}
```

5.10 Programming Exercises

1. Implement the programs from the self-test questions in order to verify your answers.
2. For one of the programming exercises in Chap. 2, you wrote a program that converted pounds to kilograms. Rewrite this program, so that the conversion takes place in a separate method which is called by the `main` method.
3. In the exercises at the end of Chap. 2 you were asked to write a program that calculated the area and perimeter of a rectangle. Re-write this program so that now the instructions for calculating the area and perimeter of the rectangle are contained in two separate methods.
4. (a) Design and implement a program that converts a sum of money to a different currency. The amount of money to be converted, and the exchange rate, are entered by the user. The program should have separate methods for:

   - obtaining the sum of money from the user;
   - obtaining the exchange rate from the user;
   - making the conversion;
   - displaying the result.

   (b) Adapt the above program so that after the result is displayed the user is asked if he or she wishes to convert another sum of money. The program continues in this way until the user chooses to quit.
5. (a) Write a menu-driven program that provides three options:

   - the first option allows the user to enter a temperature in Celsius and displays the corresponding Fahrenheit temperature;

- the second option allows the user to enter a temperature in Fahrenheit and displays the corresponding Celsius temperature;
- the third option allows the user to quit.

The formulae that you need are as follows, where C represents a Celsius temperature and F a Fahrenheit temperature:

$$F = \frac{9C}{5} + 32$$

$$C = \frac{5(F - 32)}{9}$$

(b) Adapt the above program so that the user is not allowed to enter a temperature below absolute zero; this is −273.15C, or −459.67F.

Check for updates

6 Arrays

Outcomes:

By the end of this chapter you should be able to:

- *create* ***arrays***;
- *use* ***for*** *loops to process arrays*;
- *use an enhanced* ***for*** *loop to process an array*;
- *use arrays as method inputs and outputs*;
- *use arrays to send a* ***variable number of arguments*** *to a method*;
- *develop routines for accessing and manipulating arrays*;
- *distinguish between* ***one-dimensional*** *arrays and* ***multi-dimensional*** *arrays*;
- *create and process* ***two-dimensional*** *arrays*;
- *create* ***ragged arrays***.

6.1 Introduction

In previous chapters we have shown you how to create variables and store data in them. In each case the variables created could be used to hold a *single* item of data. How, though, would you deal with a situation in which you had to create and handle a very large number of data items?

An obvious approach would be just to declare as many variables as you need. Declaring a large number of variables is a nuisance but simple enough. For example, let's consider a very simple application that records seven temperature readings (one for each day of the week):

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_6

```
public class TemperatureReadings
{
   public static void main(String[] args)
   {
      // declare 7 variables to hold readings
      double    temperature1, temperature2, temperature3, temperature4, temperature5,
                temperature6, temperature7;
      // more code will go here
   }
}
```

Here we have declared seven variables each of type **`double`** (as temperatures will be recorded as real numbers). So far so good. Now to write some code that allows the user to enter values for these temperatures. Getting one temperature is easy (assuming we have created a `Scanner` object, `keyboard`), as shown below:

```
System.out.println("max temperature for day 1 ?");
temperature1 = keyboard.nextDouble();
```

But how would you write the code to get the second temperature, the third temperature and all the remaining temperatures? Well, you could repeat the above pair of lines for each temperature entered, but surely you've got better things to do with your time!

Essentially you want to repeat the same pair of lines seven times. You already know that a **`for`** loop is useful when repeating lines of code a fixed number of times. Maybe you could try using a **`for`** loop here?

```
for (int i=1; i<=7; i++)
{
   // what goes here?
}
```

This looks like a neat solution, but the problem is that there is no obvious instruction we could write in the **`for`** loop that will allow a value to be entered into a *different* variable each time the loop repeats. As things stand there is no way around this, as each variable has a *distinct* name.

Ideally we would like each variable to be given the *same* name (`temperature`, say) so that we could use a loop here, but we would like some way of being able to distinguish between each successive variable. In fact, this is exactly what an **array** allows us to do.

6.2 Creating an Array

An array is a special data type in Java that can be thought of as a *container* to store a *collection of items*. These items are sometimes referred to as the **elements** of the array. All the elements stored in a particular array must be of the *same type* but there

is no restriction on which type this is. So, for example, an array can be used to hold a collection of **int** values or a collection of **char** values, but it cannot be used to hold a mixture of **int** and **char** values.

Let's look at how to use arrays in your programs. First you need to know how to create an array. Array creation is a two-stage process:

1. declare an array variable;
2. allocate memory to store the array elements.

An array variable is declared in much the same way as a simple variable except that a pair of square brackets is added after the type. For example, if an array was to hold a collection of integer variables it could be declared as follows:

```
int[] someArray;
```

Here a name has been given to the array in the same way you would name any variable. The name we have chosen is `someArray`. If the square brackets were missing in the above declaration this would just be a simple variable capable of holding a *single* integer value only. But the square brackets indicate this variable is an array allowing *many* integer values to be stored.

So, to declare an array `temperature` containing **double** values, you would write the following:

```
double[] temperature;
```

At the moment this simply defines `temperature` to be a variable that can be *linked* to a collection of **double** values. The `temperature` variable itself is said to hold a **reference** to the array elements. A reference is a variable that holds a *location* in the computer's memory (known as a memory *address*) where data is stored, rather than the data itself. This illustrated in Fig. 6.1.

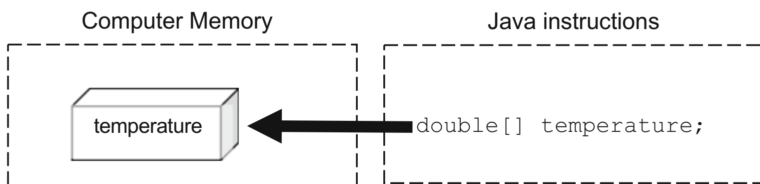


Fig. 6.1 The effect on computer memory of declaring an array reference

At the moment the memory address stored in the temperature reference is not meaningful as the memory that will eventually hold the array elements has not been allocated yet. This is stage two.

What information do you think would be required in order to reserve enough space in the computer's memory for all the array elements?

Well, it would be necessary to state the size of the array, that is the *maximum* number of elements required by the array. Also, since each data type requires a different amount of memory space, it is also necessary to state the type of each individual array element (this will be the same type used in stage one of the array declaration). The array type and size are then put together with a special **new** operator. For example, if we required an array of 10 integers the following would be appropriate.

The **new** operator creates the space in memory for an array of the given size and element type.[1]

```
someArray = new int[10];
```

We will come back to look at this **new** operator when looking at classes and objects in the next chapter. Once the size of the array is set it cannot be changed, so always make sure you create an array that is big enough for your purpose. Returning to the temperature example above, if you wanted the array to hold seven temperatures you would allocate memory as follows:

```
temperature = new double[7];
```

Let's see what effect the **new** operator has on computer memory by looking at Fig. 6.2.

As can be seen from Fig. 6.2, the array creation has created seven continuous locations in memory. Each location is big enough to store a value of type **double**. The temperature variable is linked to these seven elements by containing the address of the very first element. In effect, the array reference, temperature, is linked to seven new variables. Each of these variables will automatically have some initial value placed in them. If the variables are of some number type (such as **int** or **double**) the value of each will be initially set to zero; if the variables are of type **char** their values will be set initially to a special Unicode value that represents an empty character; if the variables are of **boolean** type they will each be set initially to **false**.

[1]Of course this size should not be a negative value. A negative value will cause an error in your program. We will discuss these kinds of errors further in Chap. 14.

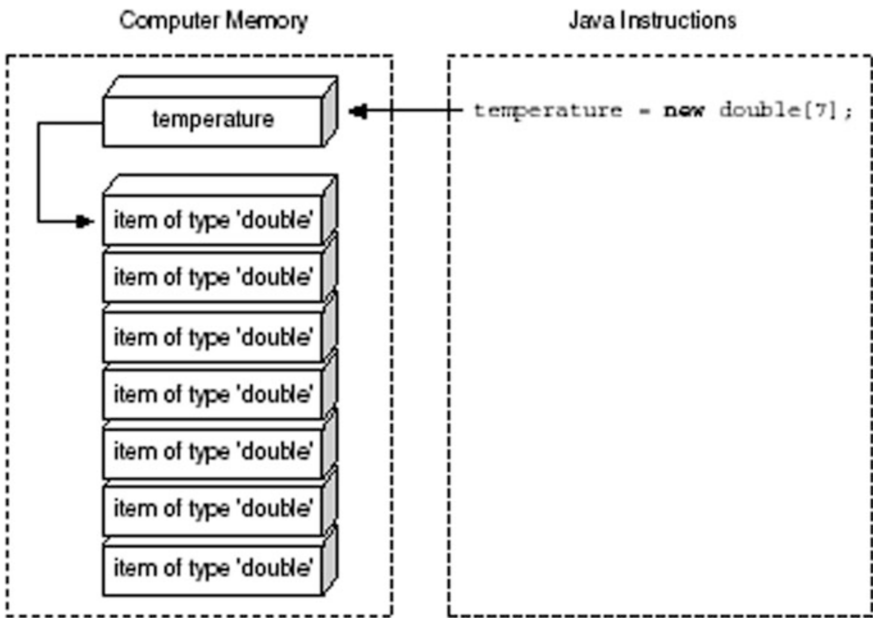


Fig. 6.2 The effect on computer memory of declaring an array of seven 'double' values

The two stages of array creation (declaring and allocating memory space for the elements) can also be combined into one step as follows:

```
double[] temperature = new double[7];
```

You may be wondering: what names have each of these variables been given? The answer is that each element in an array shares the same name as the array, so in this case each element is called `temperature`. The individual elements are then *uniquely identified* by an additional **index value**. An index value acts rather like a street number to identify houses on the same street (see Fig. 6.3). In much the same way as a house on a street is identified by the street name and a house number, an array element is identified by the array name and the index value.

Like a street number, these index values are always contiguous integers. Note carefully that, in Java, *array indices start from* 0 *and not from* 1. This index value is always enclosed in square brackets, so the first temperature in the list is identified as `temperature[0]`, the second temperature by `temperature[1]` and so on.

This means that the size of the array and the last index value are *not* the same value. In this case the size is 7 and the last index is 6. There is no such value as `temperature[7]`, for example. Remember this, as it is a common cause of

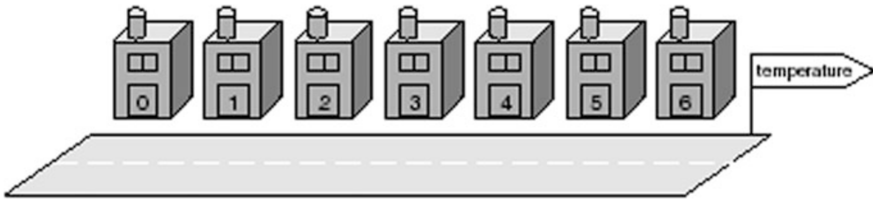


Fig. 6.3 Elements in an array are identified in much the same way as houses on a street

errors in programs! If you try to access an invalid element such as `temperature[7]`, the following program error will be generated by the system:

```
java.lang.ArrayIndexOutOfBoundsException
```

This type of error is called an *exception*. You will find out more about exceptions in the second semester but you should be aware that, very often, exceptions will result in program termination.

Usually, when an array is created, values will be added into it as the program runs. If, however, all the values of the array elements are known beforehand, then an array can be created without the use of the **new** operator by initializing the array as follows:

```
double[] temperature = {9, 11.5, 11, 8.5, 7, 9, 8.5} ;
```

The initial values are placed in braces and separated by commas. The compiler determines the length of the array by the number of initial values (in this case 7). Each value is placed into the array in order, so `temperature[0]` is set to 9, `temperature[1]` to 11.5 and so on. This is the only instance in which *all the elements* of an array can be assigned explicitly by listing out the elements in a single assignment statement.

Once an array has been created, elements must be accessed *individually*.

6.3 Accessing Array Elements

Once an array has been created, its elements can be used like any other variable of the given type in Java. If you look back at the temperature example, initializing the values of each temperature when the array is created is actually quite unrealistic. It is much more likely that temperatures would be entered into the program as it runs. Let’s look at how to achieve this.

Whether an array is initialized or not, values can be placed into the individual array elements. We know that each element in this array is a variable of type **double**. As with any variable of a primitive type, the assignment operator can be used to enter a value.

The only thing you have to remember when using the assignment operator with an array element is to specify *which* element to place the value in. For example, to allow the user of the program to enter the value of the first temperature, the following assignment could be used (again, assuming the existence of a Scanner object, keyboard):

```
temperature[0] = keyboard.nextDouble();
```

Note again that, since array indices begin at 0, the first temperature is not at index 1 but index 0.

Array elements could also be printed on the screen. For example, the following command prints out the value of the *sixth* array element:

```
System.out.println(temperature[5]); // index 5 is the sixth element!
```

Note that an array index (such as 5) is just used to *locate* a position in the array; it is *not* the item at that position.

For example, assume that the user enters a value of 25.5 for the first temperature in the array; the following statement:

```
System.out.println("temperature for day 1 is " + temperature[0]);
```

would then print out the message:

```
temperature for day 1 is 25.5
```

Statements like the println command above might seem a bit confusing at first. The message refers to "temperature for day **1**" but the temperature that is displayed is temperature[0]. Remember though that the temperature at index position 0 *is* the first temperature! After a while you will get used to this indexing system.

As you can see from the examples above, you can use array elements in exactly the same way you can use any other kind of variable of the given type. Here are a few more examples:

```
temperature[4] = temperature[4] * 2;
```

This assignment doubles the value of the *fifth* temperature. The following **`if`** statement checks if the temperature for the *third* day was a hot temperature:

```
if (temperature[2] >= 18)
{
   System.out.println("it was hot today");
}
```

So far so good, but if you are just going to use array elements in the way you used regular variables, why bother with arrays at all?

The reason is that the indexing system of arrays is in fact a very powerful programming tool. The index value does not need to be a literal number such as 5 or 2 as in the examples we have just shown you; it can be *any expression that evaluates to an integer*.

More often than not an integer *variable* is used, in place of a fixed index value, to access an array element. For example, if we assume that `i` is some integer variable, then the following is a perfectly legal way of accessing an array element:

```
System.out.println(temperature[i]); // index is a variable
```

Here the array index is not a literal number (like 2 or 5) but the variable `i`. The value of `i` will determine the array index. If the value of `i` is 4 then this will display `temperature[4]`, if the value of `i` is 6 then this will display `temperature[6]`, and so on. One useful application of this is to place the array instructions within a loop (usually a **`for`** loop), with the loop counter being used as the array index. For example, returning to the original problem of entering all seven temperature readings, the following loop could now be used:

```
for(int i = 0; i<7; i++) // note, loop counter runs from 0 to 6
{
   System.out.println("enter max temperature for day "+(i+1));
   temperature[i] = keyboard.nextDouble(); // use loop counter
}
```

Note carefully the following points from this loop:

- Unlike many of the previous examples of **for** loop counters that started at 1, this counter starts at 0. Since the counter is meant to track the array indices, 0 is the appropriate number to start from.
- The counter goes up to, but does not include, the number of items in the array. In this case this means the counter goes up to 6 and not 7. Again this is because the array index for an array of size 7 stops at 6.
- The `println` command uses the loop counter to display the number of the given day being entered. The loop counter starts from 0, however. We would not think of the first day of the week as being day 0! In order for the message to be more meaningful for the user, therefore, we have displayed (`i + 1`) rather than `i`.

Effectively the following statements are executed by this loop:

```
System.out.println("enter max temperature for day 1 ");    } 1st time round loop
temperature[0] = keyboard.nextDouble();

System.out.println("enter max temperature for day 2 ");    } 2nd time round loop
temperature[1] = keyboard.nextDouble();

//as above but with indices 2-5                             } 3rd–6th time round loop

System.out.println("enter max temperature for day 7 ");    } 7th time round loop
temperature[6] = keyboard.nextDouble();
```

You should now be able to see the benefit of an array. This loop can be made more readable if we make use of a built-in feature of all arrays that returns the length of an array. It is accessed by using the word `length` after the name of the array. The two are joined by a full stop. Here is an example:

```
System.out.print("number of temperatures = ");
System.out.println(temperature.length); // returns the size of the array
```

which displays the following on to the screen:

```
number of temperatures = 7
```

Note that `length` feature returns the size of the array, not necessarily the number of items currently stored in the array (which may be fewer). This attribute can be used in place of a fixed number in the **for** loop as follows:

```
for (int i = 0; i < temperature.length, i++)
{
   // code for loop goes here
}
```

To see this technique being exploited, look at the program below to see the completed `TemperatureReadings` program, which stores and displays the maximum daily temperatures in a week.

TemperatureReadings

```
import java.util.Scanner;

public class TemperatureReadings
{
   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner(System.in);
      // create array
      double[] temperature;
      temperature = new double[7];
      // enter temperatures
      for (int i = 0; i < temperature.length; i++)
      {
         System.out.println("enter max temperature for day " + (i+1));
         temperature[i] = keyboard.nextDouble();
      }
      // display temperatures
      System.out.println(); // blank line
      System.out.println("***TEMPERATURES ENTERED***");
      System.out.println(); // blank line
      for (int i = 0; i < temperature.length; i++)
      {
         System.out.println("day "+(i+1)+" "+ temperature[i]);
      }
   }
}
```

Note how `length` was used to control the two **`for`** loops. Here is a sample test run.

```
enter max temperature for day 1 12.2
enter max temperature for day 2 10.5
enter max temperature for day 3 13
enter max temperature for day 4 15
enter max temperature for day 5 13
enter max temperature for day 6 12.5
enter max temperature for day 7 12

***TEMPERATURES ENTERED***

day 1 12.2
day 2 10.5
day 3 13.0
day 4 15.0
day 5 13.0
day 6 12.5
day 7 12.0
```

6.4 Passing Arrays as Parameters

In Chap. 5 we looked at how methods can be used to break up a programming task into manageable chunks. Methods can receive data in the form of parameters and can send back data in the form of a return value. Arrays can be used both as parameters to methods and as return values. In the next section we will see an example of an array as a return value from a method. In this section we will look at passing arrays as parameters to a method. As an example of passing an array to a method, consider once again the TemeratureReadings program from the previous section. That program contains all the processing within the main method. As a result, the code for this method is a little difficult to read. Let's do something about that. We will create two methods, enterTemps and displayTemps, to enter and display temperatures respectively. To give these methods access to the array they must receive it as a parameter. Here, for example, is the header for the enterTemps method. Notice that when a parameter is declared as an array type, the size of the array is not required but the empty square brackets are:

```
static void enterTemps( double[] temperatureIn )
{
    // rest of method goes here
}
```

Now, although in the previous chapter we told you that a parameter just receives a copy of the original variable, this works a little differently with arrays. We will explain this a little later, but for now just be aware that this method will actually fill the original array. The code for the method itself is straightforward:

```
Scanner keyboard = new Scanner(System.in); // create local Scanner object
for (int i = 0; i < temperatureIn.length; i++)
{
    System.out.println("enter max temperature for day " + (i+1));
    temperatureIn[i] = keyboard.nextDouble();
}
```

Similarly the displayTemps method will require the array to be sent as a parameter. The program below rewrites our previous program by adding the two methods mentioned above. Take a look at it and then we will discuss it.

TemperatureReadings2

```
import java.util.Scanner;
public class TemperatureReadings2
{
    public static void main(String[] args)
    {
        double[] temperature;
        temperature = new double[7];
        enterTemps(temperature); // call method
        displayTemps(temperature); // call method
    }
    // method to enter temperatures
    static void enterTemps(double[] temperatureIn)
    {
        Scanner keyboard = new Scanner(System.in);
        for (int i = 0; i < temperatureIn.length; i++)
        {
            System.out.println("enter max temperature for day " + (i+1));
            temperatureIn[i] = keyboard.nextDouble();
        }
    }

    // method to display temperatures
    static void displayTemps(double[] temperatureIn)
    {
        System.out.println();
        System.out.println("***TEMPERATURES ENTERED***");
        System.out.println();
        for (int i = 0; i < temperatureIn.length; i++)
        {
            System.out.println("day "+(i+1)+" "+ temperatureIn[i]);
        }
    }
}
```

Notice that when sending an array as a parameter, the array name alone is required:

```
public static void main(String[] args)
{
   double[] temperature;
   temperature = new double[7];
   enterTemps(temperature); // array name sent in
   displayTemps(temperature); // array name sent in
}
```

Now let us return to the point we made earlier. You are aware that, in the case of a simple variable type such as an **`int`**, it is the *value* of the variable that is copied when it is passed as a parameter. This means that if the value of a parameter is altered within a method, the original variable is unaffected outside that method. This works differently with arrays.

As we said earlier, the `enterTemps` method actually fills the *original* array. How can this be? The answer is that in the case of arrays, the value sent as a parameter is not a copy of each array element but, instead, a copy of the array *reference*. In other words, the *location* of the array is sent to the receiving method not the value of the contents of the array. Now, even though the receiving parameter (`temperatureIn`) has a different name to the original variable in `main` (`temperature`), they are both pointing to the same place in memory so both are modifying the same array! This is illustrated in Fig. 6.4: Sending the array reference to a method rather than a copy of the whole array is a very efficient use of the computer's resources, especially when arrays become very large.

6.5 Varargs

Look back at the `displayTemps` method given in Sect. 6.4 above. By declaring the parameter to be an array of **`double`** values, the `displayTemps` method is able to work with *any number of double values* contained in the given array parameter:

```
static void displayTemps(double[] temperatureIn) // can accept an array of any size
{
   // code to display temperatures goes here
}
```

In our `TemperatureReadings2` program we created an array of size 7, but we could have created an array of any size—for example an array of size 365 for each day of the year. This `displayTemps` method would still correctly receive this data and display all temperatures, irrespective of the size of an array.

Very closely related to the idea of sending an array to a method is a Java feature called **varargs**, which is short for **variable arguments**. Just like sending an array, the *varargs* feature allows us to send a variable number of data items to a method, as long as each item is of the same type, without having to fix the number of data items in any way. We can re-write the `displayTemps` header by using the *varargs* syntax as follows:

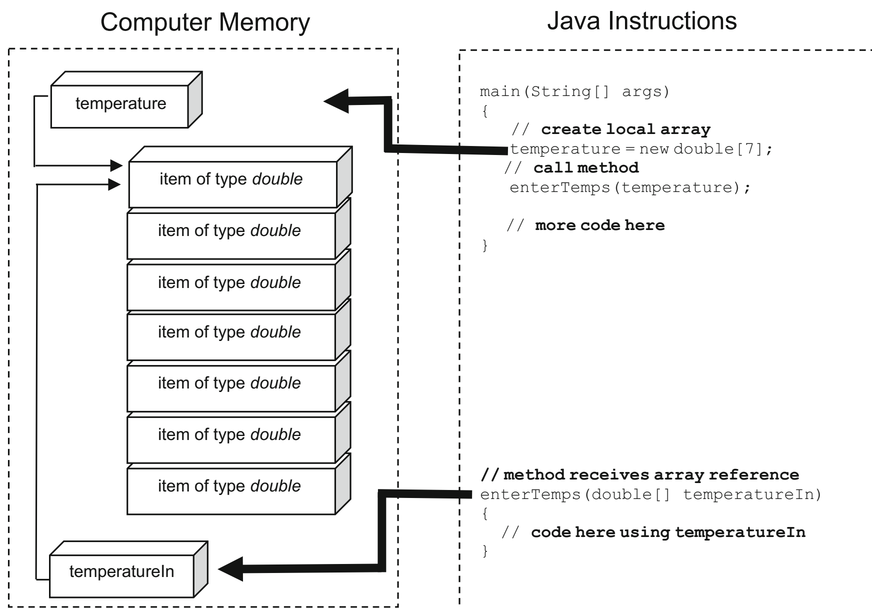


Fig. 6.4 The effect on computer memory of passing an array as a parameter

```
static void displayTemps(double... temperatureIn) // note the varargs syntax
{
   // code to display temperatures goes here
}
```

As you can see, instead of using the array syntax of square brackets (**`double []`**) we use the *varargs* syntax (`double...`)—as you can see this consists of an ellipsis (three consecutive dots). This indicates that a variable number (zero or more) of **`double`** values will be sent to the method. The code inside the method remains the same as the *varargs* parameter, `temperatureIn` in this case, is implicitly converted into an array within the method.

```
static void displayTemps(double... temperatureIn) // temperatureIn converted to an array
{
   // code inside the method remains the same
   System.out.println();
   System.out.println("***TEMPERATURES ENTERED***");
   System.out.println();
   for (int i = 0; i < temperatureIn.length; i++)
   {
      System.out.println("day "+(i+1)+" "+ temperatureIn[i]);
   }
}
```

You may be thinking, if *varargs* is just another notation for sending an array of values, why not just stick to array syntax? Well, while the code for the method remains the same we are given more flexible ways of calling this method. We can send an array as before, but we can also send individual values if we wish. The `DisplayTemperaturesWithVarargs` program below illustrates this. Take a look at it and then we will discuss it.

DisplayTemperaturesWithVarargs

```
public class DisplayTemperaturesWithVarargs
{
    public static void main(String[] args)
    {
       double[] temperature = {7.5, 8.2, 7.7, 11.3, 10.75}; // create array with 5 readings
       System.out.println("Sending Array");
       displayTemps(temperature); // call method with a single array
       System.out.println();
       System.out.println("Sending individual items");
       displayTemps(7.5, 8.2, 7.7, 11.3, 10.75); // call method with 5 individual readings
       displayTemps(9.9); // call method with 1 value only
       displayTemps( ); // call method with no values
    }

    // method to display temperatures using varargs
    static void displayTemps(double... temperatureIn)
    {
        System.out.println();
        System.out.println("***TEMPERATURES***");
        System.out.println("Number of temperatures: "+ temperatureIn.length); // count items
        // display temperatures
        for (int i = 0; i < temperatureIn.length; i++)
        {
            System.out.println(temperatureIn[i]+ "  ");
        }
    }
}
```

You can see, in the `main` method, we have initialised an array with five temperature readings:

```
double[] temperature = {7.5, 8.2, 7.7, 11.3, 10.75};
```

Our `displayTemps` method has been written to accept a *varargs* collection of values:

```
// method to display temperatures using varargs
static void displayTemps(double... temperatureIn)
{
   // code to display temperatures goes here
}
```

One way to call this method is with the array of five temperatures we created:

```
displayTemps(temperature); // call method with a single array
```

However, we can also send individual values to a *varargs* parameter rather than using an array. To illustrate we have called the `displayTemps` method with the same values that were in our array, but sent as individual values rather than stored in an array:

```
displayTemps(7.5, 8.2, 7.7, 11.3, 10.75); // call method with 5 individual readings
```

We can send any number of values in this way, including a single value or no values at all:

```
displayTemps(9.9); // call method with 1 value only
displayTemps( ); // call method with no values
```

Here is a program run to clarify the results:

```
Sending Array

***TEMPERATURES***
Number of temperatures: 5
7.5
8.2
7.7
11.3
10.75
```

```
Sending individual items

***TEMPERATURES***
Number of temperatures: 5
7.5
8.2
7.7
11.3
10.75

***TEMPERATURES***
Number of temperatures: 1
9.9

***TEMPERATURES***
Number of temperatures: 0
```

Notice that if you wish to send additional parameters to a method as well as a *varargs* parameter, the *varargs* parameter must come last in the parameter list. So, for example, the following method header would not compile:

```
// this method header will cause a compiler error
static void someMethd(int... varargParam, int param2)
{
   // code to for method goes here
}
```

Here we have two parameters, a *vararg* collection of integers, `varagsParam`, and a single integer parameter, `param2`. This will cause a compiler error as the *varargs* parameter should be the last parameter in the list. The correct method header is given as follows:

```
// this method header will not cause a compiler error
static void someMethd(int param1, int... varargParam)
{
   // code to for method goes here
}
```

6.6 Returning an Array from a Method

A method can return an array as well as receive arrays as parameters. As an example, let us reconsider the `enterTemps` method from the `TemperatureReading2` program of Sect. 6.4 again. At the moment, this method accepts an array as a parameter and fills this array with temperature values. Since this method fills the

original array sent in as a parameter, it does not need to return a value—its return type is therefore **void**:

```
static void enterTemps(double[] temperatureIn)
{
   // code to fill the parameter, 'temperatureIn', goes here
}
```

An alternative approach would be *not* to send an array to this method but, instead, to create an array *within* this method and fill *this* array with values. This array can then be returned from the method:

```
// this method receives no parameters but returns an array of doubles

static double[] enterTemps()
{
   Scanner keyboard = new Scanner(System.in);
   // create an array within this method
   double[] temperatureOut = new double[7];
   // fill up the array created in this method
   for (int i = 0; i < temperatureOut.length; i++)
   {
      System.out.println("enter max temperature for day " + (i+1));
      temperatureOut[i] = keyboard.nextDouble();
   }
   // send back the array created in this method
   return temperatureOut;
}
```

As you can see, we use square brackets to indicate that an array is to be returned from a method:

```
static double[] enterTemps()
```

The array itself is created within the method. We have decided to call this array `temperatureOut`:

```
double[] temperatureOut = new double[7];
```

After the array has been filled it is sent back with a **return** statement. Notice that, to return an array, the name alone is required:

```
return temperatureOut;
```

Now that we have changed the `enterTemps` method, we need to revisit the original `main` method also. It will no longer compile now that the `enterTemps` method has changed:

```
// the original 'main' method will no loger compile!
public static void main(String[] args)
{
    double[] temperature = new double[7];
    enterTemps(temperature); // this line will now cause a compiler error !!
    displayTemps(temperature);
}
```

The call to `enterTemps` will no longer compile as the new `enterTemps` does not expect to be given an array as a parameter. The correct way to call the method is as follows:

```
enterTemps(); // this method requires no parameter
```

However, this method now *returns* an array. We really should do something with the array value that is returned from this method. We should use the returned array value to set the value of the original `temperature` array:

```
// just declare the 'temperature' array but do not allocate it memory yet
double[] temperature;
// 'temperature' array is now set to the return value of 'enterTemps'
temperature = enterTemps();
```

As you can see, we have not sized the `temperature` array once it has been declared. Instead the `temperature` array will be set to the size of the array returned by `enterTemps`, and it will contain all the values of the array returned by `enterTemps`. The complete program, `TemperatureReadings3`, is shown below:

TemperatureReadings3

```
import java.util.Scanner;
public class TemperatureReadings3
{
    public static void main(String[] args)
    {
        double[] temperature ;
        temperature = enterTemps(); // call new version of this method
        displayTemps(temperature);
    }

    // method to enter temperatures returns an array
    static double[] enterTemps()
    {
        Scanner keyboard = new Scanner(System.in);
        double[] temperatureOut = new double[7];
        for (int i = 0; i < temperatureOut.length; i++)
        {
            System.out.println("enter max temperature for day " + (i+1));
            temperatureOut[i] = keyboard.nextDouble();
        }
        return temperatureOut;
    }

    // this method is unchanged
    static void displayTemps(double[] temperatureIn)
    {
        System.out.println();
        System.out.println("***TEMPERATURES ENTERED***");
        System.out.println();
        for (int i = 0; i < temperatureIn.length; i++)
        {
            System.out.println("day "+(i+1)+" "+ temperatureIn[i]);
        }
   }
}
```

This program behaves in exactly the same way as the previous one, so whichever way you implement `enterTemps` is really just a matter of preference.

6.7 The Enhanced 'for' Loop

As you can see from the examples above, when processing an entire array a loop is required. Very often, this will be a **`for`** loop. With a **`for`** loop, the loop counter is used as the array index within the body of the loop. In the examples above, the loop counter was used not only as an array index but also to display meaningful messages to the user. For example:

```
for(int i = 0; i < temperature.length; i++)
{
   System.out.println("day " + (i+1) + " " + temperature[i]);
}
```

Here the loop counter was used to determine the day number to display on the screen, as well as the index of an array element. Very often, when a **`for`** loop is required, the *only* use made of the loop counter is as an array index to access all the elements of the array consecutively. Java provides an enhanced version of the **`for`** loop especially for this purpose.

Rather than use a counter, the enhanced **for** loop consists of a variable that, upon each iteration, stores consecutive elements from the array.[2] For example, if we wished to display on the screen each value from the `temperature` array, the enhanced **for** loop could be used as follows:

```
/* the enhanced for loop iterates through elements of an array without the need
  for an array index */

for (double item : temperature) // see discussion below
{
   System.out.println(item);
}
```

In this case we have named each successive array element as `item`. The loop header is to be read as "for each `item` in the `temperature` array". For this reason the enhanced **for** loop is often referred to as the *for each* loop. Notice that the type of the array item also has to be specified. The type of this variable is **double** as we are processing an array of **double** values. Remember that this is the type of each *individual* element within the array.

You should note that the variable `item` can be used only within the loop, we cannot make reference to it outside the loop. Within the body of the loop we can now print out an array element by referring directly to the `item` variable rather than accessing it via an index value:

```
System.out.println(item); // 'item' is an array element
```

This is a much neater solution than using a standard **for** loop, which would require control of a loop counter in the loop header, and array look up within the body of the loop.

You will remember, from Sect. 6.5, that *varargs* parameters are implicitly turned into arrays when they are received. So the enhanced **for** loop can also be used with such parameters. Here, for example, is the `displayTemps` method of the `DisplayTemperaturesWithVarargs` program, from Sect. 6.5, re-written with an enhanced **for** loop:

```
// method to display temperatures using varargs
static void displayTemps(double... temperatureIn)// temperaturenIn implictly convereted to array
{
   // previous code here

   // dipslay items using enhanced for loop
   for (double item: temperatureIn)
   {
      System.out.println(item+ "  ");
   }
}
```

[2]The enhanced **for** loop also works with other classes in Java, which act as alternatives to arrays. We will explore some of these classes in Chap. 15.

Be aware that the enhanced **for** loop should *not* be used if you wish to modify the array items. Modifying array items with such a loop will not cause a compiler error, but it is unsafe as it may cause your program to behave unreliably. So you should use an enhanced **for** loop only when:

- you wish to access the *entire* array (and not just part of the array);
- you wish to *read* the elements in the array, not *modify* them;
- you do not require the array index for additional processing.

Very often, when processing an array, it is the case that these three conditions apply. In the following sections we will make use of this enhanced **for** loop where appropriate.

6.8 Some Useful Array Methods

Apart from the `length` feature, an array does not come with any useful built in routines. So we will develop some of our own methods for processing an array. We will use a simple integer array for this example. Here is the outline of the program we are going to write in order to do this:

```
import java.util.Scanner;

public class SomeUsefulArrayMethods
{
    public static void main (String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        int[] someArray; // declare an integer array
        // ask user to determine size of array
        System.out.println("How many elements to store?");
        int size = keyboard.nextInt();
        // size array now
        someArray = new int[size];

        // call methods here
    }
    // methods to process an array here
}
```

As you can see, we have delayed the second stage of array creation here until the user tells us how many elements to store in the array. Now to some methods.

6.8.1 Array Maximum

The first method we will develop will allow us to find the maximum value in an array. For example, we may have a list of scores and wish to know the highest score in this list. Finding the maximum value in an array is a much better approach than

the one we took in Chap. 5, where we looked at a method to find the maximum of two values and another method to find the maximum of three values. This array method can instead be used with lists of two, three, four or any other number of values. The approach we will use will be similar to the max method we developed in Chap. 5 for finding the maximum of three values. Here is the pseudocode again.

```
SET result TO first number
IF second number > result
BEGIN
  SET result TO second number
END
IF third number > result
BEGIN
  SET result TO third number
END
RETURN result
```

Here, the final result is initialized to the first value. All other values are then compared with this value to determine the largest value. Now that we have an array, we can use a loop to process this comparison, rather than have a series of many **if** statements. Here is a suitable algorithm:

```
SET result TO first value in array
LOOP FROM second element in array TO last element in array
BEGIN
    IF current element > result
    BEGIN
        SET result TO current element
    END
END
RETURN result
```

This method will need the array that it has to search to be sent in as a parameter. Also, this method will return the maximum item so it must have an integer return type.

```
static int max (int[] arrayIn)
{
    int result = arrayIn[0]; // set result to the first value in the array

    // this loops runs from the 2nd item to the last item in the array
    for (int i = 1; i < arrayIn.length; i++)
    {
        if(arrayIn[i] > result)
        {
            result = arrayIn[i]; // reset result to new maximum
        }
    }
    return result;
}
```

Notice we did not use the enhanced **for** loop here, as we needed to iterate from the *second* item in the array rather than through *all* items, and the standard **for** loop gives us this additional control.

6.8.2 Array Summation

The next method we will develop will be a method that calculates the total of all the values in the array. Such a method might be useful, for example, if we had a list of deposits made into a bank account and wished to know the total value of these deposits. A simple way to calculate the sum is to keep a running total and add the value of each array element to that running total. Whenever you have a running total it is important to initialize this value to zero. We can express this algorithm using pseudocode as follows:

```
SET total TO zero
LOOP FROM first element in array TO last element in array
BEGIN
   SET total TO total + value of current element
END
RETURN total
```

This method will again need the array to be sent in as a parameter, and will return an integer (the value of the sum), giving us the following:

```
static int sum(int[] arrayIn)
{
   int total = 0;
   for (int currentElement : arrayIn)
   {
      total = total + currentElement;
   }
   return total;
}
```

Notice the use of the enhanced **for** loop here—as we need to iterate through *all* elements within the array.

6.8.3 Array Membership

It is often useful to determine whether or not an array contains a particular value. For example, if the list were meant to store a collection of unique student ID numbers, this method could be used to check if a new ID number already exists before adding it to the list. A simple technique is to check each item in the list one by one, using a loop, to see if the given value is present. If the value is found the loop is exited. If the loop reaches the end without exiting then we know the item is not present. Here is the pseudocode:

```
LOOP FROM first element in array TO last element in array
BEGIN
  IF current element = item to find
  BEGIN
       EXIT loop and RETURN true
  END
END
RETURN false
```

Notice in the algorithm above that a value of **`false`** would be returned only if the given item is not found. If the value is found, the loop would terminate without reaching its end and a value of **`true`** would be returned.

Here is the Java code for this method. We need to ensure that this method receives the array to search and the item being searched for. Also the method must return a **`boolean`** value:

```
static boolean contains(int[] arrayIn, int valueIn)
{
    // enhanced 'for' loop used here
    for (int currentElement : arrayIn)
    {
    if (currentElement == valueIn)
        {
            return true; // exit loop early if value found
        }
    }
    return false; // value not present
}
```

6.8.4 Array Search

One of the most common tasks relating to a list of values is to determine the position of an item within the list. For example, we may wish to know the position of a job waiting in a printer queue.

How will we go about doing this?

Just as we did when devising the `contains` algorithm, we will need to use a loop to examine every item in the array. Inside the loop we check each item one at a time and compare it to the item we are searching for.

What do we do if we find the item we are searching for? Well, we are interested in its position in the array so we just return the index of that item.

Now we need to decide what we do if we reach the end of the loop, having checked all the elements in the array, without finding the item we are searching for. This method needs to return an integer regardless of whether or not we find an item. What number shall we send back if the item is not found? We need to send back a

value that could never be interpreted as an array index. Since array indices will always be positive numbers we could send back a negative number, such as −999, to indicate a valid position has not been found.

Here is the pseudocode:

```
LOOP FROM first element in array TO last element in array
BEGIN
  IF current element = item to find
  BEGIN
        EXIT loop and RETURN current index
  END
END
RETURN -999
```

This approach is often referred to as a *linear search*. Here is the Java code for this method. Once again we need to ensure that this method receives the array to search and the item being searched for. This method must return an integer value:

```
static int search (int[] arrayIn, int valueIn)
{
    // enhanced 'for' loop should not be used here!
    for (int i=0; i < arrayIn.length; i++)
    {
        if (arrayIn[i] == valueIn)
        {
            return i; // exit loop with array index
        }
    }
    return -999; // indicates value not in list
}
```

Notice, in this case, we could not use the enhanced **`for`** loop. The reason for this is that we required the method to return the array index of the item we are searching for. This index is best arrived at by making use of a loop counter in a standard **`for`** loop.

6.8.5 The Final Program

The complete program for manipulating an array is now presented below. The array methods are accessed via a menu. We have included some additional methods here for entering and displaying an array:

SomeUsefulArrayMethods

```
import java.util.Scanner;

// a menu driven program to test a selection of useful array methods

public class SomeUsefulArrayMethods
{
    public static void main (String[] args)
    {
        char choice;
        Scanner keyboard = new Scanner (System.in);
        int[] someArray; // declare an integer array
        System.out.print ("How many elements to store?: ");
        int size = keyboard.nextInt ();
        // size the array
        someArray = new int [size];
        // menu
        do
        {
            System.out.println();
            System.out.println("[1] Enter values");
            System.out.println("[2] Find maximum");
            System.out.println("[3] Calculate sum");
            System.out.println("[4] Check membership");
            System.out.println("[5] Search array");
            System.out.println("[6] Display values");
            System.out.println("[7] Exit");
            System.out.print("Enter choice [1-7]: ");
            choice = keyboard.next().charAt(0);
            System.out.println();
            // process choice by calling additional methods
            switch(choice)
            {
               case '1':  fillArray(someArray);
                          break;
               case '2':  int max = max(someArray);
                          System.out.println("Maximum array value = " + max); break;
               case '3':  int total = sum(someArray);
                          System.out.println("Sum of array values = " + total); break;
               case '4':  System.out.print ("Enter value to find: ");
                          int value = keyboard.nextInt();
                          boolean found = contains(someArray, value);
                          if (found)
                          {
                              System.out.println(value + " is in the array");
                          }
                          else
                          {
                              System.out.println(value + " is not in the array");
                          }
                          break;
               case '5':  System.out.print ("Enter value to find: ");
                          int item = keyboard.nextInt();
                          int index = search(someArray, item);
                          if (index == -999) // indicates value not found
                          {
                              System.out.println ("This value is not in the array");
                          }
                          else
                          {
                              System.out.println ("This value is at array index " + index);
                          }
                          break;
               case '6':  System.out.println("Array values");
                          displayArray(someArray);
                          break;
            }
       } while (choice != '7');
       System.out.println("Goodbye");
   }

   // additional methods

   // fills an array with values
   static void fillArray(int[] arrayIn)
   {
       Scanner keyboard = new Scanner (System.in);
       for (int i = 0; i < arrayIn.length; i++)
       {
           System.out.print("enter value ");
           arrayIn[i] = keyboard.nextInt();
       }
   }
```

```
    // returns the total of all the values held within an array
    static int sum (int[] arrayIn)
    {
        int total = 0;
        for (int currentElement : arrayIn)
        {
            total = total + currentElement;
        }
        return total;
    }

    // returns the maximum value in an array
    static int max (int[] arrayIn)
    {
        int result = arrayIn[0]; // set result to the first value in the array
        // this loops runs from the 2nd item to the last item in the array
        for (int i=1; i < arrayIn.length; i++)
        {
            if (arrayIn[i] > result)
            {
                result = arrayIn[i]; // reset result to new maximum
            }
        }
        return result;
    }
    // checks if a given item is contained within the array
    static boolean contains (int[] arrayIn, int valueIn)
    {
        // enhanced 'for' loop used here
        for (int currentElement : arrayIn)
        {
            if (currentElement == valueIn)
            {
                return true; // exit loop early if value found
            }
        }
        return false; // value not present
    }

    /* returns the position of an item within an array or -999 if the value is not present within the array */
    static int search (int[] arrayIn, int valueIn)
    {
        for (int i = 0; i < arrayIn.length; i++)
        {
            if (arrayIn[i] == valueIn)
            {
                return i;
            }
        }
        return -999;
    }

    // displays the array values on the screen
    static void displayArray(int[] arrayIn)
    {
        System.out.println();
        // standard 'for' loop used here as the array index is required
        for (int i = 0; i < arrayIn.length; i++)
        {
            System.out.println("array[" + i + "] = " + arrayIn[i]);
        }
    }
}
```

Here is a sample program run:

```
How many elements to store?: 5

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
```

```
Enter choice [1-7]: 1

enter value 12
enter value 3
enter value 7
enter value 6
enter value 2

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 2

Maximum array value = 12

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 3

Sum of array values = 30

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit Enter choice [1-7]: 4

Enter value to find: 10

10 is not in the array

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
```

```
[6] Display values
[7] Exit
Enter choice [1-7]: 4

Enter value to find: 7

7 is in the array

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 5

Enter value to find: 7
This value is at array index 2

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 6

Array values

array[0] = 12
array[1] = 3
array[2] = 7
array[3] = 6
array[4] = 2

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 7

Goodbye
```

6.9 Multi-dimensional Arrays

In the temperature reading example we used at the beginning of this chapter we used an array to hold seven temperature readings (one for each day of the week). Creating an array allowed us to use loops when processing these values, rather than having to repeat the same bit of code seven times—once for each different temperature variable.

Now consider the situation where temperatures were required for the four weeks of a month. We could create four arrays as follows:

```
double[] temperature1 = new double [7]; // to hold week 1 temperatures
double[] temperature2 = new double [7]; // to hold week 2 temperatures
double[] temperature3 = new double [7]; // to hold week 3 temperatures
double[] temperature4 = new double [7]; // to hold week 4 temperatures
```

How would the temperatures for these four months be entered? The obvious solution would be to write four loops, one to process each array. Luckily there is a simpler approach—create a **multi-dimensional** array.

A multi-dimensional array is an array that has *more than one* index. So far, the arrays that we have shown you have had only one index—for this reason they are very often referred to as **one-dimensional** arrays. However, an array may have as many indices as is necessary (up to the limit of the memory on your machine). In this particular example we need *two* indices to access a temperature reading (one for the week number the other for the day number). If we required temperatures for each month of the year we may require *three* indices (one for the month number, one for the week number and one for the day number) and so on. The number of dimensions an array has corresponds to the number of indices required. Usually, no more than two indices will ever need to be used. An array with two indices is called a **two-dimensional** array.

6.9.1 Creating a Two-Dimensional Array

To create a two-dimensional (2D) array, simply provide the size of both indices. In this example we have four lots of seven temperatures:

```
double [][] temperature ; // declares a 2D array
temperature = new double [4][7]; // creates memory for a 4 by 7 array
```

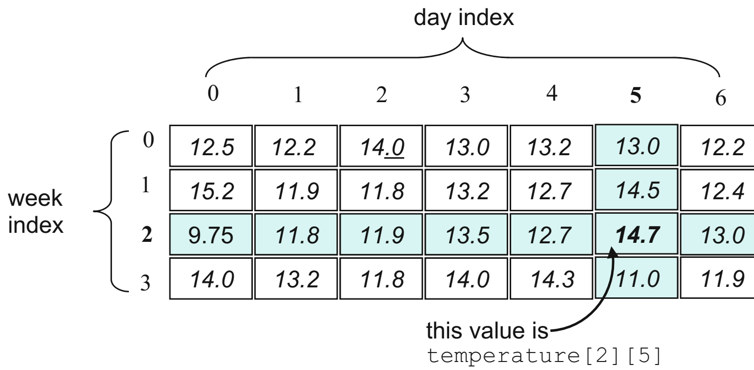


Fig. 6.5 To access an element in a 2D array requires both a row and a column index

As you can see, this is very similar to creating a one-dimensional array except that we have *two* pairs of brackets for a two-dimensional array. For larger dimensions we can have more pairs of brackets, 3 pairs for 3 dimensions, 4 for 4 dimensions and so on. In this example we have chosen to treat the first index as representing the number of weeks (4) and the second representing the number of days (7), but we could have chosen to treat the first index as the number of days and the second as the number of weeks.

While you would think of a one-dimensional array as a list, you would probably visualize a two-dimensional array as a table with rows and columns (although actually it is implemented in Java as an array of arrays). The name of each item in a two-dimensional array is the array name, plus the row and column index (see Fig. 6.5).

Note again that both indices begin at zero, so that the temperature for the *third* week of the month and the *sixth* day of the week is actually given as `temperature[2][5]`.

6.9.2 Initializing Two-Dimensional Arrays

As with one-dimensional arrays, it is possible to declare and initialize a multi-dimensional array with a collection of values all in one instruction. With a one-dimensional array we separated these values by commas and enclosed these values in braces. For example:

```
// this array of integers is initialized with four values
int[] a1DArray = { 11, -25, 4, 77};
```

This creates an array of size 4 with the given elements stored in the given order. A similar technique can be used for multi-dimensional arrays. With a two-dimensional array the sets of values in each row are surrounded with braces as above, then these row values are themselves enclosed in another pair of braces. A two-dimensional array of integers might be initialized as follows, for example:

```
// this creates a 2 dimensional array with two rows and four columns
int[][] a2DArray =        {
                                  { 11, -25, 4, 77},
                                  {-21, 55, 43, 11}
                          };
```

This instruction creates the same array as the following group of instructions:

```
int[][] a2DArray = new int[2][4]; // size array
// initialize first row of values
a2DArray[0][0] = 11;
a2DArray[0][1] = -25;
a2DArray[0][2] = 4;
a2DArray[0][3] = 77;
// initialize second row of values
a2DArray[1][0] = -21;
a2DArray[1][1] = 55;
a2DArray[1][2] = 43;
a2DArray[1][3] = 11;
```

As with one dimensional arrays, however, it is not common to initialize two-dimensional arrays in this way. Instead, once an array has been created, values are added individually to the array once the program is running.

6.9.3 Processing Two-Dimensional Arrays

With the one-dimensional arrays that we have met we have used a single `for` loop to control the value of the single array index. How would you process a two-dimensional array that requires two indices?

With a two-dimensional array, a *pair* of nested loops is commonly used to process each element—one loop for each array index. Let’s return to the two-dimensional array of temperature values. We can use a pair of nested loops, one to control the week number and the other the day number. As with the example of the one-dimensional array of `TemperatureReadings` programs in previous sections, in the following code fragment we’ve started our day and week counters at 1, and then taken 1 off these counters to get back to the appropriate array index:

```
// create Scanner object for user input
Scanner keyboard = new Scanner (System.in);

// the outer loop controls the week row
for (int week = 1; week <= temperature.length; week++)
{
   // the inner loop controls the day column
   for (int day = 1; day <= temperature[0].length; day++)
   {
      System.out.println("enter temperature for week " + week + " and day " + day);

      // as array indices start at zero not 1, we must take one off the loop counters
      temperature[week-1][day-1] = keyboard.nextDouble();
   }
}
```

Notice that in a two-dimensional array, the `length` attribute returns the length of the *first* index (this is, what we have visualized as the number of rows):

```
// here, the length attribute returns 4 (the number of rows)
for (int week = 1; week <= temperature.length; week++)
```

The number of columns is determined by obtaining the length of a particular row. In the example below we have chosen the first row but we could have chosen any row here:

```
// the length of a row returns the number of columns (7 in this case)
for (int day = 1; day <= temperature[0].length; day++)
```

Here we have used a pair of nested loops as we wish to process the entire two-dimensional array. If, however, you just wished to process part of the array (such as one row or one column) then a single loop may still suffice. In the next section we present a program that demonstrates this.

6.9.4 The *MonthlyTemperatures* Program

The program below provides the user with a menu of options. The first option allows the user to enter the temperature readings for the 4 weeks of the month. The second option allows the user to display *all* these readings. The third option allows the user to display the reading for a particular week (for example all the temperatures for week 3). The final option allows the user to display the temperatures for a particular day of the week (for example all the readings for the first day of each week). Take a look at it and then we will discuss it.

MonthlyTemperatures

```
import java.util.Scanner;

public class MonthlyTemperatures
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner (System.in);
        char choice;

        double[][] temperature = new double[4][7]; // create 2D array
        // offer menu
        do
        {
            System.out.println();
            System.out.println("[1] Enter temperatures");
            System.out.println("[2] Display all");
            System.out.println("[3] Display one week");
            System.out.println("[4] Display day of the week");
            System.out.println("[5] Exit");
            System.out.print("Enter choice [1-5]: ");
            choice = keyboard.next().charAt(0);
            System.out.println();
            // process choice by calling additional methods
            switch(choice)
            {
                case '1':  enterTemps(temperature);
                           break;
                case '2':  displayAllTemps(temperature);
                           break;
                case '3':  displayWeek(temperature);
                           break;
                case '4':  displayDays(temperature);
                           break;
                case '5':  System.out.println ("Goodbye");
                           break;
                default:   System.out.println("ERROR: options 1-5 only!");
            }
        } while (choice != '5');
    }


    // method to enter temperatures into the 2D array requires a nested loop
    static void enterTemps(double[][] temperatureIn)
    {
        Scanner keyboard = new Scanner (System.in);
        // the outer loop controls the week number
        for (int week = 1; week <= temperatureIn.length; week++)
        {
            // the inner loop controls the day number
            for (int day = 1; day <= temperatureIn[0].length; day++)
            {
                System.out.println("enter temperature for week " + week + " and day " + day);
                temperatureIn[week-1][day-1] = keyboard.nextDouble();
            }
        }
    }

        // method to display all temperatures in the 2D array requires a nested loop
        static void displayAllTemps(double[][] temperatureIn)
        {
            System.out.println();
            System.out.println("***TEMPERATURES ENTERED***");
            // the outer loop controls the week number
            for (int week = 1; week <= temperatureIn.length; week++)
            {
                // the inner loop controls the day number
                for (int day = 1; day <= temperatureIn[0].length; day++)
                {
                        System.out.println("week " +week+" day "+day+": "+ temperatureIn[week-1][day-1]);
                }
            }
        }

        // method to display temperatures for a single week requires a single loop
        static void displayWeek(double[][] temperatureIn)

        {
            Scanner keyboard = new Scanner (System.in);
            int week;
            // enter week number
        System.out.print("Enter week number (1-4): ");
        week = keyboard.nextInt();
        // input validation: week number should be between 1 and 4
        while (week<1 || week > 4)
        {
            System.out.println("Inavlid week number!!");
            System.out.print("Enter again (1-4 only): ");
            week = keyboard.nextInt();
        }
```

```
        // display temperatures for given week
        System.out.println();
        System.out.println("***TEMPERATURES ENTERED FOR WEEK "+week+"***");
        System.out.println();
        // week number is fixed so loop required to process day numbers only
        for (int day = 1; day <= temperatureIn[0].length; day++)
        {
              System.out.println("week "+week+" day "+day+": "+ temperatureIn[week-1][day-1]);
        }
    }

    // method to display temperatures for a single day of each week requires a single loop
    static void displayDays(double[][] temperatureIn)
    {
        Scanner keyboard = new Scanner (System.in);
        int day;
        // enter day number
        System.out.print("Enter day number (1-7): ");
        day = keyboard.nextInt();
        // input validation: day number should be between 1 and 7
        while (day<1 || day > 7)
        {
            System.out.println("Inavlid day number!!");
            System.out.print("Enter again (1-7 only): ");
            day = keyboard.nextInt();
        }
        // display temperatures for given day of the week
        System.out.println();
        System.out.println("***TEMPERATURES ENTERED FOR DAY "+day+"***");
        System.out.println();
        // day number is fixed so loop required to process week numbers only
        for (int week = 1; week <= temperatureIn.length; week++)
        {
              System.out.println("week "+week+" day "+day+": " + temperatureIn[week-1][day-1]);
        }
    }
}
```

Here you can see that the first menu option contains the code we have just discussed for entering values into a two-dimensional array. Notice how you pass a two-dimensional array to a method. As with a one-dimensional array you do not refer to the size of the array, just its dimensions:

```
/* As with a standard 1D array, to pass a 2D array to a method the number of dimensions needs to
be indicated but not the size of these dimensions */

static void enterTemps(double[][] temperatureIn)
{
    // code for entering temperatures goes here
}
```

This method uses a pair of nested loops as we wish to process the entire two-dimensional array. Similarly, when we wish to display the entire array we use a pair of nested loops to control the week and day number:

```
// method to display all temperatures in the 2D array requires a nested loop
static void displayAllTemps(double[][] temperatureIn)
{
    System.out.println();
    System.out.println("***TEMPERATURES ENTERED***");
    // the outer loop controls the week number
    for (int week = 1; week <= temperatureIn.length; week++)
    {
        // the inner loop controls the day number
        for (int day = 1; day <= temperatureIn[0].length; day++)
        {
                System.out.println("week " +week+" day "+day+": "+
                                                temperatureIn[week-1][day-1]);
        }
    }
}
```

However, when we need to display just one of the dimensions of an array we do not need to use a pair of loops. For example, the `displayWeek` method, that allows the user to pick a particular week number so that just the temperatures for that week alone are displayed, just requires a *single* loop to move through the day numbers, as the week number is fixed by the user:

```
// method to display temperatures for a single week requires a single loop
static void displayWeek(double[][] temperatureIn)
{
    Scanner keyboard = new Scanner (System.in);
    int week;
    // enter week number
    System.out.print("Enter week number (1-4): ");
    week = keyboard.nextInt();
    // input validation: week number should be between 1 and 4
    while (week<1 || week > 4)
    {
        System.out.println("Inavlid week number!!");
        System.out.print("Enter again (1-4 only): ");
        week = keyboard.nextInt();
    }
    // display temperatures for given week
    System.out.println();
    System.out.println("***TEMPERATURES ENTERED FOR WEEK "+week+"***");
    System.out.println();
    // week number is fixed so loop required to process day numbers only
    for (int day = 1; day <= temperatureIn[0].length; day++)
    {
            System.out.println("week "+week+" day "+day+": "+
                                        temperatureIn[week-1][day-1]);
    }
}
```

First the user enters the week number:

```
System.out.print("Enter week number (1-4): ");
week = keyboard.nextInt();
```

We will use this week number to determine the value of the first index when looking up temperatures in the array, so we need to be careful that the user inputs a valid week number (1–4) as invalid numbers would lead to an illegal array index being generated. We have used a **`while`** loop here to implement this input validation:

```
// input validation: week number should be between 1 and 4
while (week<1 || week > 4)
{
    System.out.println("Inavlid week number!!");
    System.out.print("Enter again (1-4 only): ");
    week = keyboard.nextInt();
}
```

Fig. 6.6 To access temperatures for a single week, the week index remains fixed and the day index changes

Once we get past this loop we can be sure that the user has entered a valid week number. For example, assume that we have filled the 2D array with the temperatures given in Fig. 6.5. Now assume that the user calls the option to display one week's temperature, and chooses week 2. This is illustrated in Fig. 6.6.

Since array indices in Java begin at zero, the week index (1) is determined by taking one off the week number entered by the user (2). All we need to do now is to iterate through all the day numbers for that week by using a single **for** loop:

```
// week number is fixed by the user so a single loop is required to process day numbers only

for (int day = 1; day <= temperatureIn[0].length; day++)
{
       System.out.println("week "+ week +" day " + day + ": " +
                                    temperatureIn[week-1][day-1]);
}
```

The `displayDays` method works in a similar way but with the day number fixed and the week number being determined by the **for** loop.

6.10 Ragged Arrays

In the examples of two-dimensional arrays discussed so far, each row of the array had the same number of columns. Each row of the two-dimensional `temperature` array, for example, had 7 columns and each row of `a2DArray` had 4 columns. Very often this will be the case. However, very occasionally, it may be necessary for rows to have a variable number of columns. A two-dimensional array

0	‘M’	‘O’	‘N’	‘K’	‘E’	‘Y’
1	‘C’	‘A’	‘T’			
2	‘B’	‘I’	‘R’	‘D’		
	0	**1**	**2**	**3**	**4**	**5**

Fig. 6.7 The array ‘`animals`’ is a *ragged array*

with a variable number of columns is called a **ragged array**. For example, here is how we might declare and initialize a two-dimensional array of characters with a variable number of columns for each row:

```
// this creates a 2 dimensional array with a variable number of columns
char[][] animals =        {
                    {‘M’, ‘O’, ‘N’, ‘K’, ‘E’, ‘Y’}, // 6 columns
                    {‘C’, ‘A’, ‘T’ }, // 3 columns
                    {‘B’, ‘I’, ‘R’, ‘D’} // 4 columns
                  };
```

Figure 6.7 illustrates the array created after this initialization.

To declare such an array without initialization, we need to specify the number of rows first and leave the number of columns unspecified. In the example above we have 3 rows:

```
// columns left unspecified
char[][] animals = new char[3][];
```

Now, for each row we can fix the appropriate size of the associated column. In the example above the first row has 6 columns, the second row 3 columns and the last row 4 columns:

```
animals[0] = new char[6]; // number of columns in first row
animals[1] = new char[3]; // number of columns in second row
animals[2] = new char[4]; // number of columns in third row
```

You can see clearly from these instructions that Java implements a two-dimensional array as an array of arrays. When processing ragged arrays you must be careful not to use a fixed number to control the number of columns. The actual number of columns can always be retrieved by calling the `length` attribute

of each row. For example the following instructions would display the number of columns for each row:

```
System.out.println(animals[0].length); // displays 6
System.out.println(animals[1].length); // displays 3
System.out.println(animals[2].length); // displays 4
```

The program below uses a pair of nested loops to display the `animals` array:

RaggedArray

```
public class RaggedArray
{
    public static void main(String[] args)
    {
         // initialize ragged array
         char[][] animals =  {
                                          {'M', 'O', 'N', 'K', 'E', 'Y'}, // 6 columns
                                          {'C', 'A', 'T' }, // 3 columns
                                          {'B', 'I', 'R', 'D'} // 4 columns
                              };

         for (int row = 0; row < animals.length; row++) //row number is fixed
         {
                 for (int col = 0; col < animals[row].length; col++) // column number is variable
                 {
                         System.out.print(animals[row][col]); // display one character
                 }
                 System.out.println(); // new line after one row displayed
         }
    }
}
```

Notice how the inner loop, controlling the column number, instead of being fixed to the length of one particular row, will vary each time depending upon the *current* row number:

```
for (int row = 0; row < animals.length; row++)
{
    // column number is variable and is determined by current row number
    for (int col = 0; col < animals[row].length; col++)
    {
        System.out.print(animals[row][col]);
    }
    System.out.println();
}
```

As expected this program produces the following output when run:

```
MONKEY
CAT
BIRD
```

6.11 Self-test Questions

1. When is it appropriate to use an *array* in a program?

2. Consider the following explicit creation of an array:

```
int[] someArray = {2,5,1,9,11};
```

   (a) What would be the value of `someArray.length`?
   (b) What is the value of `someArray[2]`?
   (c) What would happen if you tried to access `someArray[6]`?
   (d) Create the equivalent array by using the **new** operator and then assigning the value of each element individually.
   (e) Write a standard **for** loop that will double the value of every item in `someArray`.
   (f) Explain why, in this example, it would not be appropriate to use an enhanced **for** loop.
   (g) Use an enhanced **for** loop to display the values inside the array.
   (h) Modify the enhanced **for** loop above so that only numbers greater than 2 are displayed.

3. Look back at the program `TemperatureReadings3` from Sect. 6.6, which read in and displayed a series of temperature readings. Design and write the code for an additional method, `wasHot`, which displays all days that recorded temperatures of 18° or over.

4. Assume that an array has been declared in `main` as follows:

```
int[] javaStudents;
```

   This array is to be used to store a list of student exam marks. Now, for each of the following methods, write the code for the given method and the instruction in `main` to call this method:

   (a) A method, `enterExamMarks`, that prompts the user to enter some exam marks (as integers), stores the marks in an array and then returns this array.
   (b) A method, `increaseMarks`, that accepts an array of exam marks and increases each mark by 5.
   (c) A method, `allHavePassed`, that accepts an array of exam marks and returns **true** if all marks are greater than or equal to 40, and **false** otherwise.

5. (a) Describe the **varargs** feature of Java.
   (b) Re-write the `contains` method below, from Sect. 6.8.5, to make use of this **varargs** feature:

```
static boolean contains(int[] arrayIn, int valueIn)
{
   // enhanced 'for' loop used here
   for (int currentElement : arrayIn)
   {
   if (currentElement == valueIn)
      {
         return true; // exit loop early if value found
      }
   }
   return false; // value not present
}
```

   (c) Give examples of different ways in which you could call this re-written `contains` method now that you have used **varargs**.

6. Consider the following array declaration, to store a collection of student grades.

```
char [] [] grades = new char [4] [20];
```

   Grades are recorded for 4 tutorial groups, and each tutorial group consists of 20 students.

   (a) How many dimensions does this array have?
   (b) What is the value of `grades.length`?
   (c) What is the value of `grades[0].length`?
   (d) Write the instruction to record the grade '`B`' for the first student in the first tutorial group.

7. Consider the following scenarios and, for each, declare the appropriate array:

   (a) `goals`: an array to hold the number of goals each team in a league scores in each game of a season. The league consist of 20 teams and a season consists of 38 games.
   (b) `seats`: an array to record whether or not a seat in a theatre is booked or not. There are 70 rows of seats in the theatre and each row has 20 seats.

8. Consider the `MonthlyTemperatures` program of Sect. 6.9.4. Write an additional method, `max`, that returns the maximum temperature recorded in the given two-dimensional array.

   Hint: look back at the algorithm for finding the maximum value in a one-dimensional array in Sect. 6.8.1

9. Consider an application that records the punctuality of trains on a certain route.

   (a) Declare a 2D array, `late`, to hold the number of times a train on this route was late for each day of the week, and for each week of the year.

   (b) Write a fragment of code that adds up the total number of days in the year when a train was late more than twice in a given day.

10. A **magic word square** is a square in which a word can be formed by reading each row and each column. For example, the following is a 4 by 4 magic word square:

‘P’	‘R’	‘E’	‘Y’
‘L’	‘A’	‘V’	‘A’
‘O’	‘V’	‘E’	‘R’
‘T’	‘E’	‘N’	‘D’

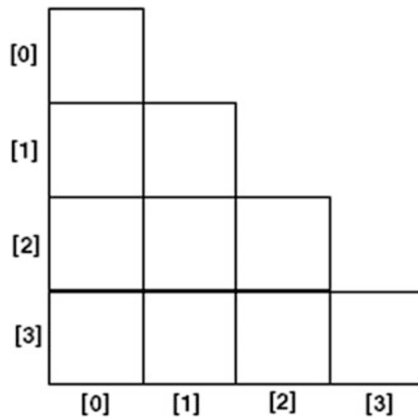
   (a) Declare and initialize a 2D array, `magicSquare`, to hold the words illustrated above.

   (b) Write a method, `displayRow`, that accepts the `magicSquare` array and a row number and displays the word in that row.

   (c) Write a method, `displayColumn`, that accepts the `magicSquare` array and a column number and displays the word in that column.

11. (a) Distinguish between a regular 2D array and a ragged array.

   (b) Write instructions to create a ragged 2D array of integers, called `triangle`, that has the following form:



(c) Write a fragment of code to find the largest number in the `triangle` array.

6.12 Programming Exercises

1. Implement the program `TemperatureReadings3` from Sect. 6.6, which read in and displayed a series of temperature readings for a given week. Now

   (a) implement the `wasHot` method that you designed in self-test question 3 above;

   (b) add a final instruction into the `main` method that calls this `wasHot` method.

2. Implement the program `SomeUsefulArrayMethods` from Sect. 6.8.5, which manipulates an array of integers, then

   (a) re-write the `contains` method using **varargs** syntax as discussed in self test question 5 above;

   (b) add an additional method to return the average from the array of integers (make use of the `sum` method to help you calculate the average);

   (c) add one more method to display on the screen all those values greater than or equal to the average.

3. Implement a program for entering and displaying student scores that tests your answers to self-test question 4 above.

4. Implement the `MonthlyTemperatures` program from Sect. 6.9.4, which read in and displayed temperature readings for four weeks of a month. Now

   (a) implement the `max` method that you designed in self-test question 8 above;
   (b) add a final instruction into the `main` method that calls this `max` method.

5. Design and implement a magic word square program that allows you to test your answers to self-test question 10 above. The program should initialize the word square given in the question and then use the methods `displayRow` and `displayColumn` to display all the words in the magic word square.

6. Design and implement a program that allows the user to enter into an array the price of 5 products in pounds sterling. The program should then copy this array into another array but convert the price of each product from pounds sterling to US dollars. The program should allow the user to enter the exchange rate of pounds to dollars, and should, when it terminates, display the contents of both arrays. Once again, make use of methods in your program to carry out these tasks.

7. Amend the program in Exercise 6 above so that

   (a) the user is asked how many items they wish to purchase and the arrays are then sized accordingly;
   (b) the total cost of the order is displayed in both currencies.

8. Design and implement a program that allows you to test your answers to self-test question 11 above. The program should allow the user to enter numbers into the ragged `triangle` array and then find the largest number in the array as discussed in the question.

Check for updates

7 Classes and Objects

Outcomes:

By the end of this chapter you should be able to:

- *explain the meaning of the term* ***object-oriented***;
- *explain and distinguish between the terms* ***class*** *and* ***object***;
- *create objects in Java*;
- *call the methods of an object*;
- *use a number of methods of the String class*;
- *create and use arrays of objects*;
- *create an* `ArrayList` *and make use of the* `add` *and* `get` *methods of this class*.

7.1 Introduction

In the 1990s it became the norm for programming languages to use special constructs called **classes** and **objects**. Such languages are referred to as **object-oriented programming languages**. In this chapter and the next one we will explain what is meant by these terms, and show you how we can exploit the full power of object-oriented languages like Java.

7.2 Classes as Data Types

So far you have been using data types such as **`char`**, **`int`** and **`double`**. These are simple data types that hold a *single* piece of information. But what if we want a variable to hold more than one related piece of information? Think for example of a book—a book might have a title, an author, an ISBN number and a price—or a

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_7

student might have a name, an enrolment number and marks for various subjects. Types such as **char** and **int** can hold a single piece of information only, and would therefore be completely inadequate for holding all the necessary information about a book or a student. An array would also not do because the different bits of data will not necessarily be all of the same type. Earlier languages such as C and Pascal got around this problem by allowing us to create a type that allowed more than one piece of information to be held—such types were known by various names in different languages, the most common being *structure* and *record*.

Object-oriented languages such as Java and C++ went one stage further however. They enabled us not only to create types that stored many pieces of data, but also to define within these types the methods by which we could process that data. For example a book 'type' might have a method that adds tax to the sale price; a student 'type' might have a method to calculate an average mark.

Do you remember that in the exercises at the end of Chap. 2 you wrote a little program that asked the user to provide the length and height of a rectangle, and then displayed the area and perimeter of that rectangle? In Chap. 5 you were asked to adapt this program so that it made use of separate methods to perform the calculations. Such a program might look like this:

RectangleCalculations

```
import java.util.Scanner;

public class RectangleCalculations
{
       public static void main(String[] args)
       {
          double length, height, area, perimeter;
          Scanner keyboard = new Scanner(System.in);
          System.out.print("What is the length of the rectangle? "); // prompt for length
          length = keyboard.nextDouble(); // get length from user
          System.out.print("What is the height of the rectangle? "); // prompt for height
          height = keyboard.nextDouble(); // get height from user
          area = calculateArea(length, height); // call calculateArea method
          perimeter = calculatePerimeter(length, height); // call calculatePerimeter method
          System.out.println("The area of the rectangle is " + area); // display area
          System.out.println("The perimeter of the rectangle is " + perimeter); // display perimeter
       }

       // method to calculate area
       static double calculateArea(double lengthIn, double heightIn)
       {
          return lengthIn * heightIn;
       }

       // method to calculate perimeter
       static double calculatePerimeter(double lengthIn, double heightIn)
       {
          return 2 * (lengthIn + heightIn);
       }
}
```

Can you see how useful it might be if, each time we wrote a program dealing with rectangles, instead of having to declare several variables and write methods to calculate the area and perimeter of a rectangle, we could just use a rectangle 'type' to create a single variable, and then use its pre-written methods? In fact you wouldn't even have to know how these calculations were performed.

This is exactly what an object-oriented language like Java allows us to do. You have probably guessed by now that this special construct that holds both data and methods is called a **class**. You have already seen a class as the basic unit which

contains our `main` method and any other additional methods. Now we can also use classes to define new 'types'.

You can see that there are two aspects to a class:

- the data that it holds;
- the tasks it can perform.

In the next chapter you will see that the different items of data that a class holds are referred to as the **attributes** of the class; the tasks it can perform, as we have seen, are referred to as the **methods** of the class—you have seen in Chap. 5 how we define methods. However, in Chap. 5, the methods were called only from *within* the class itself. Now we are going to see how to call the methods of *another* class. In fact you have already been doing this without quite realizing it—because you have, since the second chapter, been calling the methods of the `Scanner` class!

7.3 Objects

In order to use the methods of a class you need to create an **object** of that class. To understand the difference between classes and objects, you can think of a class as a blueprint, or template, from which objects are generated, whereas an object refers to an individual *instance* of that class. For example, imagine a system that is used by a bookshop. The shop will contain many hundreds of books—but we do not need to define a book hundreds of times. We would define a book once (in a class) and then generate as many objects as we want from this blueprint, each one representing an individual book.

This is illustrated in Fig. 7.1.

In one program we may have many classes, as we would probably wish to generate many kinds of objects. A bookshop system might have books, customers, suppliers and so on. A university administration system might have students, courses, lecturers etc.

Object-oriented programming therefore consists of defining one or more classes that may interact with each other.

We will now illustrate all of this by creating and using objects of predefined classes—defined either by ourselves or defined by the Java developers and provided as a standard part of the Java Development Kit. We are going to start by considering the example we discussed in Sect. 7.2 where we proposed a single "type" that dealt with calculating the area and perimeter of a rectangle. We are in fact not going to call our new class `Rectangle`, but `Oblong`. The reason for this is that there are in fact more than one `Rectangle` classes defined in the "built-in" Java libraries; now, while it is perfectly possible to have more than one class with the same name, it is better to avoid any possible confusion at this stage. In Chap. 19, you will learn how to avoid naming conflicts, but for now it is better to call our new class by a unique name.

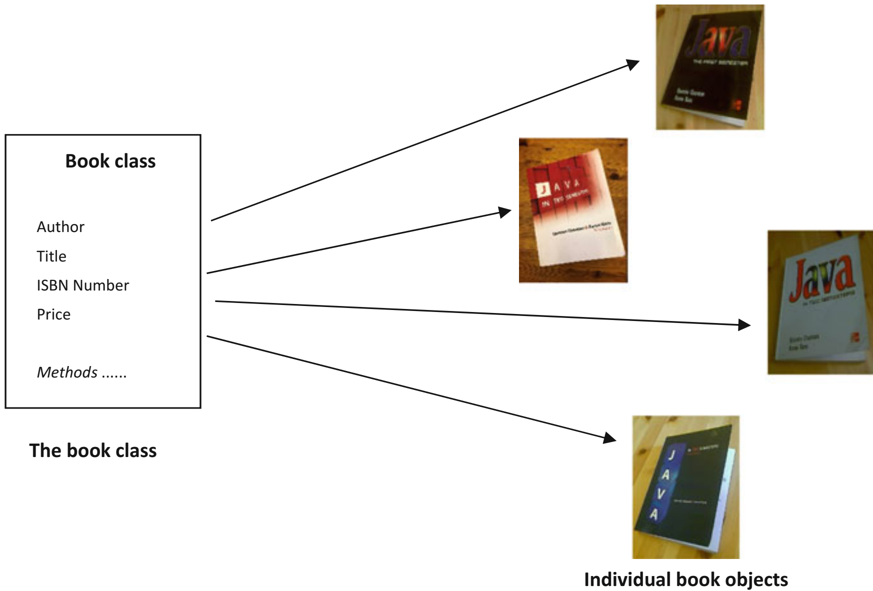


Fig. 7.1 Many objects can be generated from a single class template

And here is a "fun fact" for you—an oblong is not actually the same as a rectangle, because in an oblong the length and the height must be unequal, whereas a rectangle can have four equal sides—in other words it can be a square. Put another way a square is a kind of rectangle but not a kind of oblong.

7.4 The *Oblong* Class

We have written an `Oblong` class for you. The class we have created is saved in a file called `Oblong.java`. and you will need to download it from the website in order to use it. You must make sure that it is in the right place for your compiler to find it. You will need to place it in your project according to the rules of the particular IDE you are using.

In the next chapter we will look inside this class and see how it was written. For now, however, you can completely ignore the program code, because you can use a class without knowing anything about the details.

Once you have been provided with this `Oblong` class, instead of being restricted to making simple declarations like this:

```
int x;
```

you will now be able to make declarations like:

```
Oblong myOblong;
```

You can see that this line is similar to a declaration of a variable; however what we are doing here is not declaring a variable of a primitive type such as **`int`**, but declaring the name of an *object* (`myOblong`) of the *class* (`Oblong`)—effectively we have created a new type, `Oblong`.

You need to be sure that you understand what this line actually does; all it does in fact is to create a variable that holds a **reference** to an object, rather than the object itself. As explained in the previous chapter, a reference is simply a *name* for a location in memory. At this stage we have *not* reserved space for our new `Oblong` object; all we have done is named a memory location `myOblong`, as shown in Fig. 7.2.

Now of course you will be asking the question "How is memory for the `Oblong` object going to be created, and how is it going to be linked to the reference `myOblong`?".

As we have indicated, an object is often referred to as an *instance* of a class; the process of creating an object is referred to as **instantiation**. In order to create an object we use a very special method of the class called a **constructor**.

The constructor is a method that *always has the same name as the class*. When you create a new object this special method is always called; its function is to reserve some space in the computer's memory just big enough to hold the required object (in our case an object of the `Oblong` class).

As we shall see in the next chapter, a constructor can be defined to do other things, as well as reserve memory. In the case of our `Oblong` class, the constructor has been defined so that every time a new `Oblong` object is created, the length and the height are set—and they are set to the values that the user of the class sends in. So every time you create an `Oblong` object you have to specify its length and its height at the same time. Here for example is how you would call the constructor and create an oblong with length 7.5 and height 12.5:

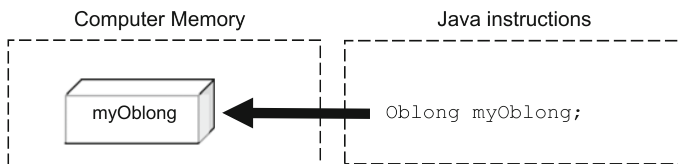


Fig. 7.2 Declaring an object reference

```
myOblong = new Oblong(7.5, 12.5);
```

This is the statement that reserves space in memory for a new Oblong. Using the constructor with the keyword **new**, reserves memory for a new Oblong object. Now, in the case of the Oblong class, the people who developed it (in this case it was us!) defined the constructor so that it requires that two items of data, both of type **double**, get sent in as parameters. Here we have sent in the numbers 7.5 and 12.5. The location of the new object is stored in the named location myOblong. This is illustrated in Fig. 7.3.

Now every time we want to refer to our new Oblong object we can use the variable name myOblong.

As is the case with the declaration and initialization of simple types, Java allows us to declare a reference and create a new object all in one line:

```
Oblong myOblong = new Oblong(7.5, 12.5);
```

There are all sorts of ways that we can define constructors (for example, in a BankAccount class we might want to set the overdraft limit to a particular value when the account is created) and we shall see examples of these as we go along. You have of course already seen another example of this, namely with the Scanner class:

```
Scanner keyboard = new Scanner(System.in);
```

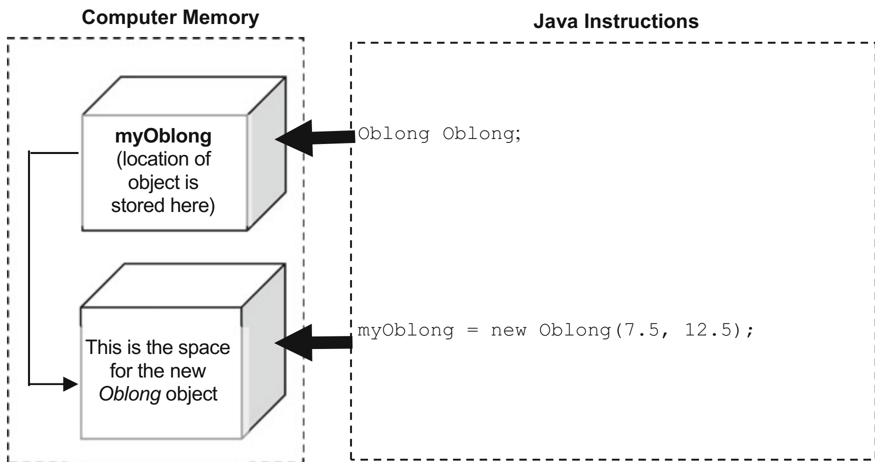


Fig. 7.3 Creating a new object

You can understand now how this line creates a new `Scanner` object, `keyboard`, by calling the constructor. The parameter that we are sending in, `System.in`, represents a keyboard object and by using this parameter we are associating the new `Scanner` object with the keyboard.

You will see in the next chapter that when a class is written we make sure that no program can assign values to the attributes directly. In this way the data in the class is protected. Protecting data in this way is known as **encapsulation**. The *only way* to interact with the data is via the methods of the class.

This means that in order to use a class all we need to know are details about its methods: their names, what the methods are expected to do, and also their **inputs** and **outputs**.[1] In other words you need to know what parameters the method requires, and its return type. Once we know this we can interact with the class by using its methods—and it is important to understand that the *only* way we can interact with a class is via its methods.

Table 7.1 lists all the methods of our `Oblong` class with their inputs and outputs—including the constructor.

As far as our `Oblong` class is concerned we have, as expected, provided two methods which will return values for the area and perimeter of the oblong respectively. However, the class wouldn't be very useful if we did not have some means of giving values to the length and height of the oblong. As you have seen we do this initially via the constructor, but we might also want to be able to change these values during the course of a program. We have therefore provided methods called `setLength` and `setHeight` so that we can *write* values to the attributes. It is very likely that we will want to display these values—we have therefore provided methods to return, or *read*, the values of the attributes. These we have called `getLength` and `getHeight`.

You have used methods of the `Scanner` class on many occasions, for example:

```
x = keyboard.nextInt();
```

You can see that in order to call a method of one class from another class we use the name of the object (in this case `keyboard`) together with the name of the method (`nextInt`) separated by a full stop (often referred to as the **dot operator**).

In the case of the `Oblong` class, we might, for example, call the `setLength` method with a statement such as:

```
myOblong.setLength(5.0);
```

[1]A list of a method's inputs and outputs is often referred to as the method's **interface**—though this should not be confused with the *user interface*, the meaning of which we described in the first chapter.

Table 7.1 The methods of the *Oblong* class

Method	Description	Inputs	Output
`Oblong`	The constructor	Two items of data, both of type **`double`**, representing the length and height of the oblong respectively	Not applicable
`setLength`	Sets the value of the length of the oblong	An item of type **`double`**	None
`setHeight`	Sets the value of the height of the oblong	An item of type **`double`**	None
`getLength`	Returns the length of the oblong	None	An item of type **`double`**
`getHeight`	Returns the height of the oblong	None	An item of type **`double`**
`calculateArea`	Calculates and returns the area of the oblong	None	An item of type **`double`**
`calculatePerimeter`	Calculates and returns the perimeter of the oblong	None	An item of type **`double`**

In Chap. 5, when we called the methods from *within* a class, we used the name of the method on its own. In actual fact, what we were doing is form of shorthand. When we write a line such as

```
demoMethod(x);
```

we are actually saying call `demoMethod`, which is a method of *this* class. In Java there exists a special keyword **`this`**. The keyword **`this`** is used within a class when we wish to refer to an attribute of the class itself, rather than an attribute of some other class. The line of code above is actually shorthand for:

```
this.demoMethod(x);
```

As your programming skills advance, you will find that there are occasions when you actually have to use the **`this`** keyword, rather than simply allowing it to be assumed.

You should be aware of the fact that, just as you cannot use a variable that has not been initialized, you cannot call a method of an object if no storage is allocated for the object; so watch out for this happening in your programs—it would cause a problem at run-time. In Java, when a reference is first created without assigning it to a new object in the same line, it is given a special value of **null**; a **null** value indicates that no storage is allocated. We can also *assign* a **null** value to a reference at any point in the program, and test for it as in the following example:

```
Oblong myOblong; // at this point myOblong has a null value
myOblong = new Oblong(5.0, 7.5); // create a new Oblong object with length 5.0 and height 7.5

// more code goes here

myOblong = null; // re-assign a null value
if(myOblong == null) // test for null value
{
    System.out.println("No storage is allocated to this object");
}
```

In the next section we will write a program that creates an `Oblong` object and then uses the methods described in Table 7.1 to interact with this object.

7.5 The *OblongTester* Program

The following program shows how the `Oblong` class can be used by another class, in this case a class called `OblongTester`. Study the program code and then we will discuss it.

OblongTester

```
import java.util.Scanner;

public class OblongTester
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        // declare two variables to hold the length and height of the Oblong as input by the user
        double oblongLength, oblongHeight;

        // declare a reference to an Oblong object
        Oblong myOblong;

        // now get the values from the user
        System.out.print("Please enter the length of your Oblong: ");
        oblongLength = keyboard.nextDouble();
        System.out.print("Please enter the height of your Oblong: ");
        oblongHeight = keyboard.nextDouble();

        // create a new Oblong object
        myOblong = new Oblong(oblongLength, oblongHeight);

        /* use the various methods of the Oblong class to display the length, height, area and
           perimeter of the Oblong */
        System.out.println("Oblong length is " + myOblong.getLength());
        System.out.println("Oblong height is " + myOblong.getHeight());
        System.out.println("Oblong area is " + myOblong.calculateArea());
        System.out.println("Oblong perimeter is " + myOblong.calculatePerimeter());
    }
}
```

Let's analyse the `main` method line by line. After creating the new `Scanner` object, the method goes on to declare two variables:

```
double oblongLength, oblongHeight;
```

As you can see, these are of type **`double`** and they are going to be used to hold the values that the user chooses for the length and height of the oblong.

The next line declares the `Oblong` object:

```
Oblong myOblong;
```

After getting the user to enter values for the length and height of the oblong we have this line of code:

```
myOblong = new Oblong(oblongLength, oblongHeight);
```

Here we have called the constructor and sent through the length and height as entered by the user.

Now the next line:

```
System.out.println("Oblong Length is " + myOblong.getLength());
```

This line displays the length of the oblong. It uses the method of `Oblong` called `getLength`, and as we said in the previous section to do this we use the dot operator to separate the name of the object and the name of the method.

The next three lines are similar:

```
System.out.println("Oblong height is " + myOblong.getHeight());
System.out.println("Oblong area is " + myOblong.calculateArea());
System.out.println("Oblong Perimeter is "+ myOblong.calculatePerimeter());
```

We have called the `getHeight` method, the `calculateArea` method and the `calculatePerimeter` method to display the height, area and perimeter of the oblong on the screen.

You might have noticed that we haven't used the `setLength` and `setHeight` methods—that is because in this program we didn't wish to change the length and height once the oblong had been created—but this is not the last you will see of our `Oblong` class—and in future programs these methods will come in useful.

Now we can move on to look at using some other classes. The first is not one of our own, but the built-in `String` class provided with all versions of Java.

7.6 Strings

You know from Chap. 1 that a string is a sequence of characters—like a name, a line of an address, a car registration number, or indeed any meaningless sequence of characters such as "h83hdu2&e£8". Java provides a String class that allows us to use and manipulate strings.

As we shall see in a moment, the String class has a number of constructors—but in fact Java actually allows us to declare a string object in the same way as we declare variables of simple type such as **int** or **char**. You should remember of course that String is a class, and starts with a capital letter. For example we could make the following declaration:

```
String name;
```

and we could then give this string a value:

```
name = "Quentin";
```

We could also do this in one line:

```
String name = "Quentin";
```

We should bear in mind, however, that this is actually just a convenient way of declaring a String object by calling its constructor, which we would do like this with exactly the same effect:

```
String name = new String("Quentin");
```

You should be aware that the String class is the only class that allows us to create new objects by using the assignment operator in this way.

7.6.1 Obtaining Strings from the Keyboard

In order to get a string from the keyboard, you should use the next method of Scanner. However, a word of warning here—when you do this you should not enter strings that include spaces, as this will give you unexpected results. We will show you in the next section a way to get round this restriction.

Below is a little program that uses the Java String class. Some of you might find it amusing (although others might not!).

StringTest

```
import java.util.Scanner;

public class StringTest
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        String name; // declaration of a String
        int age;
        System.out.print("What is your name? ");
        name = keyboard.next(); // the 'next' method is for String input
        System.out.print("What is your age? ");
        age = keyboard.nextInt();
        System.out.println();
        System.out.println("Hello " + name);
        // now comes the joke!!
        System.out.println("When I was your age I was "  + (age + 1));
    }
}
```

One thing to notice in this program is the way in which the + operator is used for two very different purposes. It is used with strings for concatenation—for example:

```
"Hello" + name
```

It is also used with numbers for addition—for example:

```
age + 1
```

Notice that we have had to enclose this expression in brackets to avoid any confusion:

```
System.out.println("When I was your age I was "  + (age + 1));
```

Here is a sample run from the above program:

```
What is your name? Aaron
What is your age? 15

Hello Aaron
When I was your age I was 16
```

7.6.2 The Methods of the *String* Class

The `String` class has a number of interesting and useful methods, and we have listed some of them in Table 7.2.

Table 7.2 Some *String* methods

Method	Description	Inputs	Output
`length`	Returns the length of the string	None	An item of type **`int`**
`charAt`	Accepts an integer and returns the character at that position in the string. Note that indexing starts from zero, not 1! You have been using this method in conjunction with the `next` method of the `Scanner` class to obtain single characters from the keyboard	An item of type **`int`**	An item of type **`char`**
`substring`	Accepts two integers (for example `m` and `n`) and returns a copy of a chunk of the string. The chunk starts at position `m` and finishes at position `n-1`. Remember that indexing starts from zero. (Study the example below.)	Two items of type **`int`**	A `String` object
`concat`	Accepts a string and returns a new string which consists of the string that was sent in joined on to the end of the original string	A `String` object	A `String` object
`toUpperCase`	Returns a copy of the original string, all upper case	None	A `String` object
`toLowerCase`	Returns a copy of the original string, all lower case	None	A `String` object
`compareTo`	Accepts a string (say `myString`) and compares it to the object's string. It returns zero if the strings are identical, a negative number if the object's string comes before `myString` in the alphabet, and a positive number if it comes later	A `String` object	An item of type **`int`**
`equals`	Accepts an object (such as a `String`) and compares this to another object (such as another `String`). It returns **`true`** if these are identical, otherwise returns **`false`**	An object of any class	A **`boolean`** value
`equalsIgnoreCase`	Accepts a string and compares this to the original string. It returns **`true`** if the strings are identical (ignoring case), otherwise returns **`false`**	A `String` object	A **`boolean`** value
`startsWith`	Accepts a string (say `str`) and returns **`true`** if the original string starts with `str` and **`false`** if it does not (e.g. "hello world" starts with "h" or "he" or "hel" and so on)	A `String` object	A **`boolean`** value
`endsWith`	Accepts a string (say `str`) and returns **`true`** if the original string ends with `str` and **`false`** if it does not (e.g. "hello world" ends with "d" or "ld" or "rld" and so on)	A `String` object	A **`boolean`** value
`trim`	Returns a `String` object, having removed any spaces at the beginning or end	None	A `String` object

There are many other useful methods of the `String` class which you can look up. The following program provides examples of how you can use some of the methods listed above; others are left for you to experiment with in your practical sessions.

StringMethods

```
import java.util.Scanner;

public class StringMethods
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        // create a new string
        String str;
        // get the user to enter a string
        System.out.print("Enter a string without spaces: ");
        str = keyboard.next();
        // display the length of the user's string
        System.out.println("The length of the string is " + str.length());
        // display the third character of the user's string
        System.out.println("The character at position 3 is " + str.charAt(2));
        // display a selected part of the user's string
        System.out.println("Characters 2 to 4 are " + str.substring(1,4));
        // display the user's string joined with another string
        System.out.println(str.concat(" was the string entered"));
        // display the user's string in upper case
        System.out.println("This is upper case: " + str.toUpperCase());
        // display the user's string in lower case
        System.out.println("This is lower case: " + str.toLowerCase());
    }
}
```

A sample run:

```
Enter a string without spaces: Europe
The length of the string is 6
The character at position 3 is r
Characters 2 to 4 are uro
Europe was the string entered
This is upper case: EUROPE
This is lower case: europe
```

7.6.3 Comparing Strings

When comparing two objects, such as `Strings`, we should do so by using a method called `equals`. We should *not* use the equality operator (==); this should be used for comparing primitive types only. If, for example, we had declared two strings, `firstString` and `secondString`, we would compare these in, say, an **`if`** statement as follows:

```
if(firstString.equals(secondString))
{
    // more code here
}
```

Using the equality operator (==) to compare strings is a very common mistake that is made by programmers. Doing this will not result in a compilation error, but it won't give you the result you expect! The reason for this is that all you are doing is finding out whether the objects occupy the same address space in memory—what you actually want to be doing is comparing the actual value of the string attributes of the objects.

Notice that an object of type `String` can also be used within a **`switch`** statement to check to see if it is equal to one of several possible `String` values. The simple `StringCheckWithSwitch` program below illustrates this by giving a meaning for three symbols on a game controller for a particular game:

StringCheckWithSwitch

```
import java.util.Scanner;

public class StringCheckWithSwitch
{
    public static void main(String[] args)
    {
       Scanner keyboard = new Scanner (System.in);
       String symbol;
       // get symbol from user
       System.out.println("Enter the symbol(square/circle/triangle)");
       symbol = keyboard.next();
       // use String object in switch
       switch(symbol)
       {
            case "square": System.out.println("ATTACK"); break;
            case "circle": System.out.println("BLOCK"); break;
            case "triangle": System.out.println("JUMP"); break;
            default: System.out.println("Invalid Choice");
       }
    }
}
```

Here is a sample run from the program:

```
Enter the symbol(square/circle/triangle)
triangle
JUMP
```

Here is another sample run from the program:

```
Enter the symbol(square/circle/triangle)
square
ATTACK
```

The `String` class also has a very useful method called `compareTo`. As you can see from Table 7.2 this method accepts a string (called `myString` for example) and compares it to the string value of the object itself. It returns zero if the strings are identical, a negative number if the original string comes before `myString` in the alphabet, and a positive number if it comes later.

The program below provides an example of how the `compareTo` method is used.

StringComparison

```
import java.util.Scanner;

public class StringComparison
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        String string1, string2;
        int comparison;

        // get two strings from the user
        System.out.print("Enter a String: ");
        string1 = keyboard.next();
        System.out.print("Enter another String: ");
        string2 = keyboard.next();

        // compare the strings
        comparison = string1.compareTo(string2);
        if(comparison < 0) // compareTo returned a negative number
        {
            System.out.println(string1 + " comes before " + string2 + " in the alphabet");
        }
        else if(comparison > 0) // compareTo returned a positive number
        {
            System.out.println(string2 + " comes before " + string1 + " in the alphabet");
        }
        else // compareTo returned zero
        {
            System.out.println("The strings are identical");
        }
    }
}
```

Here is a sample run from the program:

```
Enter a String: hello
Enter another String: goodbye
goodbye comes before hello in the alphabet
```

You should note that (as with the `equals` method) the `compareTo` method is case-sensitive—upper-case letters will be considered as coming before lower-case letters (their Unicode value is lower). If you are not interested in the case of the letters, you should convert both strings to upper (or lower) case before comparing them.

If all you are interested in is whether the strings are identical, it is easier to use the `equals` method. If the case of the letters is not significant you can use `equalsIgnoreCase`.

7.6.4 Entering Strings Containing Spaces

As we mentioned above there is a problem with using the `next` method of `Scanner` when we enter strings that contain spaces. If you try this you will see that the resulting string stops at the first space, so if you enter the string “Hello world” for example, the resulting string would actually be “Hello”.

To enter a string that contains spaces you need to use the method `nextLine`. Unfortunately however there is also an issue with this. If the `nextLine` method is used after a `nextInt` or `nextDouble` method, then it is necessary to create a separate `Scanner` object (because using the same `Scanner` object will make

your program behave erratically). So, if your intention is that the user should be able to enter strings that contain spaces, the best thing to do is to declare a separate `Scanner` object for string input. This is illustrated below:

StringExample2

```
import java.util.Scanner;

public class StringExample2
{
    public static void main(String[] args)
    {
        double d;
        int i;
        String s;
        Scanner keyboardString = new Scanner(System.in); // Scanner object for string input
        Scanner keyboard = new Scanner(System.in); // Scanner object for all other types of input
        System.out.print("Enter a double: ");
        d = keyboard.nextDouble();
        System.out.print("Enter an integer: ");
        i = keyboard.nextInt();
        System.out.print("Enter a string: ");
        s = keyboardString.nextLine(); // use the Scanner object reserved for string input
        System.out.println();
        System.out.println("You entered: ");
        System.out.println("Double: " + d);
        System.out.println("Integer: " + i);
        System.out.println("String: " + s);
    }
}
```

Here is a sample run from this program:

```
Enter a double: 3.4
Enter an integer: 10
Enter a string: Hello world

You entered:
Double: 3.4
Integer: 10
String: Hello world
```

7.7 Our Own *Scanner* Class for Keyboard Input

It might have occurred to you that using the `Scanner` class to obtain keyboard input can be a bit of a bother.

- it is necessary to create a new `Scanner` object in every method that uses the `Scanner` class;
- there is no simple method such as `nextChar` for getting a single character like there is for the **`int`** and **`double`** types;
- as we have just seen there is an issue when it comes to entering strings containing spaces.

Table 7.3 The input methods of the *EasyScanner* class

Java type	EasyScanner method
int	nextInt()
double	nextDouble()
char	nextChar()
String	nextString()

To make life easier, we have created a new class which we have called EasyScanner. In the next chapter we will "look inside" it to see how it is written—in this chapter we will just show you how to use it. The methods of EasyScanner are described in Table 7.3.

To make life really easy we have written the class so that we don't have to create new Scanner objects in order to use it (that is taken care of in the class itself)—and we have written it so that you can simply use the name of the class itself when you call a method (you will see how to do this in the next chapter). The following program demonstrates how to use these methods.

EasyScannerTester

```
public class EasyScannerTester
{
    public static void main(String[] args)
    {
        System.out.print("Enter a double: ");
        double d = EasyScanner.nextDouble(); // to read a double
        System.out.println("You entered: " + d);
        System.out.println();

        System.out.print("Enter an integer: ");
        int i = EasyScanner.nextInt(); // to read an int
        System.out.println("You entered: " + i);
        System.out.println();

        System.out.print("Enter a string: ");
        String s = EasyScanner.nextString(); // to read a string
        System.out.println("You entered: " + s);
        System.out.println();

        System.out.print("Enter a character: ");
        char c = EasyScanner.nextChar(); // to read a character
        System.out.println("You entered: " + c);
        System.out.println();
    }
}
```

You can see from this program how easy it is to call the methods, just by using the name of the class itself—for example:

```
double d = EasyScanner.nextDouble();
```

In the next chapter you will see how it is possible to do this.

Here is a sample run:

```
Enter a double: 23.6
You entered: 23.6
```

```
Enter an integer: 50
You entered: 50

Enter a string: Hello world
You entered: Hello world

Enter a character: B
You entered: B
```

You are now free to use the `EasyScanner` class if you wish. You can copy it from the website—as usual make sure it is in the right place for your compiler to find it.

7.8 The *Console* Class

In Chap. 1 we talked briefly about the possibility of running a program in a console, the text window provided by operating systems such as Windows™. We will discuss this in more detail in Chap. 19.

A special class called `Console` is provided (in the `java.io` library) as an alternative to `Scanner`, should you wish to use it—you should note however that the `Console` class is not suitable for running programs within an IDE. An example of a little program that uses the `Console` class is shown in Fig. 7.4.

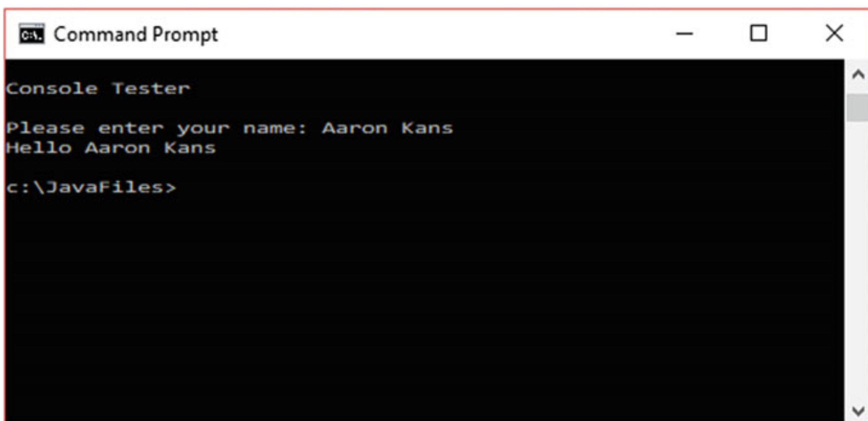


Fig. 7.4 Using the `Console` class

The program below shows how keyboard input is obtained with the `Console` class:

ConsoleTester

```
// demonstration of the console class for keyboard input

import java.io.Console;

public class ConsoleTester
{
    public static void main(String[] args)
    {
          System.out.println();
          System.out.println("Console Tester");
          System.out.println();
          Console con = System.console();
          String name; // declaration of a String
          name = con.readLine("Please enter your name: "); // allow user to enter name
          System.out.println("Hello " + name); // display a message to the user
    }
}
```

You can see from the example in Fig. 7.4 that there is no problem entering a string containing spaces; if, however, you wanted to use the `Console` class for entering **`double`**s or **`int`**s, you would have to enter a string and then convert this to the desired type. You will find out how to do this in Chap. 10.

7.9 The *BankAccount* Class

We have created a class called `BankAccount`, which you can download. This could be a very useful class in a real world application—for example as part of a financial control system. Once again you do not need to look at the details of how this class is coded in order to use it. You do need to know, however, that the class holds three pieces of information—the account number, the account name and the account balance. The first two of these will be `String` objects and the final one will be a variable of type **`double`**.

The methods are listed in Table 7.4.

The methods are straightforward, although you should pay particular attention to the `withdraw` method. Our simple `BankAccount` class does not allow for an overdraft facility, so, unlike the `deposit` method, which simply adds the specified amount to the balance, the `withdraw` method needs to check that the amount to be withdrawn is not greater than the balance of the account; if this were to be the case then the balance would be left unchanged. The method returns a **`boolean`** value to indicate if the withdrawal was successful or not. A **`boolean`** value of **`true`** would indicate success and **`boolean`** value of **`false`** would indicate failure. This enables a program that uses the `BankAccount` class to check whether the withdrawal has been made successfully. You can see how this is done in the program that follows, which makes use of the `BankAccount` class.

Table 7.4 The methods of the *BankAccount* class

Method	Description	Inputs	Output
`BankAccount`	A constructor. It accepts two strings and assigns them to the account number and account name respectively. It also sets the account balance to zero	Two `String` objects	Not applicable
`getAccountNumber`	Returns the account number	None	An item of type `String`
`getAccountName`	Returns the account name	None	An item of type `String`
`getBalance`	Returns the balance	None	An item of type **`double`**
`deposit`	Accepts an item of type **`double`** and adds it to the balance	An item of type **`double`**	None
`withdraw`	Accepts an item of type **`double`** and checks if there are sufficient funds to make a withdrawal. If there are not, then the method terminates and returns a value of **`false`**. If there are sufficient funds, however, the method subtracts the amount from the balance and returns a value of **`true`**	An item of type **`double`**	An item of type **`boolean`**

BankAccountTester

```
import java.util.Scanner;

public class BankAccountTester
{
   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner(System.in);
      double amount;
      boolean ok;

      BankAccount account1 = new BankAccount("99786754","Susan Richards");

      System.out.print("Enter amount to deposit: ");
      amount = keyboard.nextDouble();
      account1.deposit(amount);
      System.out.println("Deposit was made");
      System.out.println("Balance = " + account1.getBalance());
      System.out.println();


      System.out.print("Enter amount to withdraw: ");
      amount = keyboard.nextDouble();
      ok = account1.withdraw(amount); // get the return value of the withdraw method
      if(ok)
      {
         System.out.println("Withdrawal made");
      }
      else
      {
         System.out.println("Insufficient funds");
      }
      System.out.println("Balance = " + account1.getBalance());
      System.out.println();
   }
}
```

The program creates a `BankAccount` object and then asks the user to enter an amount to deposit. It then confirms that the deposit was made and shows the new balance.

It then does the same thing for a withdrawal. The `withdraw` method returns a **`boolean`** value indicating if the withdrawal has been successful or not, so we have assigned this return value to a **`boolean`** variable, `ok`:

```
ok = account1.withdraw(amount);
```

Depending on the value of this variable, the appropriate message is then displayed:

```
if(ok)
{
   System.out.println("Withdrawal made");
}
else
{
   System.out.println("Insufficient funds");
}
```

Two sample runs from this program are shown below. In the first the withdrawal was successful:

Enter amount to deposit: ***1000***
Deposit was made
Balance = 1000.0

Enter amount to withdraw: ***400***
Withdrawal made
Balance = 600.0

In the second there were not sufficient funds to make the withdrawal:

Enter amount to deposit: ***1000***
Deposit was made
Balance = 1000.0

Enter amount to withdraw: ***1500***
Insufficient funds
Balance = 1000.0

7.10 Arrays of Objects

In Chap. 6 you learnt how to create arrays of simple types such as **int** and **char**. It is perfectly possible, and often very desirable, to create arrays of objects. There, are, however, some important issues that we need to be aware of. We will illustrate this with a new version of the BankAccountTester from the previous section. In BankAccountTester2, instead of creating a single bank account, we have created several bank accounts by using an array. Take a look at the program, and then we will explain the important issues to you:

BankAccountTester2

```
public class BankAccountTester2
{
    public static void main(String[] args)
    {
        // create an array of references
        BankAccount[] accountList = new BankAccount[3];
        // create three new accounts, referenced by each element in the array
        accountList[0] = new BankAccount("99786754","Susan Richards");
        accountList[1] = new BankAccount("44567109","Delroy Jacobs");
        accountList[2] = new BankAccount("46376205","Sumana Khan");

        // make various deposits and withdrawals
        accountList[0].deposit(1000);
        accountList[2].deposit(150);
        accountList[0].withdraw(500);

        // print details of all three accounts
        for(BankAccount item : accountList)
        {
            System.out.println("Account number: " + item.getAccountNumber());
            System.out.println("Account name: " + item.getAccountName());
            System.out.println("Current balance: " + item.getBalance());
            System.out.println();
        }
    }
}
```

The first line of the main method looks no different from the statements that you saw in the last chapter that created arrays of primitive types:

```
BankAccount[] accountList = new BankAccount[3];
```

However, what is actually going on behind the scenes is slightly different. The above statement does *not* set up an array of BankAccount objects in memory; instead it sets up an array of *references* to such objects (see Fig. 7.5).

At the moment, space has been reserved for the three BankAccount *references* only, *not* the three BankAccount objects. As we told you earlier, when a reference is initially created it points to the constant **null**, so at this point each reference in the array points to **null**.

This means that memory would still need to be reserved for individual BankAccount objects each time we wish to link a BankAccount object to the array. We can now create new BankAccount objects and associate them with elements in the array as we have done with these lines:

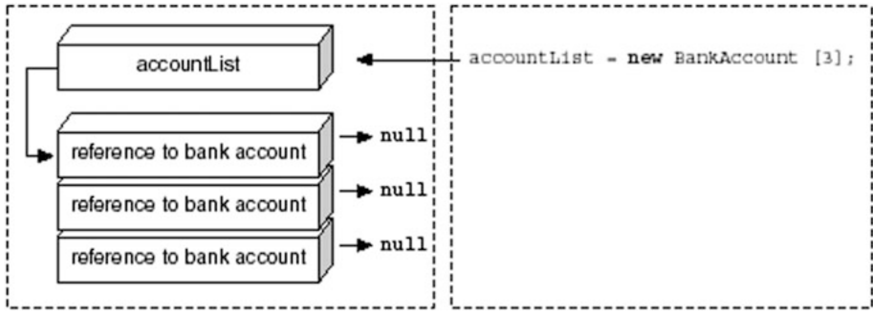


Fig. 7.5 The effect on computer memory of creating an array of objects

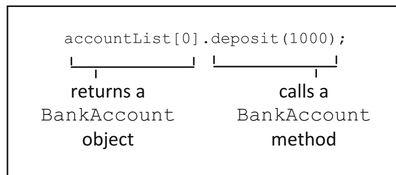
```
accountList[0] = new BankAccount("99786754","Susan Richards");
accountList[1] = new BankAccount("44567109","Delroy Jacobs");
accountList[2] = new BankAccount("46376205","Sumana Khan");
```

Three `BankAccount` objects have been created; the first one, for example, has account number of "99786754" and name "Susan Richards", and the reference at `accountList[0]` is set to point to it. This is illustrated in Fig. 7.6.

Once we have created these accounts, we make some deposits and withdrawals.

```
accountList[0].deposit(1000);
accountList[2].deposit(150);
accountList[0].withdraw(500);
```

Look carefully at how we do this. To call a method of a particular array element, we place the dot operator after the final bracket of the array index. This is made clear below:



Notice that in this case when we call the `withdraw` method we have decided not to check the **`boolean`** value returned.

```
accountList[0].withdraw(500); // return value not checked
```

It is not always necessary to check the return value of a method and you may ignore it if you choose.

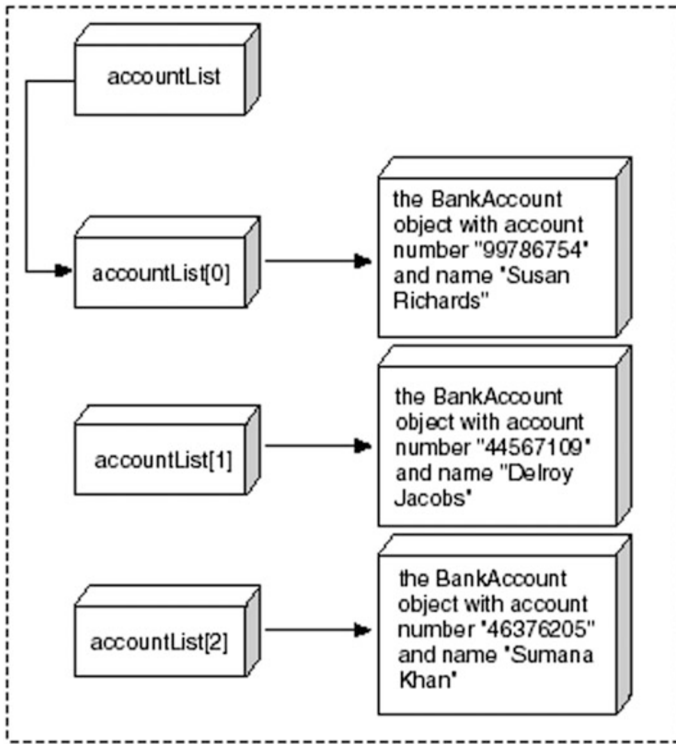


Fig. 7.6 Objects are linked to arrays by reference

Having done this we display the details of all three accounts. As we are accessing the entire array, we are able to use an enhanced **for** loop for this purpose; and since we are dealing with an array of `BankAccount` objects here, the type of the items is specified as `BankAccount`.

```
for(BankAccount item : accountList) // type of items is BankAccount
{
    System.out.println("Account number: " + item.getAccountNumber());
    System.out.println("Account name: " + item.getAccountName());
    System.out.println("Current balance: " + item.getBalance());
    System.out.println();
}
```

As you might expect, the output from this program is as follows:

```
Account number: 99786754
Account name: Susan Richards
Current balance: 500.0
```

```
Account number: 44567109
Account name: Delroy Jacobs
Current balance: 0.0

Account number: 46376205
Account name: Sumana Khan
Current balance: 150.0
```

7.11 The *ArrayList* Class

When you studied arrays in Chap. 6, it might have occurred to you that working with arrays can sometimes seem a bit cumbersome. That is because arrays are what we could term a "low-level" construct, which means that they mirror quite closely the workings of the computer itself, and match the way that data is stored in memory. As computer programmers it is vital that you have an understanding of arrays and how they work; however in modern day computing it is not uncommon for a programming language to provide higher level classes whose methods are closer to how we do things in real life; these classes deal with low level detail "behind the scenes", thus enabling the user to keep such detail at arms length.

One example of higher level classes are known as collection classes. These classes allow the programmer to deal with collections such as simple lists (like the `BankAccount` list from the previous section) with methods that will simply add a new item to the end of the list or easily remove an item from a particular position.

Collection classes are examples of classes known as **generic** classes. There are some quite advanced concepts involved in understanding generic classes, so most of this is left until the second semester. In particular we will examine collection classes in depth in Chap. 15. However, here we are going to introduce you to one such collection class called `ArrayList` (which resides in the `java.util` library) and provide you with just enough information to get you started.

You will see in the program that follows that there is some new notation involved. This is because when we declare an object of a collection class we need to indicate the type of object that it will hold, so you will see declarations like this:

```
ArrayList<BankAccount> accountList;
```

The type of object held is indicated within the angle brackets; so `accountList` will hold a list of `BankAccounts`. Similarly, if you wanted to declare an `ArrayList` object called `names` which was to hold `Strings`, you would do so like this:

```
ArrayList<String> names;
```

The program below, `BankAccountTester3`, rewrites `BankAccountTester2` using an `ArrayList` instead of an array.

BankAccountTester3

```
import java.util.ArrayList;

public class BankAccountTester3
{
    public static void main(String[] args)
    {
        // create an array of references
        ArrayList<BankAccount> accountList = new ArrayList<>();

        // create three new accounts, referenced by each element in the array
        accountList.add(new BankAccount("99786754","Susan Richards"));
        accountList.add(new BankAccount("44567109","Delroy Jacobs"));
        accountList.add(new BankAccount("46376205","Sumana Khan"));

        // make various deposits and withdrawals
        accountList.get(0).deposit(1000);
        accountList.get(2).deposit(150);
        accountList.get(0).withdraw(500);

        // print details of all three accounts
        for(BankAccount item : accountList)
        {
            System.out.println("Account number: " + item.getAccountNumber());
            System.out.println("Account name: " + item.getAccountName());
            System.out.println("Current balance: " + item.getBalance());
            System.out.println();
        }
    }
}
```

As you can see the full declaration of `accountList` is as follows:

```
ArrayList<BankAccount> accountList = new ArrayList<>();
```

When we call the constructor of a generic class we don't need to re-state the type of object held, so the angle brackets are left empty (this is sometimes referred to as a "diamond"). The compiler infers the type from the first part of the declaration—this is an example of **type inference** (more about this in Chap. 13).

The other thing you will notice is that we don't need to specify how many items an `ArrayList` will hold. It expands and contracts dynamically as we add and remove items.

To add items to the list we use the `add` method of `ArrayList`, which takes as a parameter the object that we are adding:

```
accountList.add(new BankAccount("99786754","Susan Richards"));
accountList.add(new BankAccount("44567109","Delroy Jacobs"));
accountList.add(new BankAccount("46376205","Sumana Khan"));
```

To make deposits and withdrawals, we need to retrieve the individual items that we require. We do this with the `get` method of `ArrayList`, which takes as a parameter the particular index (starting with zero, as with arrays):

```
accountList.get(0).deposit(1000);
accountList.get(2).deposit(150);
accountList.get(0).withdraw(500);
```

Finally we display the items—the enhanced for loop works nicely with classes such as `ArrayList`, so the code for displaying the bank accounts is the same as in the original program:

```
for(BankAccount item : accountList)
{
   System.out.println("Account number: " + item.getAccountNumber());
   System.out.println("Account name: " + item.getAccountName());
   System.out.println("Current balance: " + item.getBalance());
   System.out.println();
}
```

One final word about collection classes such as `ArrayList`. These classes cannot be used to hold primitive types such as **`int`** and **`double`**. But don't worry—there is a way around this using what are known as **wrapper** classes. These will be introduced to you in Chap. 9.

7.12 Self-test Questions

1. Examine the program below and then answer the questions that follow:

```
public class SampleProgram
{
     public static void main(String[] args)
     {
        Oblong oblong1 = new Oblong(3.0, 4.0);
        Oblong oblong2 = new Oblong(5.0, 6.0);
        System.out.println("The area of oblong1 is " + oblong1.calculateArea());
        System.out.println("The area of oblong2 is " + oblong2.calculateArea());
     }
}
```

(a) By referring to the program above distinguish between a *class* and an *object*.

(b) By referring to the program above explain the purpose of the *constructor*.

(c) By referring to the program above explain how you call the method of one class from another class.

(d) What output would you expect to see from the program above?

2. (a) Write the code that will create two `BankAccount` objects, `acc1` and `acc2`. The account number and account name of each should be set at the time the object is created.

(b) Write the lines of code that will deposit an amount of 200 into `acc1` and 100 into `acc2`.

(c) Write the lines of code that attempt to withdraw an amount of 150 from `acc1` and displays the message "WITHDRAWAL SUCCESSFUL" if the amount was withdrawn successfully and "INSUFFICIENT FUNDS" if it was not.

(d) Write a line of code that will display the balance of `acc1`.

(e) Write a line of code that will display the balance of `acc2`.

3. In what way does calling methods from the `EasyScanner` class differ from calling methods from the other classes you have met (`BankAccount`, `Oblong`, `String` and `Scanner`)?

4. Consider the following fragment of code that initializes one string constant with a password ("java") and creates a second string to hold the user's guess for the password. The user is then asked to enter their guess:

```
String final PASSWORD = "java"; // set password
String guess; // to hold user's guess
System.out.print("Enter guess: ");
```

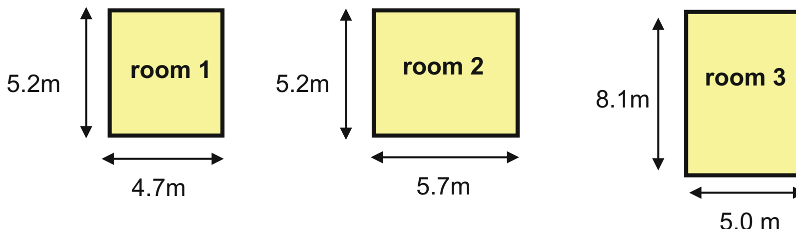
(a) Write a line of code that uses the `EasyScanner` class to read the guess from the keyboard.

(b) Write the code that displays the message "CORRECT PASSWORD" if the user entered the correct password and "INCORRECT PASSWORD" if not.

5. How do arrays of objects differ from arrays of primitive types?

6. (a) Declare an array called `rooms`, to hold three `Oblong` objects. Each `Oblong` object will represent the dimensions of a room in an apartment.

(b) The three rooms in the apartment have the following dimensions:
Add three appropriate `Oblong` objects to the `rooms` array to represent these 3 rooms.



(c) Write the line of code that would make use of the `rooms` array to display the area of room 3 to the screen.

7. Repeat the previous question using and `ArrayList` instead of an array.

7.13 Programming Exercises

In order to tackle these exercises make sure that the classes `Oblong`, `BankAccount` and `EasyScanner` have been copied from the website and placed in the correct directory for your compiler to access them.

1. (a) Implement the program given in self-test question 1 and run it to confirm your answer to part (d) of that question.
   (b) Adapt the program above so that the user is able to set the length and height of the two oblongs. Make use of the `EasyScanner` class to read in the user input.

2. Consider a program to enter and confirm a suitable code name for an agent. Declare two string objects, called `codeName` and `confirm` and then

   (a) Prompt to get the user to enter a suitable name into the `codeName` string;

   (b) Use a **`while`** loop to ensure that the string entered is greater than 6 characters in length, if it is not print "INVALID CODENAME" and ask the user to re-enter a code name;

   (c) Once a valid code name has been entered ask the user to re-enter the code name into the `confirm` string and then use an **`if else`** statement to ensure that the string entered matches the original code name; if it does, print a message "CODE NAME CONFIRMED" otherwise print a message saying "CODE NAME MIS-MATCH";

   (d) Use the `charAt` method to ensure that the code name ends with an 'X' character;

   (e) Finally use the `startsWith` method to ensure that, as well as being greater than 6 characters in length, the code name entered also starts with the words "Agent".

3. Adapt the `StringComparison` program from Sect. 7.6.3, which compares two strings, in the following ways:

   (a) Rewrite the program so that it ignores case;

   (b) Rewrite the program, using the `equals` method, so that all it does is to test whether the two strings are the same;

   (c) Repeat (b) using the `equalsIgnoreCase` method;

   (d) Use the `trim` method so that the program ignores leading or trailing spaces.

4. Design and implement a program that performs in the following way:

- When the program starts, two bank accounts are created, using names and numbers which are written into the code;
- The user is then asked to enter an account number, followed by an amount to deposit in that account;
- The balance of the appropriate account is then updated accordingly—or if an incorrect account number was entered a message to this effect is displayed;
- The user is then asked if he or she wishes to make more deposits;
- If the user answers does wish to make more deposits, the process continues;
- If the user does not wish to make more deposits, then details of both accounts (account number, account name and balance) are displayed.

5. Write a program that creates an array of `Oblong` objects to represent the dimensions of rooms in an apartment as described in self test question 6. The program should allow the user to:

   - Determine the number of rooms;
   - Enter the dimensions of the rooms;
   - Retrieve the area and dimensions of any of the rooms.

6. Repeat the previous question making use of the `ArrayList` class.

8 Implementing Classes

Outcomes:

By the end of this chapter you should be able to:

- *design classes using the notation of the* ***Unified Modeling Language (UML)***;
- *write the Java code for a specified class*;
- *explain the difference between* **`public`** *and* **`private`** *access to attributes and methods*;
- *explain the meaning of the term* ***encapsulation***;
- *explain the use of the* **`static`** *keyword*;
- *pass objects as parameters*;
- *develop their own* ***collection classes*** *in Java*;
- *identify the advantages of object-oriented programming*.

8.1 Introduction

This chapter is arguably the most important so far, because it is here that you are going to learn how to develop the classes that you need for your programs. You are already familiar with the concept of a class, and the idea that we can create objects that belong to a class; in the last chapter you saw how to create and use objects, and you saw how we could use the methods of a class without knowing anything about how they work.

In this chapter you will look inside the classes you have studied to see how they are constructed, and how you can write classes of your own. We start with the `Oblong` class.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_8

8.2 Designing Classes in UML Notation

In the last chapter you saw that a class consists of:

- a set of **attributes** (the data);
- a set of **methods** that can access or change those attributes.

When we design a class we must, of course, consider what data the class needs to hold, and what methods are needed to access that data. The `Oblong` class that we develop here will need to hold two items of data—the length and the height of the oblong; these will have to be real numbers, so **`double`** would be the appropriate type for each of these two attributes. You have already seen the methods that we provided for this class in Table 7.1 in the previous chapter.

When we design classes, it is very useful to start off by using a diagrammatic notation. The usual way this is done is by making use of the notation of the **Unified Modeling Language (UML)**.[1] In this notation, a class is represented by a box divided into three sections. The first section provides the name of the class, the second section lists the attributes, and the third section lists the methods. The UML class diagram for the `Oblong` class is shown in Fig. 8.1.

You can see that the UML notation requires us to indicate the names of the attributes along with their types, separated by a colon.

In the last chapter we introduced you to the concept of *encapsulation* or information-hiding. This is the technique of making attributes accessible only to the methods of the same class, and it is this feature of object-oriented languages that has contributed to object-orientation becoming the standard way of programming in today's world. By restricting access in this way, programmers can keep the data in their classes "sealed off" from other classes, because they are the ones in control of how it is actually accessed.

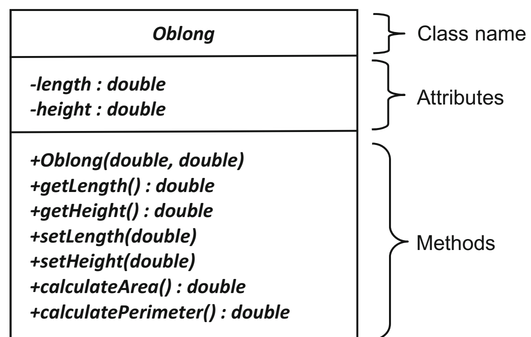


Fig. 8.1 The design of the *Oblong* class

[1]Martina Seidl et al., *UML @Classroom, An Introduction to Object Oriented Modeling*, Springer 2015.

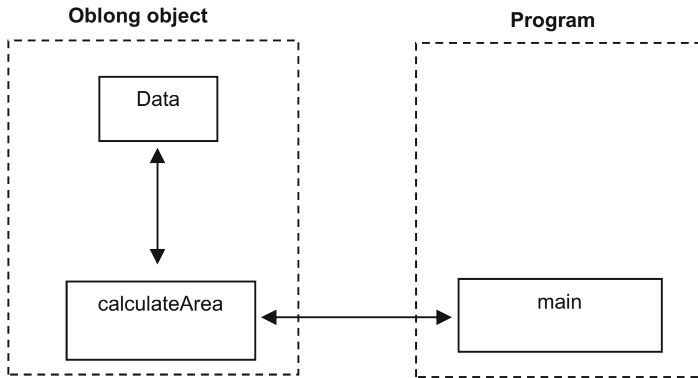


Fig. 8.2 Encapsulation requires data be kept hidden inside an object

The way our `Oblong` class has been set up means that you cannot directly use the `length` and `height` attributes in another program. If you want to find out the area of the oblong in, say, the `main` method of another program then you can't do this by accessing the `length` and `height` data directly, because access to these attributes is denied.

Instead we would, as you know, call the `calculateArea` method of the `Oblong` object. We design our classes like this because doing so means that no-one can inadvertently *change* the values of `length` and `height`—our data is kept secure. If access to these attributes were not restricted in this way, then the `length` and `height` data could inadvertently be changed. Instead we limit access of the `Oblong` class to its methods. This is illustrated in Fig. 8.2.

The plus and minus signs that you can see in the UML diagram in Fig. 8.1 are all to do with this idea of encapsulation; a minus sign means that the attribute or method is **private**—that is, it is accessible *only to methods within the same class*. A plus sign means that it is **public**—it is accessible to methods of other classes. Normally we make the attributes private, and the methods public, in this way achieving encapsulation. You will see how it is done in a Java class in the next section.

Now let's consider the notation for the methods. You can see from the diagram that the parameter types are given in brackets—for example:

+setLength(double)

Here you can see that the `setLength` method requires one **`double`** parameter (in this case the new length value). The return types are placed after the brackets, preceded by a colon—for example:

+getLength() : double

Here you can see that the `getLength` method returns a value of type **`double`** (in this case the current value of the private `length` attribute).

Where there is no return type, nothing appears after the brackets, as in the `setlength` and `setHeight` methods.

The first method, `Oblong`, is the constructor. As we know the constructor always has the same name as the class, and in this case it requires two parameters of type **`double`**:

+Oblong(double, double)

You should note that a constructor *never has a return type*. In fact you will see later that in Java we don't even put the word **`void`** in front of a constructor; if we did the compiler would think it was a regular method.

As you saw in the previous chapter, we have provided our `Oblong` class with methods for reading and writing to the attributes—and it is conventional to begin the name of such methods with `get`- and `set`-respectively. However, it is not always the case that we choose to supply methods such as `setLength` and `setHeight`, which allow us to *change* the attributes. Sometimes we set up our class so that the only way that we can assign values to the attributes is via the constructor. This would mean that the values of the `length` and `height` could be set only at the time a new `Oblong` object was created, and could not be changed after that. Whether or not you want to provide a means of writing to individual attributes depends on the nature of the system you are developing and should be discussed with potential users. However, we believe that it is a good policy to provide write access to only those attributes that clearly require to be changed during the object's lifetime, and we have taken this approach throughout this book. In this case we have included "set" methods for `length` and `height` because we are going to need them in Chap. 10.

8.3 Implementing Classes in Java

8.3.1 The *Oblong* Class

Now that we have the basic design of the `Oblong` class we can go ahead and write the Java code for it. We present the code here—when you have had a look at it we will discuss it.

The *Oblong* class

```
public class Oblong
{
    // the attributes
    private double length;
    private double height;

    // the methods

    // the constructor
    public Oblong(double lengthIn, double heightIn)
    {
        length = lengthIn;
        height = heightIn;
    }

    // this method allows us to read the length attribute
    public double getLength()
    {
        return length;
    }

    // this method allows us to read the height attribute
    public double getHeight()
    {
        return height;
    }

    // this method allows us to write to the length attribute
    public void setLength(double lengthIn)
    {
        length = lengthIn;
    }

    // this method allows us to write to the height attribute
    public void setHeight(double heightIn)
    {
        height = heightIn;
    }

    // this method returns the area of the Oblong
    public double calculateArea()
    {
        return length * height;
    }

    // this method returns the perimeter of the Oblong
    public double calculatePerimeter()
    {
        return 2 * (length + height);
    }
}
```

Let's take a closer look at this. The first line declares the `Oblong` class:

```
public class Oblong
```

Next come the attributes. An `Oblong` object will need attributes to hold values for the length and the height of the oblong, and these will be of type **`double`**. The declaration of the attributes in the `Oblong` class took the following form in our UML diagram:

-length : double
-height : double

In Java this is implemented as:

```
private double length;
private double height;
```

As you can see, attributes are declared like any other variables, except that they are declared *outside* of any method, and they also have an additional word in front of them—the word **private**, corresponding to the minus sign in the UML notation. In Java, this keyword is used to restrict the scope of the attributes to methods of this class only, as we described above.

You should note that the attributes of a class are accessible to *all* the methods of the class—unlike *local* variables, which are accessible only to the methods in which they are declared.

Figure 8.1 made it clear which methods we need to define within our Oblong class. First comes the constructor. You should recall that it has the same name as the class, and, unlike any other method, it has no return type—not even **void**! It looks like this.

```
public Oblong(double lengthIn, double heightIn)
{
   length = lengthIn;
   height = heightIn;
}
```

The first thing to notice is that this method is declared as **public**. Unlike the attributes, we want our methods to be accessible from outside so that they can be called by methods of other classes.

In our class we are defining the constructor so that when a new Oblong object is created (with the keyword **new**) then not only do we get some space reserved in memory, but some other stuff occurs also; in this case two assignment statements are executed. The first assigns the value of the parameter lengthIn to the length attribute, and the second assigns the value of the parameter heightIn to the height attribute. We are sticking to our naming convention for attributes here by appending the word 'In' to them, but of course you can use any names for your parameters. Remember once again that the attributes are visible to *all* the methods of the class.

When we define a constructor like this in a class it is termed a *user-defined*[2] constructor. If we don't define our own constructor, then one is automatically provided for us—this is referred to as the **default** constructor. The default constructor takes no parameters and when it is used to create an object—for example in a line like this:

```
Oblong myOblong = new Oblong();
```

[2]Here the word *user* is referring to the person *writing* the program, not the person using it!

then all that happens is that memory is reserved for the new object—no other processing takes place. Any attributes will be given initial values according to the rules that we give you later in Sect. 8.5.

One more thing about constructors: once we have defined our own constructors, this default constructor is no longer automatically available. If we want it to be available then we have to re-define it explicitly. In the `Oblong` case we would define it as:

```
public Oblong()
{
}
```

You can see that just like regular methods, constructers can be overloaded, and we can define several constructors in one class. When we create an object it will be clear from the parameter list which constructor we are referring to.

Now let's take a look at the definition of the next method, `getLength`. The purpose of this method is simply to send back the value of the `length` attribute. In the UML diagram it was declared as:

+getLength() : double

In Java this becomes:

```
public double getLength()
{
    return length;
}
```

Once again you can see that the method has been declared as **`public`** (indicated by the plus sign in UML), enabling it to be accessed by methods of other classes.

The next method, `getHeight`, behaves in the same way in respect of the `height` attribute.

Next comes the `setLength` method:

+setLength(double)

We implement this as:

```
public void setLength(double lengthIn)
{
    length = lengthIn;
}
```

This method does not return a value, so its return type is **`void`**. However, it does require a parameter of type **`double`** that it will assign to the `length` attribute. The body of the method consists of a single line which assigns the value of `lengthIn` to the `length` attribute.

The next method, `setHeight`, behaves in the same way in respect of the `height` attribute.

After this comes the calculateArea method:

+calculateArea() : double

We implement this as:

```
public double calculateArea()
{
   return length * height;
}
```

Once again there are no formal parameters, as this method does not need any data in order to do its job; it returns a **double**. The actual code is just one line, namely the statement that returns the area of the oblong, calculated by multiplying the value of the length attribute by the value of the height attribute.

The calculatePerimeter method is similar and thus the definition of the Oblong class is now complete.

One important thing to note here. Unlike some of the methods we developed in Chap. 5, the methods that we have defined here deal only with the basic *functionality* of the class—they do not include any routines that deal with input or output. That is because the methods of Chap. 5 were only being used by the class in which they were written—but now our methods will be used by other classes that we cannot as yet predict. So when developing a class we should always strive to restrict our methods to the essential functions that define the class (in this case, for example, calculating the area and perimeter of the oblong), and to exclude anything that is concerned with the input or output functions of a program. If we do this, then our class can be used in any sort of application, regardless of whether it is a simple console application like the ones we have developed so far, or a complex graphical application like the ones you will come across later in this book.

8.3.2 The *BankAccount* Class

The UML class diagram for the BankAccount class, which we used in the previous chapter, is shown in Fig. 8.3.

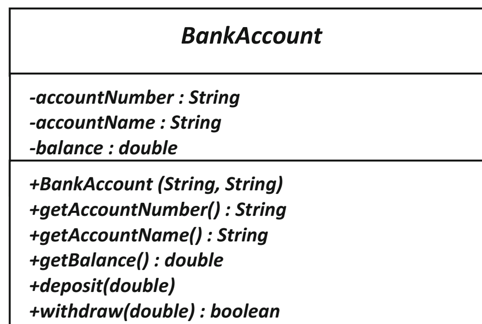


Fig. 8.3 The design of *BankAccount* class

You will notice here that *accountNumber* and *accountName* are declared as *Strings*; it is perfectly possible for the attributes of one class to be objects of another class.

We can now inspect the code for this class:

The *BankAccount* class

```
public class BankAccount
{
    // the attributes
    private String accountNumber;
    private String accountName;
    private double balance;

    // the methods

    // the constructor
    public BankAccount(String numberIn, String nameIn)
    {
        accountNumber = numberIn;
        accountName = nameIn;
        balance = 0;
    }

    // methods to read the attributes
    public String getAccountName()
    {
        return accountName;
    }

    public String getAccountNumber()
    {
        return accountNumber;
    }

    public double getBalance()
    {
        return balance;
    }

    // methods to deposit and withdraw money
    public void deposit(double amountIn)
    {
        balance = balance + amountIn;
    }

    public boolean withdraw(double amountIn)
    {
        if(amountIn > balance)
        {
            return false; // no withdrawal was made
        }
        else
        {
            balance = balance - amountIn;
            return true; // money was withdrawn successfully
        }
    }
}
```

Now that we are getting the idea of how to define a class in Java, we do not need to go into so much detail in our analysis and explanation.

The first three lines declare the attributes of the class, and are as we would expect:

```
private String accountNumber;
private String accountName;
private double balance;
```

Now the constructor:

```
public BankAccount(String numberIn, String nameIn)
{
   accountNumber = numberIn;
   accountName = nameIn;
   balance = 0;
}
```

You can see that when a new object of the BankAccount class is created, the accountName and accountNumber will be assigned the values of the parameters passed to the method. In this case, the balance will be assigned the value zero; this makes sense because when someone opens a new account there is a zero balance until a deposit is made.[3]

The next three methods, getAccountNumber, getAccountName and getBalance, are all set up so that the values of the corresponding attributes (which of course have been declared as **private**) can be read.

After these we have the deposit method:

```
public void deposit(double amountIn)
{
   balance = balance + amountIn;
}
```

This method does not return a value; it is therefore declared to be of type **void**. It does however require that a value is sent in (the amount to be deposited), and therefore has one parameter—of type **double**—in the brackets. As you would expect with this method, the action consists of adding the deposit to the balance attribute of the BankAccount object.

Now the withdraw method:

```
public boolean withdraw(double amountIn)
{
       if(amountIn > balance)
       {
          return false; // no withdrawal was made
       }
       else
       {
          balance = balance - amountIn;
          return true; // money was withdrawn successfully
       }
 }
```

[3]You would be right in thinking that the balance attribute would automatically be assigned a value of zero if we did not specifically do that here. However it is good practice always to ensure that variables are initialized with the values that we require—particularly because in many other programming languages attributes are not initialized as they are in Java.

The amount is subtracted only if there are sufficient funds—in other words if the amount to be withdrawn is no bigger than the balance. If this is not the case then a value of **false** is returned and the method terminates. Otherwise the amount is subtracted from the balance and a value of **true** is returned. The return type of the method therefore is **boolean**.

8.4 The *static* Keyword

You have already seen the keyword **static** in front of the names of methods in some Java classes. A word such as this (as well as the words **public** and **private**) is called a **modifier**. A modifier determines the particular way a class, attribute or method is accessed.

Let's explore what this **static** modifier does. Consider the BankAccount class that we discussed in the previous section. Say we wanted to have an additional method which added interest, at the current rate, to the customer's balance. It would be useful to have an attribute called interestRate to hold the value of the current rate of interest. But of course the interest rate is the same for any customer—and if it changes, we want it to change for every customer in the bank; in other words for every object of the class. We can achieve this by declaring the variable as **static**. An attribute declared as **static** is a *class* attribute; any changes that are made to it are made to all the objects in the class. The way this is achieved is by the program creating only one copy of the attribute and making it accessible to all objects.

It would make sense if there were a way to access this attribute without reference to a specific object; and so there is! All we have to do is to declare methods such as setInterestRate and getInterestRate as **static**. This makes a method into a *class* method; it does not refer to any specific object. As you will see in our next program, BankAccountTester2, we can call a class method by using the class name instead of the object name.

Fig. 8.4 The design of the *BankAccount2* class

BankAccount2
-accountNumber : String *-accountName : String* *-balance : double* *-interestRate : double*
+BankAccount2 (String, String) *+getAccountNumber() : String* *+getAccountName() : String* *+getBalance() : double* *+deposit(double)* *+withdraw(double) : boolean* *+setInterestRate(double)* *+getInterestRate() : double* *+addInterest()*

We have rewritten our BankAccount class, and called it BankAccount2. We have included three new methods as well as the new **static** attribute interestRate. The first two of these—setInterestRate and getInterestRate—are the methods that allow us to read and write to our new attribute. These have been declared as **static**. The third—addInterest—is the method that adds the interest to the customer's balance. As can be seen in Fig. 8.4, the UML notation is to underline static attributes and methods.

Here is the code for the class. The new items have been emboldened.

BankAccount2 - the modified _BankAccount_ class

```
public class BankAccount2
{
    private String accountNumber;
    private String accountName;
    private double balance;
    private static double interestRate;

    public BankAccount2(String numberIn, String nameIn)
    {
        accountNumber = numberIn;
        accountName = nameIn;
        balance = 0;
    }

    public String getAccountName()
    {
        return accountName;
    }

    public String getAccountNumber()
    {
        return accountNumber;
    }

    public double getBalance()
    {
        return balance;
    }

    public void deposit(double amountIn)
    {
    balance = balance + amountIn;
    }

            public boolean withdraw(double amountIn)
            {
                    if(amountIn > balance)
                    {
                            return false;
                    }
                    else
                    {
                            balance = balance - amountIn;
                            return true;
                    }
            }

    public static void setInterestRate(double rateIn)
    {
        interestRate = rateIn;
    }

    public static double getInterestRate()
    {
        return interestRate;
    }

    public void addInterest()
    {
        balance = balance + (balance * interestRate)/100;
    }
}
```

The following program, `BankAccountTester2`, uses this modified version of the `BankAccount` class.

BankAccountTester2

```
public class BankAccountTester2
{
    public static void main(String[] args)
    {
        // create a bank account
        BankAccount2 account1 = new BankAccount2("99786754","Gayle Forcewind");
        // create another bank account
        BankAccount2 account2 = new BankAccount2("99887776","Stan Dandy-Liver");
        // make a deposit into the first account
        account1.deposit(1000);
        // make a deposit into the second account
        account2.deposit(2000);
        // set the interest rate
        BankAccount2.setInterestRate(10);
        // add interest to accounts
        account1.addInterest();
        account2.addInterest();
        // display the account details
        System.out.println("Account number: " + account1.getAccountNumber());
        System.out.println("Account name: " + account1.getAccountName());
        System.out.println("Interest Rate " + BankAccount2.getInterestRate());
        System.out.println("Current balance: " + account1.getBalance());
        System.out.println(); // blank line
        System.out.println("Account number: " + account2.getAccountNumber());
        System.out.println("Account name: " + account2.getAccountName());
        System.out.println("Interest Rate " + BankAccount2.getInterestRate());
        System.out.println("Current balance: " + account2.getBalance());
    }
}
```

Take a closer look at the first four lines of the `main` method of the above program. We have created two new bank accounts which we have called `account1` and `account2`, and have assigned account numbers and names to them at the time they were created (via the constructor). We have then deposited amounts of 1000 and 2000 respectively into each of these accounts.

Now look at the next line:

```
BankAccount2.setInterestRate(10);
```

This line sets the interest rate to 10. Because `setInterestRate` has been declared as a **`static`** method, we have been able to call it by using the class name `BankAccount2`. Because `interestRate` has been declared as a **`static`** attribute this change is effective for any object of the class. Therefore, when we add interest to each account as we do with the next two lines:

```
account1.addInterest();
account2.addInterest();
```

we should expect it to be calculated with an interest rate of 10, giving us new balances of 1100 and 2200 respectively.

This is exactly what we get, as can be seen from the output below:

```
Account number: 99786754
Account name: Gayle Forcewind
Interest Rate 10.0
Current balance: 1100.0

Account number: 99887776
Account name: Stan Dandy-Liver
Interest Rate 10.0
Current balance: 2200.0
```

Class methods can be very useful indeed and we shall see further examples of them in this chapter. Of course, we have always declared our `main` method, and other methods within the same class as the `main` method, as **`static`**—because these methods belong to the class and not to a specific object.

8.5 Initializing Attributes

Looking back at the `BankAccount2` class in the previous section, some of you might have been asking yourselves what would happen if we called the `getInterestRate` method before the interest rate had been set using the `setInterestRate` method. In fact, the answer is that a value of zero would be returned. This is because, while Java does not give an initial value to *local* variables (which is why you get a compiler error if you try to use an uninitialized variable), Java always initializes attributes. Numerical attributes such as **`int`** and **`double`** are initialized to zero; **`boolean`** attributes are initialized to **`false`** and objects are initialized to **`null`**. Character attributes are given an initial Unicode value of zero.

Despite the above, it is nonetheless good programming practice always to give an initial value to your attributes, rather than leave it to the compiler. One very good reason for this is that you cannot assume that every programming language initializes variables in the same way—if you were using C++, for example, the initial value of any variable is completely a matter of chance—and you won't get a compiler error to warn you! In the `BankAccount2` class, it would have done no harm at all to have initialized the `interestRate` variable when it was declared:

```
private static double interestRate = 0;
```

In fact, one technique you could use is to give the `interestRate` attribute some special initial value (such as a negative value) to indicate to the user of this class that the interest rate had not been set. You will see another example where this technique can be used in question 2 of the programming exercises.

8.6 The *EasyScanner* Class

In the previous chapter we used a class called `EasyScanner` that could make keyboard input a lot easier. We have now covered all the concepts you need in order to understand how this class works. Here it is:

The *EasyScanner* class

```
import  java.util.Scanner;

public class EasyScanner
{
    public static int nextInt()
    {
        Scanner keyboard = new Scanner(System.in);
        int i = keyboard.nextInt();
        return i;
    }

    public static double nextDouble()
    {
        Scanner keyboard = new Scanner(System.in);
        double d = keyboard.nextDouble();
        return d;
    }

    public static String nextString()
    {
        Scanner keyboard = new Scanner(System.in);
        String s = keyboard.nextLine();
        return s;
    }

    public static char nextChar()
    {
        Scanner keyboard = new Scanner(System.in);
        char c = keyboard.next().charAt(0);
        return c;
    }
}
```

You can see that we have made every method a **`static`** method, so that we can simply use the class name when we call a method. For example:

```
int number = EasyScanner.nextInt();
```

You can see that the `nextString` method uses the `nextLine` method of the `Scanner` class—but as a new `Scanner` object is created each time the method is called there is no problem about using it after a `nextInt` or a `nextDouble` method as there is with `nextLine` itself.

We will use the `EasyScanner` class later, in Sect. 8.8.2.

8.7 Passing Objects as Parameters

In Chap. 5 it was made clear that when a variable is passed to a method it is simply the *value* of that variable that is passed—and that therefore a method cannot change the value of the original variable. In Chap. 6 you found out that in the case of an

array it is the value of the memory location (a *reference*) that is passed and consequently the value of the original array elements can be changed by the called method.

What about objects? Let's write a little program (`ParameterTest`) to test this out.

ParameterTest

```
public class ParameterTest
{
   public static void main(String[] args)
   {
      // create new bank account
      BankAccount testAccount = new BankAccount("1", "Ann T Dote");
      test(testAccount); // send the account to the test method
      System.out.println("Account Number: " + testAccount.getAccountNumber());
      System.out.println("Account Name: " + testAccount.getAccountName());
      System.out.println("Balance: " + testAccount.getBalance());
   }

   // a method that makes a deposit in the bank account
   static void test(BankAccount accountIn)
   {
      accountIn.deposit(2500);
   }
}
```

The output from this program is as follows:

```
Account Number: 1
Account Name: Ann T Dote
Balance: 2500.0
```

You can see that the deposit has successfully been made—in other words the attribute of the object has actually been changed. This is because what was sent to the method was, of course, a *reference* to the original `BankAccount` object, `testAccount`. Thus `accountIn` is a *copy* of the `testAccount` reference and so points to the original object and invokes that object's methods. So the following line of code:

```
accountIn.deposit(2500);
```

calls the `deposit` method of the original `BankAccount` object.

You might think this is a very good thing, and will make life easier for you as a programmer. However, you need a word of caution here. It is very easy inadvertently to allow a method to change an object's attributes, so you need to take care—more about this in the second semester.

8.8 Collection Classes

In Chap. 7 we introduced you to the idea of a collection class—a class which holds a collection of objects. We showed you how to use the ArrayList class to hold a collection of objects of a specific type, and how to use a couple of ArrayList methods.

Methods of Java's collection classes are of course not tailored to any specific type. A method such as remove, for example, requires us to send in a reference to the object to be removed. Normally, however, we would reference an object such as a bank account by a unique field such as an account number—so any program using an ArrayList would need to first search the list to find the object we want, and then to remove it.

While this approach would work perfectly well, it would be a lot more convenient if we could tailor our collection classes so that they provided the specific methods that we need. For example, in the case of a list of bank accounts it would be useful to deposit or withdraw funds from a specific account referenced by its account number, or to remove an account with a particular number.

We can do this quite easily by creating our own collection class with an attribute which is itself a collection, such as ArrayList. We have done this below; we have called our collection class Bank.

8.8.1 The *Bank* Class

When one object itself consists of other objects, this relationship is called **aggregation**. This association, represented in UML by a diamond, is often referred to as a *part-of* relationship. For example, the association between a car and the passengers in the car is aggregation. **Composition** (represented by a filled diamond) is a special, stronger, form of aggregation whereby the "whole" is actually dependent on the "part". For example, the association between a car and its engine is one of *composition*, as a car cannot exist without an engine. A collection class is an implementation of the aggregation relationship.

The association between the container object, Bank, and the contained object, BankAccount, is shown in the UML diagram of Fig. 8.5.

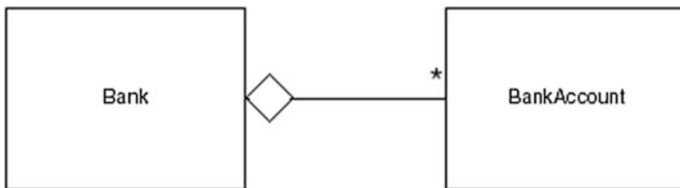


Fig. 8.5 The *Bank* object can contain many *BankAccount* objects

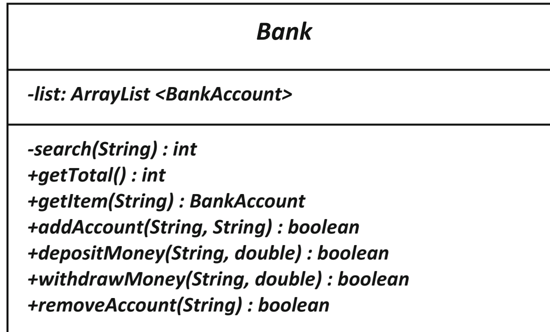


Fig. 8.6 The design of the *Bank* class

The asterisk at the other end of the joining line indicates that the `Bank` object contains *zero or more* `BankAccount` objects. The design for the `Bank` class is now given in Fig. 8.6.

As can be seen in Fig. 8.6, the class will have a single attribute, `list`, which is a collection of `BankAccounts`. Here we have decided to use an `ArrayList` of `BankAccount` objects to store this collection:

-list: ArrayList <BankAccount>

There are seven methods, which are described below:

-search(String) : int

This is what we can term a **helper** method; it will be declared as **`private`** (note the minus sign in the UML notation), because it is not intended for it to be called by other classes. It accepts a `String` representing the account number. It then returns the index of the account with that account number in the `ArrayList`. If the account number does not exist, then a "phoney" index (−999) will be returned to indicate failure.

+getTotal(): int

This method simply returns the total number of accounts currently in the system.

+getItem(String): BankAccount

This method receives a `String` representing an account number, and returns the `BankAccount` with that account number.

If the account number is not valid, a **`null`** value will be returned.

+addAccount(String, String): boolean

This method receives two strings representing the account number and name respectively, and adds an account with these details to the list of accounts. If an account with this number already exists, the new account will not be added and the method will return a value of **`false`**. However, if the operation has been completed successfully a value of **`true`** is returned.

+depositMoney(String, double) : boolean

Accepts a `String`, representing the account number of a particular account, and an amount of money which is to be deposited in that account. Returns **`true`** if the deposit was made successfully, or **`false`** otherwise (no such account number).

+withdrawMoney(String, double) : boolean

Accepts a `String`, representing the account number of a particular account, and an amount of money which is to be withdrawn from that account. Returns **`true`** if the withdrawal was made successfully, or **`false`** otherwise. The reason for the withdrawal not taking place could be that there is no such account number or that there are insufficient funds. In this version of `Bank`, the method does not indicate which of these reasons caused the failure—that is left for you as an end of chapter exercise.

+removeAccount(String) : boolean

Accepts a `String`, representing an account number, and removes that account from the list. Returns **`true`** if the account was removed successfully, or **`false`** otherwise (no such account number).

The code for the `Bank` class is presented below. Take a careful look at it, then we will discuss it.

The *Bank* class

```
import java.util.ArrayList;

public class Bank
{
    ArrayList<BankAccount> list = new ArrayList<>();

   // helper method to find the index of a specified account
   private int search(String accountNumberIn)
   {
       for(int i = 0; i <= list.size() - 1; i++)
       {
           BankAccount tempAccount = list.get(i); // find the account at index i
           String tempNumber = tempAccount.getAccountNumber(); // get account number
           if(tempNumber.equals(accountNumberIn)) // if this is the account we are looking for
           {
               return i; // return the index
           }
       }
       return -999;
   }

   // return the total number of items
   public int getTotal()
   {
       return list.size();
   }


   // return an account with a particular account number
   public BankAccount getItem(String accountNumberIn)
   {
       int index = search(accountNumberIn);
       if(index != -999) // check that account exists
       {
            return list.get(index);
       }
       else
       {
            return null; // no such account
       }
   }

   // add an item to the list
   public boolean addAccount(String accountNumberIn, String nameIn)
   {
       if(search(accountNumberIn) == -999) // check that account does not already exist
       {
           list.add(new BankAccount(accountNumberIn, nameIn)); // add new account
           return true;
       }
       return false;
   }

   // deposit money in a specified account
   public boolean depositMoney(String accountNumberIn, double amountIn)
   {
       BankAccount acc = getItem(accountNumberIn);
       if(acc != null)
       {
           acc.deposit(amountIn);
           return true; // indicate success
       }
       else
       {
           return false; // indicate failure
       }
   }

   // withdraw money from a specified account
   public boolean withdrawMoney(String accountNumberIn, double amountIn)
   {
       BankAccount acc = getItem(accountNumberIn);
       if(acc != null && acc.getBalance() >= amountIn)
       {
           acc.withdraw(amountIn);
           return true; // indicate success
       }
       else
       {
           return false; // indicate failure
       }
   }

   // remove an account
   public boolean removeAccount(String accountNumberIn)
   {
       int index = search(accountNumberIn); // find index of account
       if(index != -999) // if account exists account
       {
           list.remove(index);
           return true; // remove was successful
       }
       else
       {
           return false; // remove was unsuccessful
       }
   }
}
```

As you can see, we have declared and initialized a single attribute, an `ArrayList` which will hold `BankAccount` objects.

```
ArrayList<BankAccount> list = new ArrayList<>();
```

Now the methods. Firstly the `search` method, which is declared as **`private`** because it is there only to assist other methods of the class, rather than to be accessed by other classes:

```
private int search(String accountNumberIn)
{
    for(int i = 0; i <= list.size() - 1; i++)
    {
        BankAccount tempAccount = list.get(i); // find the account at index i
        String tempNumber = tempAccount.getAccountNumber(); // get account number
        if(tempNumber.equals(accountNumberIn)) // if this is the account we are looking for
        {
            return i; // return the index
        }
    }
    return -999;
}
```

You have seen something like this before in Chap. 6 when we searched an integer array—you can see we are using the same technique of sending back a "dummy" value if the account number is not valid.

On each iteration of the loop the account at that index is retrieved using the `get` method of `ArrayList` and assigned to a `BankAccount` object, `tempAccount`. The account number of `tempAccount` is then assigned to a `String` variable `tempNumber`. This is compared with the account number that has been input. If the account number matches, the loop returns the index of that item and terminates. Otherwise, the loop continues to the end of the list (determined by using the `size` method of `ArrayList`). The method then returns the dummy value of −999, indicating that no item with that account number exists.

The next method simply returns the total number of items currently in the list, again using the `size` method of `ArrayList`:

```
public int getTotal()
{
    return list.size();
}
```

Next we have a method to retrieve an account with a particular account number:

```
public BankAccount getItem(String accountNumberIn)
{
    int index = search(accountNumberIn);
    if(index != -999) // check that account exists
    {
        return list.get(index);
    }
    else
    {
        return null; // no such account
    }
}
```

Here you can see how we utilize our search method—we use it to find the index of the account with the given account number, then we check that the index is not equal to −999 (in other words that the account exists), and as long it is a valid index we return the relevant account. If the index is not valid a **null** value is returned.

Now we come to the addAccount method:

```
public boolean addAccount(String accountNumberIn, String nameIn)
{
    if(search(accountNumberIn) == -999) // check that account does not already exist
    {
        list.add(new BankAccount(accountNumberIn, nameIn)); // add new account
        return true;
    }
    return false;
}
```

Once again we use the search method, this time to check that an account with this number does *not* already exist—so we are hoping that search returns a value −999. If this is the case we use the add method of ArrayList to add a new BankAccount object, which we create from the account number and account name that are received as parameters to the method—the method then returns **true**, indicating success.

Should the account already exist, a value of **false** is returned, indicating that no new account was added.

Now for the depositMoney method.

```
public boolean depositMoney(String accountNumberIn, double amountIn)
{
    BankAccount acc = getItem(accountNumberIn);
    if(acc != null)
    {
        acc.deposit(amountIn);
        return true; // indicate success
    }
    else
    {
        return false; // indicate failure
    }
}
```

The method receives the account number of the account in which we wish to place the money, and the amount to be deposited. We retrieve the correct account with the getItem method that we developed earlier. We check that this is not a **null** value, and if all is well we use the deposit method of BankAccount to deposit the money, and return a value of **true**, to indicate success. If the account returned was **null**, that indicates that there was no account with the account number in question, and in that case a value of **false** is returned.

The withdrawMoney method is similar except that we need to have an additional check in the **if** statement to see whether or not there were sufficient funds for the withdrawal to go ahead:

```
public boolean withdrawMoney(String accountNumberIn, double amountIn)
{
    BankAccount acc = getItem(accountNumberIn);
    if(acc != null && acc.getBalance() >= amountIn)
    {
        acc.withdraw(amountIn);
        return true; // indicate success
    }
    else
    {
        return false; // indicate failure
    }
}
```

As we mentioned above, the method could be improved if there were a way to determine whether the withdrawal was declined because there was no such account or because there were insufficient funds. This is left for the exercises at the end of the chapter.

Finally we have the method that removes an account:

```
public boolean removeAccount(String accountNumberIn)
{
    int index = search(accountNumberIn); // find index of account
    if(index != -999) // if account exists account
    {
        list.remove(index);
        return true; // remove was successful
    }
    else
    {
        return false; // remove was unsuccessful
    }
}
```

As you can see, we make use of the `remove` method of `ArrayList`, which removes an item at a particular index. The index is found by calling the `search` method as before, and as before we first test to make sure the account exists; if it does, then the account is removed and a value of **`true`** is returned—if not the method returns a value of **`false`**.

It is worth considering how much more complex it would have been to remove an item from the collection if we had used an array rather than an `ArrayList`. Since the array might be only partially full, we would have to introduce a variable to keep track of the position of the last item in the array. Then all the items following the one to be removed would have to be shuffled along by one position in order to overwrite the given item. Finally, we would need to make sure when any new item is added, it is added to the end of the reduced array, so we would have to reduce the 'end-of-array' variable by 1. That's rather a lot of work we don't need to do now we have used an `ArrayList` and can do all that with just one call to its `remove` method!

8.8.2 Testing the *Bank* Class

The program below, `BankApplication` uses the `Bank` class—notice that we are using our new `EasyScanner` class here.

BankApplication

```
public class BankApplication
{
    public static void main(String[] args)
    {
        char choice;

        Bank myBank = new Bank();

        // offer menu
        do
        {
            System.out.println();
            System.out.println("1. Create new account");
            System.out.println("2. Remove an account");
            System.out.println("3. Deposit money");
            System.out.println("4. Withdraw money");
            System.out.println("5. Check account details");
            System.out.println("6. Quit");
            System.out.println();
            System.out.print("Enter choice [1-6]: ");

            // get choice
            choice = EasyScanner.nextChar();
            System.out.println();

            // process menu options
            switch (choice)
            {
                case '1': option1(myBank);
                break;
                case '2': option2(myBank);
                break;
                case '3': option3(myBank);
                break;
                case '4': option4(myBank);
                break;
                case '5': option5(myBank);
                break;
                case '6': break;
                default: System.out.println("Invalid entry");
                }
            }
            while (choice != '6');
        }

        // add account
        static void option1(Bank bankIn)
        {
            // get details from user
            System.out.print("Enter account number: ");
            String number = EasyScanner.nextString();
            System.out.print("Enter account name: ");
            String name = EasyScanner.nextString();
            // add account to list
            boolean success = bankIn.addAccount(number, name);
            if(success)
            {
                System.out.println("Account added");
            }
            else
            {
                System.out.println("Account number already exists");
            }
        }
           // remove account
           static void option2(Bank bankIn)
           {
               // get account number of account to remove
               System.out.print("Enter account number: ");
               String number = EasyScanner.nextString();
               // delete item if it exists
               boolean found = bankIn.removeAccount(number);

               if (found)
               {
                   System.out.println("Account removed");
               }
               else
               {
               System.out.println("No such account number");
               }
```

```
        }

        // deposit money in an account
        static void option3(Bank bankIn)
        {
            // get details from user
            System.out.print("Enter account number: ");
            String number = EasyScanner.nextString();
            System.out.print("Enter amount to deposit: ");
            double amount = EasyScanner.nextDouble();

            boolean found = bankIn.depositMoney(number, amount);

            if(found)
            {
                System.out.println("Money deposited");
            }
            else
            {
             System.out.println("No such account");
            }

    }

    // withdraw money from an account
    static void option4(Bank bankIn)
    {
        // get details from user
        System.out.print("Enter account number: ");
        String number = EasyScanner.nextString();
        System.out.print("Enter amount to withdraw: ");
        double amount = EasyScanner.nextDouble();
        boolean ok = bankIn.withdrawMoney(number, amount);

        if(ok)
        {
            System.out.println("Withdrawal made");
        }
        else
        {
            System.out.println("No such account or insufficient funds");
        }

    }

    // check account details
    static void option5(Bank bankIn)
    {
        // get details from user
        System.out.print("Enter account number ");
        String number = EasyScanner.nextString();

        BankAccount account = bankIn.getItem(number);

        if(account != null)
        {
            System.out.println("Account number: " + account.getAccountNumber());
            System.out.println("Account name: " +account.getAccountName());
            System.out.println("Balance: " + account.getBalance());
            System.out.println();
        }
        else
        {
            System.out.println("No such account");
        }
    }
}
```

You are familiar with this sort of menu-driven program, so there is not too much to say about it, except to observe that this is probably the first example of an application which, although not all that complex, could actually be thought of as the kind of application that could be used in a real business environment. Of course, in the outside world such applications are much more sophisticated than this, but they are, in principle, not too different from the sort of thing we have just done. Notice that our application involves a number of classes that we have written ourselves, and have pulled together to form a single application.

It is worth drawing attention to the way that the program makes use of some of the features of the `Bank` class that we incorporated into the `BankApplication`. For example, in `option1` (and similarly in other methods) we make use of the fact that the `addAccount` method of `Bank` returns **`true`** if the new account was successfully added, and **`false`** otherwise:

```
boolean success = bankIn.addAccount(number, name);
if(success)
{
     System.out.println("Account added");
}
else
{
     System.out.println("Account number already exists");
}
```

In a similar way, in `option5`, we use the fact that the `getItem` method returns **`null`** if the account was not found:

```
if(account != null)
{
    System.out.println("Account number: " + account.getAccountNumber());
    System.out.println("Account name: " +account.getAccountName());
    System.out.println("Balance: " + account.getBalance());
    System.out.println();
}
else
{
    System.out.println("No such account");
}
```

We end this chapter with an example program run from `BankApplication`, followed by a few ideas on how our application could be improved.

```
1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1-6]: 1

Enter account number: 63488965
Enter account name: Mary Land-Cookies
Account added

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit
```

```
Enter choice [1–6]: 1

Enter account number: 98654322
Enter account name: Laura Norder
Account added

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 1

Enter account number: 12347890
Enter account name: Gary Baldi–Biscuits
Account added

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 3

Enter account number: 12347890
Enter amount to deposit: 1500
Money deposited

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 2

Enter account number: 98654322
Account removed
```

```
1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 5

Enter account number 55566777
No such account

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 5

Enter account number 12347890
Account number: 12347890
Account name: Gary Baldi–Biscuits
Balance: 1500.0

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 6
```

We should point out that for our application to be useful to any organization, it would need to be able to store the account information even after the application terminates. However, before you are able to achieve this you will have to wait until the second semester, where you will find out how to create files to hold permanent records.

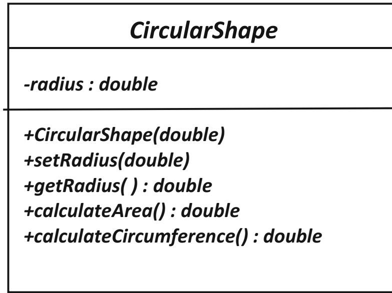
8.9 The Benefits of Object-Oriented Programming

In this chapter and the previous one you have seen how to create classes and use them as data types in your programs. You have seen how the process of building classes enables us to hide data within a class. Programming languages based on classes and objects—in other words object-oriented languages—have brought a number of benefits, and are now the standard. Below we have summarized some of the benefits that this has brought us.

- As we have demonstrated, the ability to encapsulate data within a class has enabled us to build far more secure systems.
- The object-oriented approach makes it far easier for us to *re-use* classes again and again. Having defined a `BankAccount` class or a `Student` class for example, we can use them in many different programs without having to write a new class each time. In the next chapter you will also see how it is possible to refine existing classes to meet additional needs by the technique known as **inheritance**. If systems can be assembled from re-usable objects, this leads to far higher productivity.
- With the object-oriented approach it is possible to define and use classes which are not yet complete. They can then be extended without upsetting the operation of other classes. This greatly improves the testing process. We can easily build prototypes without having to build a whole system before testing it and letting the user of the system see it.
- The object-oriented approach makes it far easier to make changes to systems once they have been completed. Whole classes can be replaced, or new classes can easily be added.
- The object-oriented way of doing things is a far more "natural" approach. We base our programs on objects that exist in the real world—students, bank accounts, customers and so on.
- The modular nature of object-oriented programming improves the whole development process. The modular approach means that the old methodologies whereby systems were first analysed, then designed, and then implemented and tested were able give way to new methods whereby these processed were far more integrated and systems were developed far more rapidly.

8.10 Self-test Questions

1. In question 7 of the programming exercises at the end of Chap. 2 you wrote a program that calculated the area and circumference of a circle. Now consider a class that we could develop for this purpose; we have called it `CircularShape`. Here is the UML design:

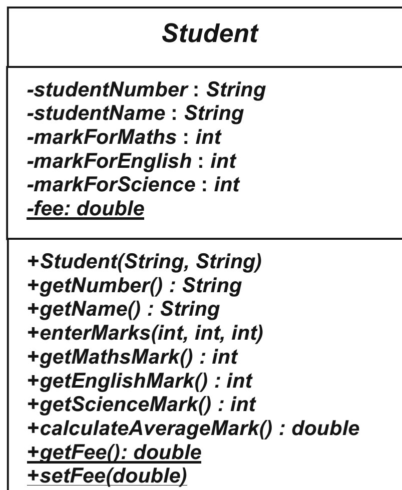


(a) Distinguish between *attributes* and *methods* in this class.
(b) Explain what it meant by the term *encapsulation*, how it is recorded in this UML diagram and how it is implemented in a Java class.
(c) For each method in the `CircularShape` class, determine

- the number of parameters;
- the type of any parameters;
- the return type;
- the equivalent method header in Java.

(d) Add an additional method into this UML diagram, `calculateDiameter`, which calculates and returns the diameter of the circle.
(e) Write the Java code for the `calculateDiameter` method.

2. The UML diagram below represents the design for a `Student` class.



You can see that students have a name, a number, some marks for subjects they are studying and the fee. Methods are then provided to process this data.

(a) What is indicated by the fact that certain attributes and methods have been underlined?
(b) Write the Java code for the parts of the class that have been underlined.

3. Consider the following class:

```
public class SomeClass
{
   private int x;

   public SomeClass( )
   {
      x = 10;
   }

   public SomeClass(int xIn)
   {
      x = xIn;
   }

   public void setX(int xIn)
   {
      x = xIn;
   }

   public int getX()
   {
      return x;
   }
}
```

(a) What would be the output from the following program?

```
public class Test1
{
   public static void main(String[] args)
   {
      SomeClass myObject = new SomeClass();
      System.out.println(myObject.getX());
   }
}
```

(b) What would be the output from the following program?

```
public class Test2
{
   public static void main(String[] args)
   {
      SomeClass myObject = new SomeClass(5);
      System.out.println(myObject.getX());
   }
}
```

(c) Explain why the following program would not compile.

```
public class Test3
{
   public static void main(String[] args)
   {
      SomeClass myObject = new SomeClass(5, 8);
      System.out.println(myObject.getX());
   }
}
```

(d) What would be the output from the following program?

```
public class Test4
{
   public static void main(String[] args)
   {
      int y = 20;
      SomeClass myObject = new SomeClass(5);
      System.out.println(myObject.getX());
      test(y, myObject);
      System.out.println(y);
      System.out.println(myObject.getX());
   }

   static void test(int z, SomeClass classIn)
   {
      z = 50;
      classIn.setX(100);
   }
}
```

4. Consider the `Bank` program from Sect. 8.8.1.

(a) Adapt the `withdrawMoney` method so that it distinguishes the two reasons why the method might fail—namely that there is no account with the given account number, or there is not enough money in the account to make a withdrawal.

A **`boolean`** method would no longer suffice as there is more then one possibility. One solution would be for the method to return an integer—perhaps 1 for success, −1 to indicate that the method failed because there was no such account number, and −2 to indicate that it failed because there were insufficient funds.

(b) Adapt the `BankApplication` program from Sect. 8.8.2 so that option 4 now uses the new version of `withdrawMoney`.

5. Identify some of the reasons why the object-oriented approach has become the norm for programming.

8.11 Programming Exercises

1. (a) Implement the `CircularShape` class that was discussed in self-test question 1 above.
   (b) Add the `calculateDiameter` method into this class as discussed in self-test question 1d and 1e above.
   (c) Write a program to test out your class. This program should allow the user to enter a value for the radius of the circle, and then display the area, circumference and diameter of this circle on the screen by calling the appropriate methods of the `CircularShape` class.
   (d) Modify the tester program above so that once the information has been displayed the user is able to reset the radius of the circle. The area, circumference and diameter of the circle should then be displayed again.

2. (a) Write the code for the `Student` class discussed in self-test question 2 above. You should note that in order to ensure that a **`double`** is returned from the `calculateAverageMark` method you should specifically divide the total of the three marks by 3.0 and not simply by 3 (look back at Chap. 2 to remind yourself why this is the case).

   Another thing to think about is what you choose for the initial values of the marks. If you chose to give each mark an initial value of zero, this could be ambiguous; a mark of zero could mean that the mark simply has not been entered—or it could mean the student actually scored zero in the subject! Can you think of a better initial value?

   You can assume that the fees for the student are set initially to 3000.
   (b) Write a tester class to test out your `Student` class; it should create two or three students (or even better an `ArrayList` of students), and use the methods of the `Student` class to test whether they work according to the specification.

3. A system is being developed for use in a store that sells electrical appliances. A class called `StockItem` is required for this system. An object of the `StockItem` class will require the following attributes:

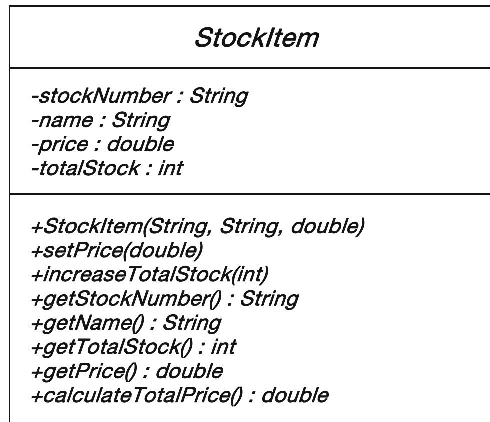
   - a stock number;
   - a name;
   - the price of the item;
   - the total number of these items currently in stock.

   The first three of the above attributes will need to be set at the time a `StockItem` object is created—the total number of items in stock will be set to zero at this time. The stock number and name will not need to be changed after the item is created.

The following methods are also required:

- a method that allows the price to be re-set during the object's lifetime;
- a method that receives an integer and adds this to the total number of items of this type in stock;
- a method that returns the total value of items of this type in stock; this is calculated by multiplying the price of the item by the number of items in stock;
- methods to read the values of all four attributes.

The design of the `StockItem` class is shown in the following UML diagram:



(a) Write the code for the `StockItem` class.

(b) Consider the following program, which uses the `StockItem` class, and in which some of the code has been replaced by comments:

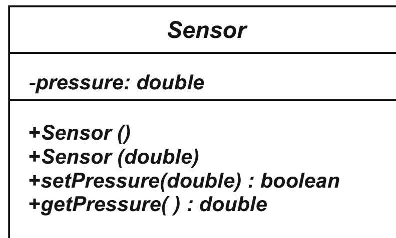
```
import  java.util.Scanner;
public class TestProg
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        Scanner keyboardString = new Scanner(System.in);
        String tempNumber;
        String tempName;
        double tempPrice;

        System.out.print("Enter the stock number: ");
        tempNumber = keyboardString.nextLine();
        System.out.print("Enter the name of the item: ");
        tempName = keyboardString.nextLine();
        System.out.print("Enter the price of the item: ");
        tempPrice = keyboard.nextDouble();

        // Create a new item of stock using the values that were entered by the user

        // Increase the total number of items in stock by 5

        // Display the stock number

        // Display the total price of all items in stock
    }
}
```

Replace the comments with appropriate code.

(c) i. A further attribute, `salesTax`, is required. The value of this attribute should always be the same for each object of the class. Write the declaration for this attribute.

ii. Provide a class method, `setSalesTax`, for this class—it should receive a **`double`** and set the value of the sales tax to this value.

iii. Write a line of code that sets the sales tax for all objects of the class to 10 without referring to any particular object.



4. The class shown below keeps track of a pressure sensor in a laboratory.

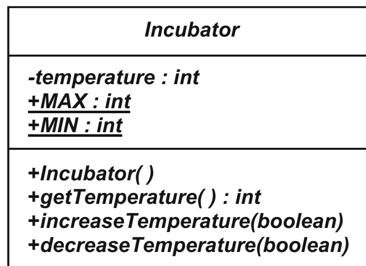
   When a `Sensor` object is created using the first constructor, the initial pressure is set to zero. When it is created using the second constructor it is set to the value of the parameter.

   The pressure should not be set to a value less than zero. Therefore, if the input parameter to the `setPressure` method is a negative number, the pressure should not be changed and a value of `false` should be returned. If the pressure is set successfully, a value of `true` should be returned.

   (a) Write the code for the `Sensor` class.

   (b) Develop a `SensorTester` program to test the `Sensor` class.

5. Consider a class that keeps track of the temperature within an incubator. The UML diagram is shown below:



   When an `Incubator` object is created, the temperature is initially set to 5°.

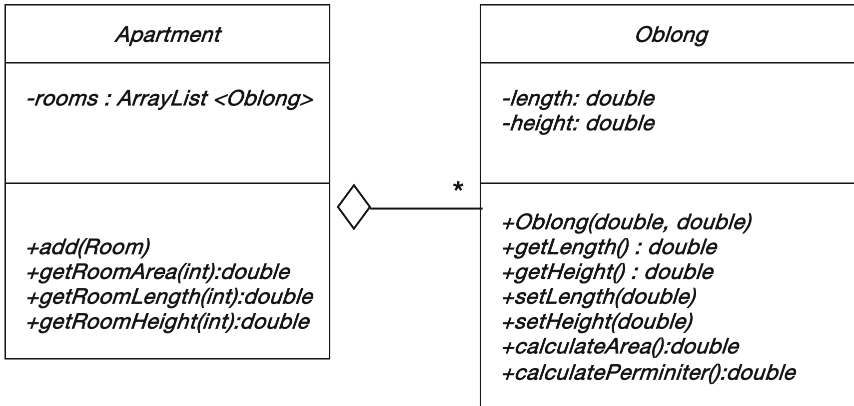
The `increaseTemp` method increases the temperature by 1, and the `decreaseTemp` method decreases the temperature by 1. However, the temperature must never be allowed to rise above a maximum value of 10 nor fall below a minimum value of −10. If an attempt is made to increase or decrease the temperature so it falls outside this range, then an alarm must be raised; the methods in this case should not increase or decrease the temperature but should return a value of **`false`**, indicating that the alarm should be raised. If the temperature is changed successfully, however, a value of **`true`** is returned.

(a) Write the code for the `Incubatotor` class.

(b) Develop a `IncubatorTester` program to test the `Incubator` class.

6. Implement the changes to the `Bank` class and the `BankApplication` program suggested in question 4 of the self-test questions. The source code for the `Bank` class and the `BankApplication` class can be downloaded from the website.

7. (a) In programming Exercise 6 of the last chapter you were asked to develop a program to process a collection of rooms in an apartment. Now consider a collection class, `Apartment`, for this purpose. The `Apartment` class would store a collection of `Oblong` objects, where each `Oblong` object represents a particular room in the apartment. The UML diagram depicting the association between the `Apartment` class and the `Oblong` class is shown below:



The single attribute of the `Apartment` class consists of a collection of `Oblong` objects, `rooms`, which makes use of an `ArrayList`.

The methods of the `Apartment` class are described below:

+add(Room) : boolean
Adds the given room to the list of `rooms`.

+getRoomArea(int) : double
Returns the area of the given room number sent in as a parameter. If an invalid room number is sent in as a parameter this method should send back some dummy value (for example −999).

+getRoomLength(int) : double
Returns the length of the given room number sent in as a parameter. If an invalid room number is sent in as a parameter this method should send back some dummy value (for example −999).

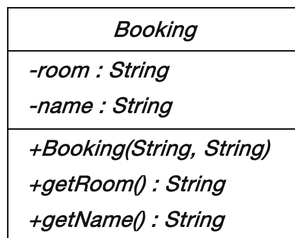
+getRoomHeight(int) : double
Returns the height of the given room number sent in as a parameter. If an invalid room number is sent in as a parameter this method should send back some dummy value (for example −999).

Implement the `Apartment` class.

(b) Develop an `ApartmentTester` program to test the `Apartment` class.

8. Consider a scenario in which a university allows lecturers to borrow equipment. The equipment is available for use 5 days a week and for 7 periods during each day. When the equipment is booked for use, the details of the booking (room number and lecturer name) are recorded. When no booking is recorded, the equipment is available for use.

(a) Create a `Booking` class defined in the UML diagram below:



(b) Now a `TimeTable` class is defined to process these bookings. Its UML diagram is given below:

TimeTable
-times: Booking[][]
+TimeTable(int, int) *+makeBooking(int, int, Booking) : boolean* *+cancelBooking(int, int) : boolean* *+isFree(int, int) : boolean* *+getBooking(int, int) : Booking* *+numberOfDays() : int* *+numberOfPeriods() : int*

As you can see, the attribute of this class is a two-dimensional array of `Booking` objects. The methods of this class are defined below:

+TimeTable(int, int)
A constructor that accepts the number of days per week and number of periods per day and sizes the timetable accordingly.

You should note that initially all elements in the array will of course have a **`null`** value—*a* **`null`** *value will represent an empty slot.*

+makeBooking(int, int, Booking) : boolean
Accepts the booking details for a particular day and period and, as long as this slot is not previously booked and the day and period numbers are valid, updates the timetable accordingly. Returns **`true`** if the booking was recorded successfully and **`false`** if not.

+cancelBooking(int, int) : boolean
Cancels the booking details for a particular day and period. Returns **`false`** if the given slot was not previously booked or the day and period number are invalid, and **`true`** if the slot was cancelled successfully.

+isFree(int, int) : boolean
Accepts a day and period number and returns **`true`** if the day and period numbers are valid and the given slot is free, and **`false`** otherwise.

+getBooking(int, int) : Booking
Accepts a day and period number and returns the booking for the given slot if the day and period number are valid and the slot has been booked or **`null`** otherwise.

+numberOfDays() : int
Returns the number of days associated with this timetable.

+numberOfPeriods() : int
Returns the number of periods associated with this timetable.
Implement this class in Java.

(c) Write a suitable tester for this class.

9. Add some additional methods such as `nextByte` and `nextLong` to the `EasyScanner` class.

Check for updates

9 Inheritance

Outcomes:

By the end of this chapter you should be able to:

- *explain the term* ***inheritance***;
- *design inheritance structures using UML notation*;
- *implement inheritance relationships in Java*;
- *distinguish between* ***method overriding*** *and* ***method overloading***;
- *explain the term* ***type cast*** *and implement this in Java*;
- *explain the use of the* **`abstract`** *modifier when applied to classes and methods*;
- *explain the use of the* **`final`** *modifier, when applied to classes and methods*;
- *describe the way in which all Java classes are derived from the* `Object` *class*.

9.1 Introduction

One of the greatest benefits of the object-oriented approach to software development is that it offers the opportunity for us to *reuse* classes that have already been written—either by ourselves or by someone else. Let's look at a possible scenario. Say you wanted to develop a software system and you have, during your analysis, identified the need for a class called `Employee`. You might be aware that a colleague in your organization has already written an `Employee` class; rather than having to write your own class, it would be easier to approach your colleague and ask her to let you use her `Employee` class.

So far so good, but what if the `Employee` class that you are given doesn't quite do everything that you had hoped? Perhaps your employees are part-time employees, and you want your class to have an attribute like `hourlyPay`, or

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_9

methods like `calculateWeeklyPay` and `setHourlyPay`, and these attributes and methods do not exist in the `Employee` class you have been given.

You may think it would be necessary to go into the old class and start messing about with the code. But there is no need, because object-oriented programming languages provide the ability to extend existing classes by adding attributes and methods to them. This is called **inheritance**.

9.2 Defining Inheritance

Inheritance is the sharing of attributes and methods among classes. We take a class, and then define other classes based on the first one. The new classes *inherit* all the attributes and methods of the first one, but also have attributes and methods of their own. Let's try to understand this by thinking about the `Employee` class.

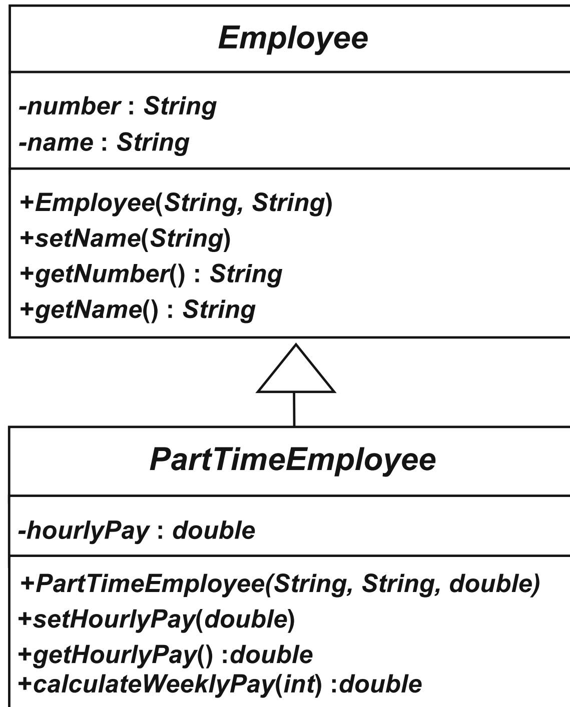


Fig. 9.1 An inheritance relationship

Say our Employee class has two attributes, number and name, a user-defined constructor, and some basic get- and set-methods for the attributes. We now define our PartTimeEmployee class; this class will *inherit* these attributes and methods, but can also have attributes and methods of its own. We will give it one additional attribute, hourlyPay, some methods to access this attribute and one additional method, calculateWeeklyPay.

This is illustrated in Fig. 9.1 which uses the UML notation for inheritance, namely a triangle.

You can see from this diagram that an inheritance relationship is a *hierarchical* relationship. The class at the top of the hierarchy—in this case the Employee class —is referred to as the **superclass** (or **base class**) and the PartTimeEmployee as the **subclass** (or **derived class**).

The inheritance relationship is also often referred to as an *is-a-kind-of* relationship; in this case a PartTimeEmployee *is a kind of* Employee.

9.3 Implementing Inheritance in Java

The code for the Employee class is shown below:

Employee

```
public class Employee
{
    private String number;
    private String name;
    public Employee(String numberIn, String nameIn)
    {
        number = numberIn;
        name = nameIn;
    }

    public void setName(String nameIn)
    {
        name = nameIn;
    }

    public String getNumber()
    {
        return number;
    }

    public String getName()
    {
        return name;
    }
}
```

There is nothing new here, so let's get on with our PartTimeEmployee class. We will present the code first and analyse it afterwards.

PartTimeEmployee

```
public class PartTimeEmployee extends Employee // this class is a subclass of Employee
{
    private double hourlyPay; // this attribute is unique to the subclass

    // the constructor
    public PartTimeEmployee(String numberIn, String nameIn, double hourlyPayIn)
    {
        super(numberIn, nameIn); // call the constructor of the superclass
        hourlyPay = hourlyPayIn;
    }

    // these methods are also unique to the subclass
    public double getHourlyPay()
    {
        return hourlyPay;
    }

    public void setHourlyPay(double hourlyPayIn)
    {
        hourlyPay = hourlyPayIn;
    }

    public double calculateWeeklyPay(int noOfHoursIn)
    {
        return noOfHoursIn * hourlyPay;
    }
}
```

The first line of interest is the class header itself:

```
public class PartTimeEmployee extends Employee // this class is a subclass of Employee
```

Here we see the use of the keyword **extends**. Using this word in this way means that the PartTimeEmployee class (the *subclass*) inherits all the attributes and methods of the Employee class (the *superclass*). So although we haven't coded them, any object of the PartTimeEmployee class will have, for example, an attribute called name and a method called getNumber. A PartTimeEmployee is now a *kind of* Employee.

But can you see a problem here? The attributes have been declared as **private** in the superclass so although they are now part of our PartTimeEmployee class, none of the PartTimeEmployee class methods can directly access them—the subclass has only the same access rights as any other class!

There are a number of possible ways around this:

1. We could declare the original attributes as **public**—but this would take away the whole point of encapsulation.
2. We could use the special keyword **protected** instead of **private**. The effect of this is that anything declared as **protected** is accessible to the methods of any subclasses. There are, however, two issues to think about here. The first is that you have to anticipate in advance when you want your class to be able to be inherited. The second problem is that it weakens your efforts to encapsulate information within the class, since, in Java, **protected** attributes are also accessible to any other class in the same package (you will find out much more about the meaning of the word **package** in Chap. 19).

The above remarks notwithstanding, this is a perfectly acceptable approach to use, particularly in situations where you are writing a class as part of a discrete application, and you will be aware in advance that certain classes will need to be subclassed. You will see an example of this in Sect. 9.5.
Incidentally, in a UML diagram a **protected** attribute is indicated by a hash symbol, #.

3. The other solution, and the one we will use now, is to leave the attributes as **private**, but to plan carefully in advance which get- and set-methods we are going to provide.

After the class header we have the following declaration:

```
private double hourlyPay;
```

This declares an attribute, hourlyPay, which is unique to our subclass—but remember that the attributes of the superclass, Employee, will be inherited, so in fact any PartTimeEmployee object will have *three* attributes.

Next comes the constructor. We want to be able to assign values to the number and name at the time that the object is created, just as we do with an Employee object; so our constructor will need to receive parameters that will be assigned to the number and name attributes.

But wait a minute! How are we going to do this? The number and name attributes have been declared as **private** in the superclass—so they aren't accessible to objects of the subclass. Luckily there is a way around this problem. We can call the constructor of the superclass by using the keyword **super**. Look how this is done:

```
public PartTimeEmployee(String numberIn, String nameIn, double hourlyPayIn)
{
   super(numberIn, nameIn); // call the constructor of the superclass
   hourlyPay = hourlyPayIn;
}
```

After calling the constructor of the superclass, we need to perform one more task —namely to assign the third parameter, hourlyPayIn, to the hourlyPay attribute. Notice, however, that the line that calls **super** has to be the first one—if we had written our constructor like this it would not compile:

```
/* This version of the constructor would not compile - the call to super has to be the
   first instruction */

public PartTimeEmployee(String numberIn, String nameIn, double hourlyPayIn)
{
        hourlyPay = hourlyPayIn;
        super(numberIn, nameIn); // this call should have been the first instruction!
}
```

The remaining methods of `PartTimeEmployee` are new methods specific to the subclass:

```
public double getHourlyPay()
{
   return hourlyPay;
}
public void setHourlyPay(double hourlyPayIn)
{
   hourlyPay = hourlyPayIn;
}
public double calculateWeeklyPay(int noOfHoursIn)
{
   return noOfHoursIn * hourlyPay;
}
```

The first two provide read and write access respectively to the `hourlyPay` attribute. The third one receives the number of hours worked and calculates the pay by multiplying this by the hourly rate. Th program below demonstrates the use of the `PartTimeEmployee` class.

PartTimeEmployeeTester

```
import java.util.Scanner;
public class PartTimeEmployeeTester
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        Scanner keyboardString = new Scanner(System.in);
        String number, name;
        double pay;
        int hours;
        PartTimeEmployee emp;

        // get the details from the user
        System.out.print("Employee Number? ");
        number = keyboardString.nextLine();
        System.out.print("Employee's Name? ");
        name = keyboardString.nextLine();
        System.out.print("Hourly Pay? ");
        pay = keyboard.nextDouble();
        System.out.print("Hours worked this week? ");
        hours = keyboard.nextInt();

        // create a new part-time employee
        emp = new PartTimeEmployee(number, name, pay);

        // display part-time employee's details, including the weekly pay
        System.out.println();

        // the next two methods have been inhreted from the Employee class
        System.out.println(emp.getName());
        System.out.println(emp.getNumber());

        System.out.println(emp.calculateWeeklyPay(hours));
    }
}
```

Here is a sample test run:

Employee Number? ***A103456***
Employee's Name? ***Mandy Lifeboats***
Hourly Pay? ***15.50***
Hours worked this week? ***20***

Mandy Lifeboats
A103456
310.0

We can now move on to look at another inheritance example; let's choose the `Oblong` class that we developed in the last chapter.

9.4 Extending the *Oblong* Class

We are going to define a new class called `ExtendedOblong`, which extends the `Oblong` class. First, let's remind ourselves of the `Oblong` class itself.

The *Oblong* class – a reminder

```
public class Oblong
{
   // the attributes
   private double length;
   private double height;

   // the methods

   // the constructor
   public Oblong(double lengthIn, double heightIn)
   {
      length = lengthIn;
      height = heightIn;
   }

   // this method allows us to read the length attribute
   public double getLength()
   {
      return length;
   }

   // this method allows us to read the height attribute
   public double getHeight()
   {
      return height;
   }

   // this method allows us to write to the length attribute
   public void setLength(double lengthIn)
   {
      length = lengthIn;
   }

   // this method allows us to write to the height attribute
   public void setHeight(double heightIn)
   {
      height = heightIn;
   }

   // this method returns the area of the Oblong
   public double calculateArea()
   {
      return length * height;
   }

   // this method returns the perimeter of the Oblong
   public double calculatePerimeter()
   {
      return 2 * (length + height);
   }
}
```

The original Oblong class had the capability of reporting on the perimeter and area of the oblong. Our extended class will have the capability of sending back a string representation of itself composed of a number of symbols such as asterisks—for example:

```
*****
*****
*****
```

Now at first glance you might think that this isn't a string at all, because it consists of several lines. But if we think of the instruction to start a new line as just another character—which for convenience we could call <NEWLINE>—then our string could be written like this.

```
*****<NEWLINE>*****<NEWLINE>*****
```

In Java we are able to represent this <NEWLINE> character with a special character that looks like this:

```
'\n'
```

This is one of a number of special characters called **escape characters**, which are always introduced by a backslash (\). Another useful escape character is '\t' which inserts a tab.[1]

Our ExtendedOblong class will need an additional attribute, which we will call symbol, to hold the character that is to be used to draw the oblong. We will also provide a setSymbol method, and of course we will need a method that sends back the string representation. We will call this method draw. The new constructor will accept values for the length and height as before, but will also receive the character to be used for drawing the oblong.

The design is shown in Fig. 9.2.

Now for the implementation. As well as those aspects of the code that relate to inheritance, there is an additional new technique used in this class—this is the technique known as **type casting**. Take a look at the complete code first—then we can discuss this new concept along with some other important features of the class.

[1]You would also have to place a backslash in front of a double quote (\"), a single quote (\') or another backslash (\\) if you wanted any of these to be output as part of a string. This is because the compiler would interpret these as having a special meaning such as terminating the string.

ExtendedOblong

```
public class ExtendedOblong extends Oblong
{
    private char symbol;

    // the constructor
    public ExtendedOblong(double lengthIn, double heightIn, char symbolIn)
    {
        super(lengthIn, heightIn);
        symbol = symbolIn;
    }

    public void setSymbol(char symbolIn)
    {
        symbol = symbolIn;
    }

    public String draw()
    {
        String s = new String(); // start off with an empty string
        int l, h;

        /* in the next two lines we type cast from double to integer so that we are able to count how
        many times we print the symbol */
        l = (int) getLength();
        h = (int) getHeight();
        for (int i = 1; i <= h; i++)
        {
            for (int j = 1; j <= l; j++)
            {
                s = s + symbol; // add the symbol to the string
            }
            s = s + '\n'; // add the <NEWLINE> character
        }
        return s; // return the string representation
    }
}
```

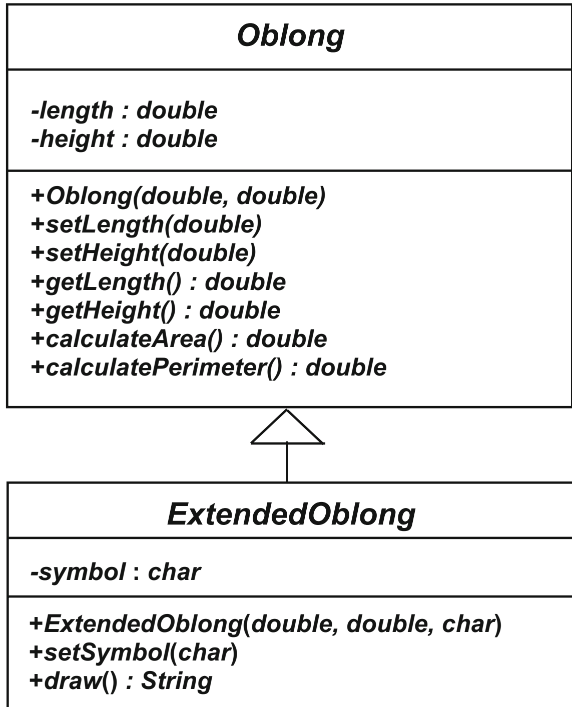


Fig. 9.2 The *Oblong* hierarchy

So let's take a closer look at all this. After the class header—which **extend**s the Oblong class—we declare the additional attribute, symbol, and then define our constructor:

```
public ExtendedOblong(double lengthIn, doubleheightIn, char symbolIn)
{
    super(lengthIn, heightIn);
    symbol = symbolIn;
}
```

Once again we call the constructor of the superclass with the keyword **super**. After the constructor comes the setSymbol method—which allows the symbol to be changed during the oblong's lifetime—and then we have the draw method, which introduces the new concept of **type casting**:

```
public String draw()
{
    String s = new String(); // start off with an empty string
    int l, h;
    l = (int) getLength();
    h = (int) getHeight();
    for (int i = 1; i <= h; i++)
    {
        for (int j = 1; j <= l; j++)
        {
            s = s + symbol; // add a symbol to end of the string
        }
        s = s + '\n'; // add a new line to the string
    }
    return s;
}
```

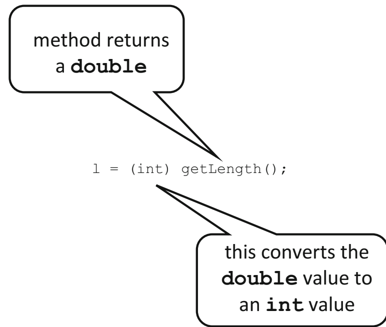
Inspect the code carefully—notice that we have declared two local variables of type **int**. In order to understand the purpose of these two variables, l and h, we need to explore this business of type casting, which means forcing an item to change from one type to another.

The draw method is going to create a string of one or more rows of stars or crosses or whatever symbol is chosen. Now the dimensions of the oblong are defined as **double**s. Clearly our draw method needs to be dealing with whole numbers of rows and columns—so we must convert the length and height of the oblong from **double**s to **int**s. There will obviously be some loss of precision here, but that won't matter in this particular case.

As you can see from the above code, type casting is achieved by placing the new type name in brackets before the item you wish to change. This is illustrated in Fig. 9.3.

The following program uses the ExtendedOblong class. It creates an oblong of length 10 and height 5, with an asterisk as the symbol; it then draws the oblong, changes the symbol to a cross, and draws it again.

Fig. 9.3 Type casting



ExtendedOblongTester

```
public class ExtendedOblongTester
{
    public static void main(String[] args)
    {
        ExtendedOblong extOblong = new ExtendedOblong(10.2,5.3,'*');
        System.out.println(extOblong.draw());
        extOblong.setSymbol('+');
        System.out.println(extOblong.draw());
    }
}
```

The output from this program is shown below:

```
**********
**********
**********
**********
**********

++++++++++
++++++++++
++++++++++
++++++++++
++++++++++
```

9.5 Method Overriding

In Chap. 5 you were introduced to the concept of polymorphism—the idea that we can have different methods and operators with the same name, but whose behaviour is different. You saw in that chapter that one way of achieving polymorphism was by method *overloading*, which involves methods of the same class having the same name, but being distinguished by their parameter lists.

Now we are going to explore another way of achieving polymorphism, namely by **method overriding**. In order to do this we are going to extend the BankAccount class that we developed in the previous chapter. You will recall that the class we developed there did not provide any overdraft facility—the withdraw method was designed so that the withdrawal would take place only if the amount to be withdrawn did not exceed the balance.

Now let's consider a special account which is the same as the original account, but allows holders of the account to be given an overdraft limit and to withdraw funds up to this limit. We will call this account GoldAccount. Since a GoldAccount *is a kind of* BankAccount, we can use inheritance here to design the GoldAccount class. In addition to the attributes of a BankAccount, a GoldAccount will need to have an attribute to represent the overdraft limit, and should have get- and set-methods for this attribute. As far as the methods are concerned, we need to reconsider the withdraw method. This will differ from the original method, because, instead of checking that the amount to be withdrawn does not exceed the balance, it will now check that the amount does not exceed the total of the balance plus the overdraft limit. So what we are going to do is to re-write—or *override*—the withdraw method in the subclass.

The UML diagram for the BankAccount class and the GoldAccount class appear in Fig. 9.4. You will notice that we have made a small change to the original BankAccount class. The balance attribute has a hash sign (#) in front of it instead of a minus sign. You will remember from our previous discussion that this means access to the attribute is **protected**, rather than **private**. The reason why we have decided to make this change is explained below.

You will also notice that the withdraw method appears in both classes—this, of course, is because we are going to override it in the subclass.

You might already be thinking about how to code the withdraw method in the GoldAccount class. If you are doing this, you will probably have worked out that this method is going to need access to the balance attribute, which of course was declared as **private** in the BankAccount class, and (for good reason) was not provided with a set-method.

When we developed the BankAccount class in Chap. 8, we developed it as a stand-alone class, and we didn't think about how it might be used in a larger application where it could be refined. Had we known about inheritance at that point we might have given the matter a little more thought, and realised that it would be useful if any sub-classes of BankAccount that were developed in the future had access to the balance attribute. As we explained in Sect. 9.3, we can achieve that by declaring that attribute as **protected** instead of **private**. That is what we have done here. The version of BankAccount that we are going to use in this

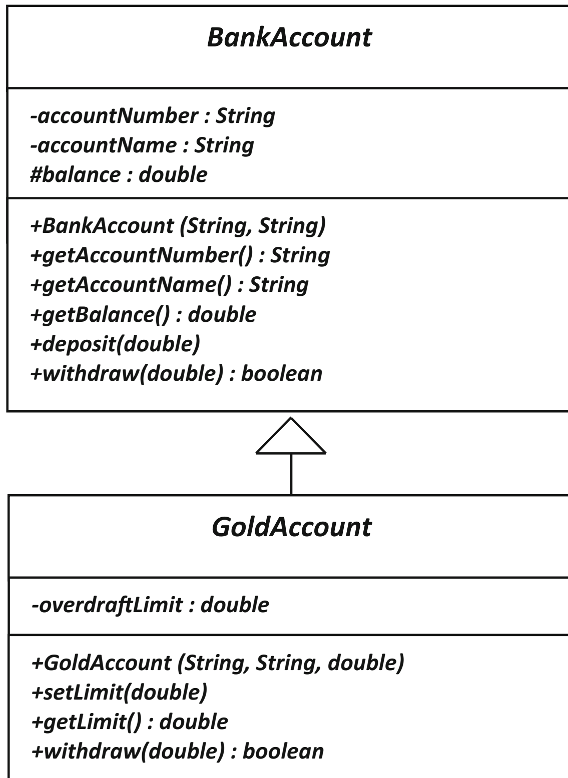


Fig. 9.4 The UML diagram for the *BankAccount* hierarchy

chapter is therefore exactly the same as the previous one, with the single difference that the declaration of the `balance` attribute now looks like this, with the keyword **`protected`** replacing **`private`**:

```
protected double balance;
```

This new version of the `BankAccount` class is available on the website.

Here is the code for the `GoldAccount` class you will notice that there is something new here, namely the line that reads `@Override`—have a look at the code, then we will explain this.

GoldAccount

```
public class GoldAccount extends BankAccount
{
    private double overdraftLimit;

    public GoldAccount(String numberIn, String nameIn, double limitIn)
    {
        super(numberIn, nameIn);
        overdraftLimit = limitIn;
    }

    public void setLimit(double limitIn)
    {
        overdraftLimit = limitIn;
    }


    public double getLimit()
    {
        return overdraftLimit;
    }

    @Override
    public boolean withdraw(double amountIn)
    {
        if(amountIn > balance + overdraftLimit) // the customer can withdraw up to the overdraft limit
        {
            return false; // no withdrawal was made
        }
        else
        {
            balance = balance - amountIn; // balance is protected so we have direct access to it
            return true; // money was withdrawn successfully
        }
    }
}
```

The thing that we are interested in here is the `withdraw` method. As we have pointed out this is introduced with **`@Override`**. This is an example of a Java **annotation**. Annotations begin with the `@` symbol, and always start with an upper case letter. Although it is not mandatory that we include this annotation, it is very good practice to do so. Its purpose is to inform the compiler that we are overriding a method from the superclass. This helps us to avoid making the common error of not giving the overridden method exactly the same name and parameter list as the method it is supposed to be overriding. Without the annotation, this would escape the notice of the compiler, and you would have simply written a new method. But with the annotation included you would get a compile error if the method headings did not match.

As far as the method itself is concerned, the test in the **`if`** statement differs from the original method in the `BankAccount` class (as shown below), in order to take account of the fact that customers with a gold account are allowed an overdraft:

`withdraw` method in `BankAccount` class

```
public boolean withdraw(double amountIn)
{
    if(amountIn > balance)
    {
        return false;
    }
    else
    {
        balance = balance - amountIn;
        return true;
    }
}
```

`withdraw` method in `GoldAccount` class

```
public boolean withdraw(double amountIn)
{
    if(amountIn > balance + overdraftLimit)
    {
        return false;
    }
    else
    {
        balance = balance - amountIn;
        return true;
    }
}
```

When we dealt with method *overloading* in Chap. 5 we told you that the methods with the same name *within* a class are distinguished by their parameter lists. In the case of method *overriding*, the methods have the same parameter list but belong to different classes—the superclass and the subclass. In this case they are distinguished by the *object with which they are associated*. We illustrate this in the program below.

OverridingDemo

```
public class OverridingDemo
{
   public static void main(String[] args)
   {
      boolean ok;
      //declare a BankAccount object
      BankAccount bankAcc = new BankAccount("123", "Ordinary Account Holder");
      //declare a GoldAccount object
      GoldAccount goldAcc = new GoldAccount("124", "Gold Account Holder", 500);

      bankAcc.deposit(1000);
      goldAcc.deposit(1000);

      ok = bankAcc.withdraw(1250); // the withdraw method of BankAccount is called
      if(ok)
      {
         System.out.print("Money withdrawn. ");
      }
      else
      {
         System.out.print("Insufficient funds. ");
      }
      System.out.println("Balance of " + bankAcc.getAccountName() + " is " + bankAcc.getBalance());
      System.out.println();

      ok = goldAcc.withdraw(1250); // the withdraw method of GoldAccount is called
      if(ok)
      {
         System.out.print("Money withdrawn. ");
      }
      else
      {
         System.out.print("Insufficient funds. ");
      }
      System.out.println("Balance of " + goldAcc.getAccountName() + " is " + goldAcc.getBalance());
      System.out.println();
   }
}
```

In this program we create an object of the `BankAccount` class and an object of the `GoldAccount` class (with an overdraft limit of 500), and deposit an amount of 1000 in each:

```
BankAccount bankAcc = new BankAccount("123", "Ordinary Account Holder");
GoldAccount goldAcc = new GoldAccount("124", "Gold Account Holder", 500);
bankAcc.deposit(1000);
goldAcc.deposit(1000);
```

Next we attempt to withdraw the sum of 1250 from the `BankAccount` object and assign the return value to a **`boolean`** variable, `ok`:

```
ok = bankAcc.withdraw(1250);
```

The `withdraw` method that is called here will be that of `BankAccount`, because it is called via the `BankAccount` object, `bankAcc`.

Once this is done we display a message showing whether or not the withdrawal was successful, followed by the balance of that account:

```
if(ok)
{
   System.out.print("Money withdrawn. ");
}
else
{
   System.out.print("Insufficient funds. ");
}
System.out.println("Balance of " + bankAcc.getAccountName() + " is " + bankAcc.getBalance());
```

Now the `withdraw` method is called again, but in this case via the `GoldAccount` object, `goldAcc`:

```
ok = goldAcc.withdraw(1250);
```

This time it is the `withdraw` method of `GoldAccount` that will be called, because `goldAcc` is an object of this class. The appropriate message and the balance are again displayed.

The output from this program is shown below:

Insufficient funds. Balance of Ordinary Account Holder is1000.0

Money withdrawn. Balance of Gold Account Holder is -250.0

As we would expect, the withdrawal from `BankAccount` does not take place—the balance is 1000, and since there is no overdraft facility a request to withdraw 1250 is denied.

In the case of the `GoldAccount`, however, a withdrawal of 1250 would result in a negative balance of 250, which is allowed, because it is within the overdraft limit of 500.

9.6 Abstract Classes

Let's think again about our `Employee` class. Imagine that our business expands, and we now employ full-time employees as well as part-time employees. A full-time employee object, rather than having an hourly rate of pay, will have an

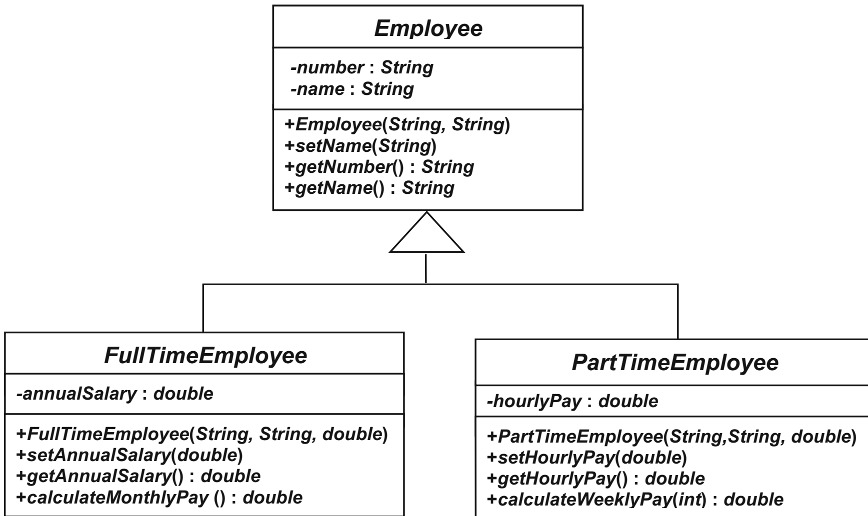


Fig. 9.5 An inheritance relationship showing the superclass *Employee* and the subclasses *FullTimeEmployee* and *PartTimeEmployee*

annual salary. It might also need a method that calculates the monthly pay (by dividing the annual salary by 12).

Figure 9.5 shows the structure of an employee hierarchy with the two types of employee, the full-time and the part-time employee.

Notice how the two subclasses contain the attributes and methods appropriate to the class. If you think about this a bit more, it will occur to you that *any* employee will always be either a full-time employee or a part-time employee. There is never going to be a situation in which an individual is just a plain old employee! So users of a program that included all these classes would never find themselves creating objects of the `Employee` class. In fact, it would be a good idea to prevent people from doing this—and, as you might have guessed, there is a way to do so, which is to declare the class as **`abstract`**. Once a class has been declared in this way it means that you are not allowed to create objects of that class. In order to make our employee class abstract all we have to do is to place the keyword **`abstract`** in the header:

```
public abstract class Employee
```

The `Employee` class simply acts a basis on which to build other classes. Now, if you tried to create an object of the `Employee` class you would get a compiler error.

We have already seen the code for `Employee` and `PartTimeEmployee`; Here is the code for the `FullTimeEmployee` class:

FullTimeEmployee

```
public class FullTimeEmployee extends Employee
{
    private double annualSalary;

    public FullTimeEmployee(String numberIn, String nameIn, double salaryIn)
    {
        super(numberIn,nameIn);
        annualSalary = salaryIn;
    }

    public void setAnnualSalary(double salaryIn)
    {
        annualSalary = salaryIn;
    }

    public double getAnnualSalary()
    {
        return annualSalary;
    }

    public double calculateMonthlyPay()
    {
        return annualSalary/12;
    }
}
```

As we said before, an inheritance relationship is often referred to as an "*is-a-kind-of*" relationship. A full-time employee is a *kind of* employee, as is a part-time employee. Therefore an object that is of type `PartTimeEmployee` is also of type `Employee`—an object is the type of its class, and also of any of the superclasses in the hierarchy.

Let's see how this relationship works in a Java program. Imagine a method which is set up to receive an `Employee` object. If we call that method and send in a `FullTimeEmployee` object or a `PartTimeEmployee` object, either is absolutely fine—because both are *kinds of* `Employee`. We demonstrate this in the program that follows:

EmployeeTester

```
public class EmployeeTester
{
    public static void main(String[] args)
    {
        FullTimeEmployee fte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
        PartTimeEmployee pte = new PartTimeEmployee("B456", "Mr Part-Time",30);
        testMethod(fte); // call testMethod with a full-time employee object
        testMethod(pte); // call testMethod with a part-time employee object
    }

    static void testMethod(Employee employeeIn) // the method expects to receive an Employee object
    {
        System.out.println(employeeIn.getName());
    }
}
```

In this program `testMethod` expects to receive an `Employee` object. It calls the `getName` method of `Employee` in order to display the employee's name.

In the main method, we create two objects, one FullTimeEmployee and one PartTimeEmployee:

```
FullTimeEmployee fte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
PartTimeEmployee pte = new PartTimeEmployee("B456", "Mr Part-Time",30);
```

We then call testMethod twice—first with FullTimeEmployee object and then with the PartTimeEmployee object:

```
testMethod(fte); // call testMethod with a full-time employee object
testMethod(pte); // call testMethod with a part-time employee object
```

The method accepts either object, and calls the getName method. The output is, as expected:

```
Ms Full-Time
Mr Part-Time
```

9.7 Abstract Methods

In the last program we conveniently gave our objects the names "Ms Full-Time" and "Mr Part-Time" so that we could easily identify them in our output. In fact, it wouldn't be a bad idea—particularly for testing purposes—if every Employee type actually had a method that returned a string telling us the kind of object we were dealing with. Adding such a method—we could call it getStatus—would be simple. For the FullTimeEmployee the method would look like this:

```
@Override
public String getStatus()
{
   return "Full-Time";
}
```

Notice that we have included the **@Override** annotation, even though it is not compulsory to do so.

For the `PartTimeEmployee`, `getStatus` would look like this:

```
@Override
public String getStatus()
{
    return "Part-Time";
}
```

It would be very useful if we could say to anyone using any of the `Employee` types, that we *guarantee* that this class will have a `getStatus` method. That way, a developer could, for example, write a method that accepts an `Employee` object, and call that object's `getStatus` method, even without knowing anything else about the class.

As you have probably guessed, we *can* guarantee it! What we have to do is to write an **abstract** method in the superclass—in this case `Employee`. Declaring a method as **abstract** means that any subclass is *forced* to override it—otherwise there would be a compiler error. So in this case we just have to add the following line into the `Employee` class:

```
public abstract String getStatus();
```

You can see that to declare an **abstract** method, we use the Java keyword **abstract**, and we define the header, but no body—the actual implementation is left to the individual subclasses. Of course, **abstract** methods can only be declared in **abstract** classes—it wouldn't make much sense to try to declare an object if one or more of its methods were undefined.

Now, having defined the **abstract** `getStatus` method in the `Employee` class, if we tried to compile the `FullTimeEmployee` or the `PartTimeEmployee` class (or any other class that extends `Employee`) without including a `getStatus` method we would be unsuccessful.

Once we have added the different `getStatus` methods into the `Employee` classes, we could re-write our `EmployeeTester` program from the previous sections using the `getStatus` method in `testMethod`. We have done this with `EmployerTester2` below:

EmployeeTester2

```
public class EmployeeTester2
{
    public static void main(String[] args)
    {
        FullTimeEmployee fte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
        PartTimeEmployee pte = new PartTimeEmployee("B456", "Mr Part-Time",30);
        testMethod(fte); // call testMethod with a full-time employee object
        testMethod(pte); // call testMethod with a part-time employee object
    }

    static void testMethod(Employee employeeIn) // the method expects to receive an Employee object
    {
        System.out.println(employeeIn.getStatus());
    }
}
```

In the above program it was clear at the time the program was compiled which version of getStatus was being referred to. The first time that the tester method is called, a FullTimeEmployee object is sent in, so the getStatus method of FullTimeEmployee is called; the second time that the tester method is called, a PartTimeEmployee object is sent in, so the getStatus method of PartTimeEmployee is called. But now have a look at the next program (where, incidentally, we have made use of our EasyScanner class for input).

EmployeeTester3

```
public class EmployeeTester3
{
    public static void main(String[] args)
    {
        Employee emp; // a reference to an Employee
        char choice;
        String numberEntered, nameEntered;
        double salaryEntered, payEntered;
        System.out.print("Choose (F)ull-Time or (P)art-Time Employee: ");
        choice = EasyScanner.nextChar();

        System.out.print("Enter employee number: ");
        numberEntered = EasyScanner.nextString();

        System.out.print("Enter employee name: ");
        nameEntered = EasyScanner.nextString();

        if(choice == 'F' || choice == 'f')
        {
            System.out.print("Enter annual salary: ");
            salaryEntered = EasyScanner.nextDouble();

            // create a FullTimeEmployee object
            emp = new FullTimeEmployee (numberEntered, nameEntered, salaryEntered);
        }
        else
        {
            System.out.print("Enter hourly pay: ");
            payEntered = EasyScanner.nextDouble();

            // create a PartTimeEmployee object
            emp = new PartTimeEmployee (numberEntered, nameEntered, payEntered);
        }
        testMethod(emp); // call tester with the object created
    }


    static void testMethod(Employee employeeIn)
    {
        System.out.println(employeeIn.getStatus());
    }
}
```

In this program, we call testMethod only once, and allow the user of the program to decide whether a FullTimeEmployee object is sent in as a parameter, or a PartTimeEmployee object. You can see that at the beginning of the program we have declared a reference to an Employee:

```
Employee emp;
```

Although Employee is an **abstract** class, it is perfectly possible to declare a reference to this class—what we would not be allowed to do, of course, is to create an Employee *object*. However, as you will see in a moment, we can point this

reference to an object of any subclass of Employee, since such objects, like FullTimeEmployee and PartTimeEmployee, are kinds of Employee.

You can see that we request the employee number and name from the user, and then ask if the employee is full-time or part-time. In the former case we get the annual salary and then create a FullTimeEmployee object which we assign to the Employee reference, emp.

```
if(choice == 'F' || choice == 'f')
{
       System.out.print("Enter annual salary: ");
       salaryEntered = input.nextDouble();

       // create a FullTimeEmployee object
       emp = new FullTimeEmployee (numberEntered, nameEntered, salaryEntered);
}
```

In the latter case we request the hourly pay and then assign emp to a new PartTimeEmployee object:

```
else
{
       System.out.print("Enter hourly pay: ");
       payEntered = input.nextDouble();

       // create a PartTimeEmployee object
       emp = new PartTimeEmployee (numberEntered, nameEntered, payEntered);
}
```

Finally we call the testMethod with emp:

```
testMethod(emp);
```

The getStatus method of the appropriate Employee object will then be called.

Here are two sample runs from this program:

Choose (F)ull-Time or (P)art-Time Employee: ***F***
Enter employee number: ***123***
Enter employee name: ***Robertson***
Enter annual salary: ***23000***
Full-Time

Choose (F)ull-Time or (P)art-Time Employee: ***P***
Enter employee number: ***876***
Enter employee name: ***Adebayo***
Enter hourly pay: ***25***
Part-Time

As you can see, we do not know *until the program is run* whether the getStatus method is going to be called with a FullTimeEmployee object or a PartTimeEmployee object—and yet when the getStatus method is called, the correct version is executed.

The technique which makes it possible for this decision to be made at run-time is quite a complex one, and differs slightly from one programming language to another.

9.8 The final Modifier

You have already seen the use of the keyword **final** in Chap. 2, where it was used to modify a variable and turn it into a constant. It can also be used to modify a class and a method. In the case of a class it is placed before the class declaration, like this:

```
public final class SomeClass
{
   // code goes here
}
```

This means that the class cannot be subclassed. In the case of a method it is used like this:

```
public final void someMethod()
{
   // code goes here
}
```

This means that the method cannot be overridden.

9.9 The *Object* Class

One of the very useful things about inheritance is the *is-a-kind-of* relationship that we mentioned earlier. For example, when the ExtendedOblong class extended the Oblong class it became a kind of Oblong—so, we can use Extended-Oblong objects with any code written for Oblong objects. When the Part-TimeEmployee class extended the Employee class it became a kind of Employee. We have seen in Sect. 9.6 that, in Java, if a method of some class expects to receive as a parameter an object of another class (say, for example, Vehicle), then it is quite happy to receive instead an object of a *subclass* of Vehicle—this is because that object will be *a kind of* Vehicle.

In Java, every single class that is created is in fact derived from what we might call a special “super superclass”. This super superclass is called Object. So every object in Java is in fact *a kind of* Object. Any code written to accept objects of type Object can be used with objects of any type.

9.10 The `toString` Method

In the previous chapter you saw a menu-driven program, BankApplication, which provided an option to display the details of a particular bank account—and this option of course made use of the get-methods of the Bank class.

If this were to be a real-world application, then—as you will see from our case study—there would be an extensive period of testing. In order to test an application, it would be very useful if we had a way of simply displaying all the information about an object without having to invoke a lot of individual methods each time.

We are able to do this by making use of a method called toString which belongs to the Object class, and is therefore inherited by all classes, and can be overridden for each individual class.

The System.out.print and println methods are overloaded, and have a version which simply accepts a whole object as a parameter, and then displays whatever has been defined in the object's toString method.

For example, we could add the following method to our BankAccount class:

```
@Override
 public String toString()
 {
     return "Name: " + accountName + '\n' + "Account number: " + accountNumber + '\n'
                                                              + "Balance: " + balance;
 }
```

We can test this out with a little program—notice how the println method is called simply with the name of the object:

ToStringDemo

```
public class ToStringDemo
{
    public static void main(String[] args)
    {
        BankAccount acc = new BankAccount("12345678", "Patel");
        System.out.println(acc);
    }
}
```

The output from this program would be:

```
Name: Patel
Account number: 12345678
Balance: 0.0
```

And if you are wondering what happens if we haven't overridden the toString method, the answer is that the output is simply the name of the class and its location in memory. So the above program would have given us something like:

```
BankAccount@15db9742
```

9.11 Wrapper Classes and Autoboxing

We mentioned previously that collection classes such as `ArrayList` cannot be used to hold simple types such as **`int`** or **`char`**. However it is not uncommon that you would want to be able to do exactly that. Well, there is no need to worry—Java provides a very simple means of doing this.

To understand how it works you need to know about **wrapper** classes. For every primitive type, Java provides a corresponding class—the name of the class is similar to the basic type, but begins with a capital letter—for example `Integer`, `Character`, `Float`, `Double`. They are called *wrappers* because they "wrap" a *class* around the basic *type*. So an object of the `Integer` class, for example, holds an integer value. In future chapters you will find that these classes also contain some other very useful methods.

We could declare a list of integers by using the `Integer` class with the following statement:

```
ArrayList<Integer> myList = new ArrayList<>();
```

One way of storing an integer value such as 37 in this array would be as follows:

```
myList.add(new Integer(37));
```

The constructor of the Integer class accepts a primitive value and creates the corresponding `Integer` object—here we have created an `Integer` object from the primitive value 37, and this is now stored in the array.

Java, however, allows us to make use of a technique known as **autoboxing**. This involves the automatic conversion of a primitive type such as an **`int`** to an object of the appropriate wrapper class. This allows us to do the following, which as you can see is much simpler:

```
myList.add(37);
```

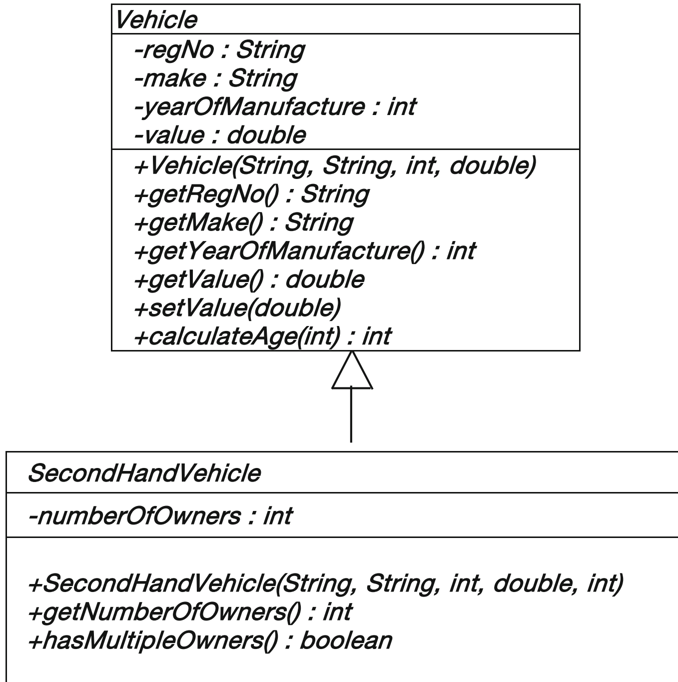
Java also allows us to make use of a technique called **unboxing**, which converts from the wrapper class back to the primitive type—so to assign the first item in the list to an integer `x`, we would simply write:

```
int x = myList.get(0);
```

Exactly the same technique would be used to store other primitive types such as **`char`** and **`double`**.

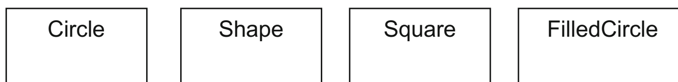
9.12 Self-test Questions

1. Below is a UML diagram for an inheritance relationship between two classes—`Vehicle` and `SecondHandVehicle`.



   (a) By referring to the diagram, explain the meaning of the term *inheritance*.
   (b) What do you think might be the function of each of the constructors?
   (c) What do you think might be the reason for the fact that in the `Vehicle` class there is a `set`-method for the `value` attribute, but not for the other three?
   (d) Write the header for the `SecondHandVehicle` class.

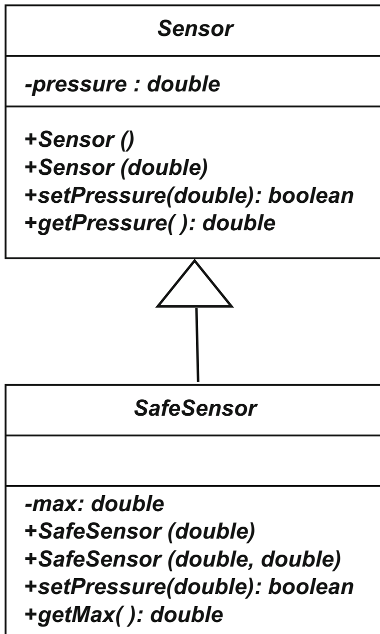
2. (a) Consider the following classes and arrange them into an inheritance hierarchy using UML notation:



(b) Write the top line of the class declaration for each of these classes when implementing them in Java.

(c) Explain what effect the **abstract** modifier has on a class and identify which, if any, of the classes above could be considered as abstract classes?

3. Consider once again an application to record the reading of a pressure sensor as discussed in programming exercise 4 of the previous chapter. Now assume a `SafeSensor` class is developed that ensures that the pressure is never set above some maximum value. A `SafeSensor` *is a kind of* `Sensor`. The UML design is given below:



The `SafeSensor` class has two constructors. The first sets the maximum safe value to the given parameter and the actual value of the sensor reading to 10. The second constructor accepts two parameters, the first is used to set the maximum safe value and the second is used to set the initial value for the reading of the sensor.
The `setPressure` method is redefined so that only safe values (values no greater than the safe maximum value and no less than zero) are set.

(a) In the example above, distinguish between *method overriding* and *method overloading*.

(b) Below is one attempt at the Java code for the first `SafeSensor` constructor. Identify why it will not compile.

```
// THIS WILL NOT COMPILE!!
public SafeSensor(double maxIn)
{
       max = maxIn;
       pressure = 10;
}
```

(c) Here is another attempt at the Java code for the first `SafeSensor` constructor. Identify why it will not compile.

```
// THIS WILL NOT COMPILE!!
public SafeSensor(double maxIn)
{
       max = maxIn;
       super();
}
```

(d) Write the correct code for the first `SafeSensor` constructor.

4. By referring to the `BankAccount` class of Sect. 9.5, distinguish between **`private`**, **`public`** and **`protected`** access.
5. How are all classes in Java related to the `Object` class?
6. Explain, with an example, the term *type cast*.
7. (a) Consider the following definition of a class called `Robot`:

```
public abstract class Robot
{
   private String id;
   private int securityLevel;
   private int warningLevel = 0;

   public Robot(String IdIn, int levelIn)
   {
        id = IdIn;
        securityLevel = levelIn;
   }

   public String getId()
   {
        return id;
   }


   public int getSecurityLevel()
   {
        return securityLevel;
   }

   public abstract void calculateWarningLevel();
}
```

(i) The following line of code is used in a program that has access to the `Robot` class:

```
Robot aRobot = new Robot("R2D2", 1000);
```

Explain why this line of code would cause a compiler error.

(ii) Consider the following class:

```
public class CleaningRobot extends Robot
{
   public String typeOfCleaningFluid;

   public CleaningRobot(String IdIn, int levelIn, String fluidIn)
   {
        super(IdIn, levelIn);
        typeOfCleaningFluid = fluidIn;
   }

   public String getTypeOfCleaningFluid()
   {
        return typeOfCleaningFluid;
   }
}
```

Explain why any attempt to compile this class would result in a compiler error.

8. What is the effect of the **final** modifier, when applied to both classes and methods?

9. Look back at the `EmployeeTester` class from Sect. 9.6. What do you think would happen if you replaced this line of `testMethod`:

```
System.out.println(employeeIn.getName());
```

with the following line?

```
System.out.println(employeeIn.getAnnualSalary());
```

Give a reason for your answer.

9.13 Programming Exercises

1. (a) Copy the `ExtendedOblong` class from the website, then implement the `ExtendedOblongTester` from Sect. 9.4. You will, of course, need to ensure that the `Oblong` class itself is accessible to the compiler.
   (b) Modify the `ExtendedOblongTester` program so that the user is able to choose the symbol used to display the oblong.

2. (a) Implement the `SafeSensor` class of self-test question 3. You will need to ensure that the `Sensor` class itself is accessible to the compiler.
   (b) Write a tester class to test the methods of the `SafeSensor` class.

3. (a) Implement the `Vehicle` and the `SecondHandVehicle` classes of self-test question 1.
   You should note that:
   - the `calculateAge` method of `Vehicle` accepts an integer representing the current year, and returns the age of the vehicle as calculated by subtracting the year of manufacture from the current year;
   - the `hasMultipleOwners` method of `SecondHandVehicle` should return **`true`** if the `numberOfOwners` attribute has a value greater than 1, or **`false`** otherwise.

   (b) Write a tester class that tests all the methods of the `SecondHandVehicle` class.

4. Write a menu-driven program that uses an `ArrayList` to hold `Vehicles`. The menu should offer the following options:

```
1. Add a vehicle
2. Display a list of vehicle details
3. Delete a vehicle
4. Quit
```

10 Introducing JavaFX

Outcomes:

By the end of this chapter you should be able to:

- *briefly describe the history of graphics programming in Java*;
- *explain the structure and life cycle of a **JavaFX** application*;
- *produce 2D graphical shapes in **JavaFX***;
- *build an interactive graphics application in **JavaFX** using common components such as buttons, textfields and labels*;
- *program a **JavaFX** control to listen for events using a **lambda expression***;
- *make use of a variety of different **JavaFX** containers*;
- *create borders, fonts and colours*;
- *format decimal numbers so that they appear in an appropriate form in a graphics application*.

10.1 Introduction

At last it is time to learn about graphics programming. In this chapter you will start to move away from that rather uninteresting text screen you have been using and build attractive windows programs for input and output.

In order to do this you are going to be using the Java graphics package known as **JavaFX**. This package provides all the graphics tools and components that you need to produce the sort of graphical interfaces that we have all become used to in modern day applications.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_10

10.2 A Brief History of Java Graphics

First we will give you a little bit of history. In the earliest versions of Java, graphical programming was achieved exclusively by making use of a package known as the **Abstract Window Toolkit (AWT)**. The idea with AWT was to provide a system of graphics in which any component that we create is associated with the corresponding component in the native operating system. So with AWT, if we were to create a graphics component (such as a button or text field for example) the component would be provided by the operating system—Windows™ or macOS™ for example—so that your button or text field would look exactly like the one you were used to in the particular operating system. Components that rely on the native operating system make extensive use of the system's resources and are therefore described as **heavyweight** components.

Because it used a lot of resources, and because of functional differences between operating systems, AWT was not entirely successful, and this system was replaced by a package called **Swing**. Swing classes are for the most part written in Java, and because they do not rely on the system components they are known as **lightweight** components. Unlike AWT, Swing components look the same regardless of the operating system the program is running on.

From around the year 2000–2014, Swing was the main platform for producing Java Graphics. However, its look and feel became rather old-fashioned compared to today's graphics that run across multiple devices, and with the release of Java 8 in 2014 came the latest version of a new technology known as JavaFX, together with the announcement that Swing will not be developed further (although it will continue to be packaged with Java).

In Fig. 10.1 you can see some examples of common graphics components using the three different technologies.

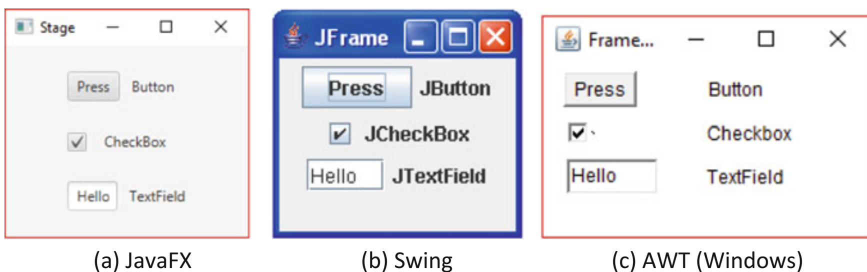


Fig. 10.1 Some typical JavaFX components compared with equivalent Swing and AWT components

10.3 JavaFX: An Overview

In this book we will be using JavaFX exclusively for our graphics applications. So first, some terminology. Firstly, you need to know that a JavaFX program is referred to as an **application**. Your JavaFX class will extend the `Application` class, for which you need the following import statement:

```
import javafx.application.Application
```

The top-level window in which the application runs is called a **stage**—normally this will be a window such as you see in Fig. 10.1a—but if, as you can do with many JavaFX applications, you run the program in full-screen mode, then the screen becomes the stage. Some applications can be made to run in a browser, in which case the browser is the stage. The contents of the stage—the graphic itself—is called a **scene**, and is often referred to as a **scene graphic**. The items that make up the scene are referred to as **nodes**. They are very commonly the kind of components that allow interaction with the user, such as buttons, text fields, labels and check boxes, which are often referred to collectively as **controls**. They can also be 2D or 3D graphics shapes. But nodes can also be **containers**. Containers are components that hold other nodes, and each container arranges the nodes in a particular way—for example, vertically, horizontally, in a grid, or stacked one on top of the other. Normally, we wouldn't see the container, but it is perfectly possible to put a border around it if we want to. Importantly, containers can contain other containers, so we can develop a hierarchy in our scene. We normally place a single top level node in our scene, and this is referred to as the **root** node. We use the terms **parent** and **children** for the containing and contained nodes respectively.

Figure 10.2 should make this clear. Here we have a sample scene in which the root node is a `VBox`—this is a container that arranges its child nodes vertically. We have given it a black border so that you can see it. The `VBox` has three children—a `TextField`, a `Label` and an `HBox`, around which we have again put a border. As you can probably guess, an `HBox` is similar to a `VBox`, but arranges its child nodes horizontally. In this case it has three child nodes which are `Buttons`. All of these components will become familiar to you as you proceed through this chapter—in particular you will see how we have made extensive use of the `VBox` and `HBox` to construct our scene graphics.

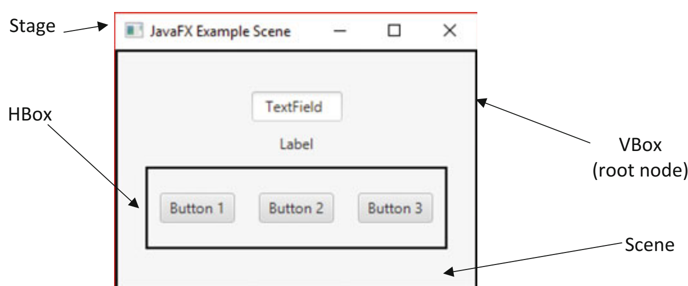


Fig. 10.2 A hierarchical scene

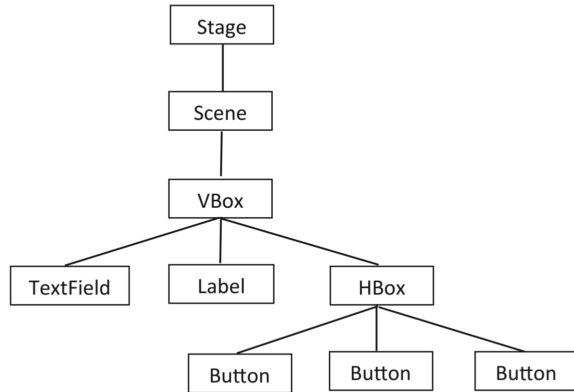


Fig. 10.3 The hierarchical structure of the scene in Fig. 10.2

To help you understand the way that a scene is constructed in a hierarchal way, we have shown you in Fig. 10.3 the hierarchy that makes up the scene graphic in the above example.

When a JavaFX application begins, there are three methods that are called in order. These are:

```
void init()
abstract void start(Stage stage)
void stop()
```

The first of these, `init`, is where we would place any routines that need to be carried out before the application itself starts, while the `stop` method is where we would place any code that we would want to be executed after the application finishes. We will not be concerning ourselves with these two methods in this chapter. What we will be concerning ourselves with, however, is the very important `start` method. As you can see, it is an **`abstract`** method and therefore has to be coded. It is in this method that the code for our application is placed. You can, of course, break this up by adding some helper methods, but it is with this method that the application itself begins.

So how do we launch a JavaFX application? Surprisingly it is not always via a `main` method. If the application is run from a command line, as described in the first chapter, then it doesn't actually require a `main` method to launch it. Neither does it need a `main` method if it is deployed as a `.jar` file, which is something you will learn about in Chap. 19. But if you run your program within an IDE, as most of you will be doing at first, then we do require a `main` method, and for that reason we have chosen to include such a method with each application that we develop here. You will see that the main method takes the following form:

```
    public static void main(String[] args)
    {
        launch(args);
    }
```

As you can see the main method calls the application's `launch` method, and passes to it any arguments received by the `main` method itself.

The `launch` method is a **`static`** method, and we can use it to launch a JavaFX application from another program. It is overloaded to accept the name of the compiled `.class` file as its first parameter. If we wanted a program called, say, `LaunchApplication` to run an application called `MyApp`, we would do it like this:

```
import javafx.application.Application;

class LaunchApplication
{
    public static void main(String[] args)
    {
        Application.launch(MyApp.class, args);
    }
}
```

So now that you know how a JavaFX application is structured, and you are aware of the sequence in which its methods are called, we can go on to develop our first graphics application.

10.4 2D Graphics: The *SmileyFace* Class

Our first graphics application is going to create a smiley face, as shown in Fig. 10.4.

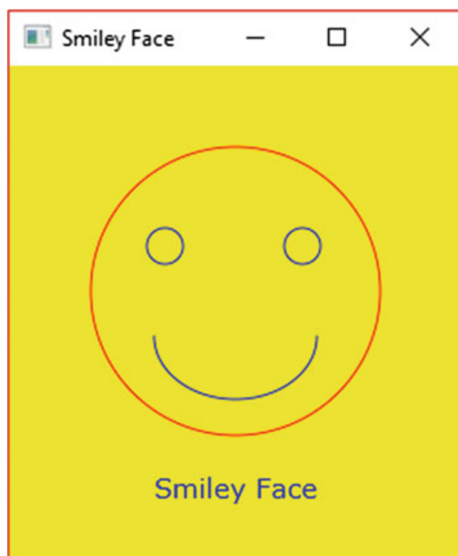


Fig. 10.4 The Smiley Face application

Although it is a rather simple application, in that there is no user interaction, it nonetheless introduces many new concepts. In particular it shows you how to create a scene, to add items to the scene, and to add the scene to a stage. It also introduces you to 2D graphics, which enables you to draw two-dimensional shapes such as circles, lines, ellipses, rectangles and arcs. Here we draw circles for the face and eyes, and an arc for the mouth. You will also see how to create text which you can configure using different colours and fonts.

As explained in the previous section, we have included a `main` method which launches the application. The complete code is shown below:

SmileyFace

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;


public class SmileyFace extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure the main circle for the face
        Circle face = new Circle(125, 125, 80);
        face.setFill(Color.YELLOW);
        face.setStroke(Color.RED);

        // create and configure the circle for the right eye
        Circle rightEye = new Circle(86, 100, 10);
        rightEye.setFill(Color.YELLOW);
        rightEye.setStroke(Color.BLUE);

        // create and configure the circle for the left eye
        Circle leftEye = new Circle(162, 100, 10);
        leftEye.setFill(Color.YELLOW);
        leftEye.setStroke(Color.BLUE);

        // create and configure a smiling mouth
        Arc mouth = new Arc(125, 150, 45, 35, 0, -180);
        mouth.setFill(Color.YELLOW);
        mouth.setStroke(Color.BLUE);
        mouth.setType(ArcType.OPEN);

        // create and configure the text
        Text caption = new Text(80, 240, "Smiley Face");
        caption.setFill(Color.BLUE);
        caption.setFont(Font.font("Verdana", 15));


        // create a group that holds all the features
        Group root = new Group(face, rightEye, leftEye, mouth, caption);

        // create and configure a new scene
        Scene scene = new Scene(root, 250, 275, Color.YELLOW);
```

```
        // add the scene to the stage, then set the title
        stage.setScene(scene);
        stage.setTitle("Smiley Face");

        // show the stage
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }

}
```

There are a number of new concepts here. First, let's take a look at the **`import`** clauses, which show you that all of our classes come from a package called `javafx`, which as you can see has many subpackages including `scene` and `stage`.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
```

Now look at the class header:

```
public class SmileyFace extends Application
```

As we explained, all JavaFX programs run as an application, and we therefore have to extend the `Application` class. `Application` requires you to code the `start` method, which we talked about in the previous section. Let's take a look at it now, starting with the header:

```
public void start(Stage stage)
```

When `start` is called, it is automatically sent an object of the `Stage` class, which will be the main container for our graphic.

The first thing we do within the `start` method is to create and configure the main circle for the face:

```
Circle face = new Circle(125, 125, 80);
face.setFill(Color.YELLOW);
face.setStroke(Color.RED);
```

The `Circle` class, which resides in a the `javafx.scene.shape` library, has a number of constructors (which you can look up on the Oracle™ site). The constructor we are using here takes three parameters of type **`double`**.

The first two of these parameters represent, respectively, the *x* and *y* positions of the centre of the circle (with respect to the top left hand corner of the parent node), measured in pixels. The third parameter represents the radius of the circle, also in pixels. You will see later that we have chosen our initial scene to be 250 × 275 pixels, so that our circle with its centre at (125, 125) will be horizontally centred, but will leave enough vertical room for a caption.

We have used two other methods of `Circle`, namely `setFill` and `setStroke`, to set the fill colour and line colour of the circle. To each of these we have passed a pre-defined attribute of the `Color` class (note the American spelling), which resides in `javafx.scene.paint`. In Sect. 10.10 you will find how to create your own colours if you want to—but the paint library provides a great many colours that you can use, and which you can look up—or which you can choose from the list of suggestions that your IDE will make after pressing the full stop.

In a similar manner we draw the right eye and the left eye:

```
Circle rightEye = new Circle(86, 100, 10);
rightEye.setFill(Color.YELLOW);
rightEye.setStroke(Color.BLUE);

Circle leftEye = new Circle(162, 100, 10);
leftEye.setFill(Color.YELLOW);
leftEye.setStroke(Color.BLUE);
```

You might be wondering how we decided upon the exact position in which to draw these circles. In theory it is possible to calculate exactly where you want everything to be on a graphic—but often it is easier (and actually quite good fun) simply to make an estimate and see how it looks, then change the values until you are happy. That's what we did here. We strongly recommend that once you have got the application up and running, you play about with the different values to explore what they do. This is the best way to become familiar with all of the graphics objects.

Now we come to the smiling mouth, which is a little more complicated.

```
Arc mouth = new Arc(125, 150, 45, 35, 0, -180);
mouth.setFill(Color.YELLOW);
mouth.setStroke(Color.BLUE);
mouth.setType(ArcType.OPEN);
```

Creating an object of the Arc class draws part (or all) of an ellipse. The constructor we have used is specified on the Oracle™ website like this:

```
Arc(double centreX, double centreY, double radiusX, double radiusY,
                                              double startAngle, double length)
```

The names of the parameters mostly speak for themselves. The first two represent the position of the centre of the ellipse. The next two are the horizontal and vertical radii respectively. startAngle represents the angle at which we start drawing the arc. The only confusing name is the last one, length, which represents the size of the angle through which the arc is drawn. Figure 10.5 should make it clear.

In our case we have chosen the radii to give us an arc of an ellipse which is somewhat wider than it is high. We have chosen a start angle of 0° and you will notice that the value of the final angle (the length parameter) is set to −180. The negative sign indicates that this angle is formed by moving from the start angle in an a clockwise direction (so that the mouth is smiling). A positive sign indicates an anticlockwise direction (as, for example, in Fig. 10.5).

The next lines of code set the fill colour and line colour (referred to as the stroke colour) of the mouth. The final line of code selects the type of arc we want, which in this case is ArcType.OPEN. Two other types exist (ArcType.CHORD and ArcType.ROUND), and these are demonstrated in Sect. 10.6.

The next thing we do is to add a caption:

```
Text caption = new Text(80, 240, "Smiley Face");
caption.setFill(Color.BLUE);
caption.setFont(Font.font("Verdana", 15));
```

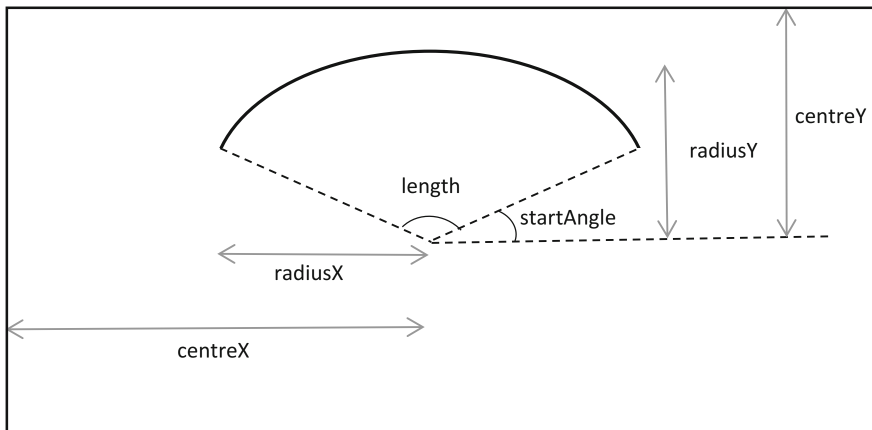


Fig. 10.5 The Arc class

For this purpose we are creating an instance of the Text class, which resides in javafx.scene.text. The constructor takes three parameters—two **double**s and a String. The first two are used to position the text (they are the co-ordinates of the beginning of the String), and the final one holds the value of the text itself.

We have gone on to set the colour, using the setFill method, and then we have set the font, with the setFont method. There will be more in Sect. 10.10 on how to create your own fonts, but for now you can just look at the syntax to see how we select the name and size of the font—in this case “Verdana”, 15 points.

Now that we have defined all of our features we want them to stay together as a group. We can do this with the Group class from the javafx.scene package. This class acts like an invisible container—it is very useful when we have already defined the position of our shapes (as we have done here), so we don’t have to worry any further about how they will be laid out within the container:

```
Group root = new Group(face, rightEye, leftEye, mouth, caption);
```

We are using the convention of naming the first node that we add to our scene root, as it is the root node. In this case it is the only node. We have created our new scene like this:

```
Scene scene = new Scene(root, 250, 275, Color.YELLOW);
```

Here we have chosen to use the constructor that allows us to set the size (width and height) of the initial scene, together with the background colour. If you don’t set these values initially, the Scene class (and other graphics components) have many set-methods such as setMinWidth and setMaxHeight that you can code later.

Now all that remains to complete the start method is to add the scene to the stage, set the title, and finally make the stage visible, which we do by calling its show method:

```
stage.setScene(scene);
stage.setTitle("Smiley Face");
stage.show();
```

As we mentioned before, we have included a main method (and will continue to do so) in order that you can run the application in any environment.

```
public static void main(String[] args)
{
     launch(args);
}
```

10.5 Event-Handling in JavaFX: The *ChangingFace* Class

The `SmileyFace` class that we developed in the last section was "passive" and didn't involve any interaction with the user. In practice of course, graphics applications will normally require input from the user in the form of clicking a button, entering text and so on.

Controls such as buttons need to respond when the user performs some action such as clicking a mouse. The way this works is that in response to the user performing such an action, the control generates an `Event` object. This object is sent to an `EventHandler` which we attach to a particular control and supply it with the instructions for what to do when the action occurs. There are many actions that the user could perform, such as pressing a key, dragging a mouse and so on, but in this chapter we will concern ourselves only with a simple mouse-click on a button.

Our first application will modify our `SmileyFace` class and turn it into a `ChangingFace` class that can change its mood so it can be sad as well as happy. We are going to add a couple of buttons, as shown in Figs. 10.6 and 10.7.

You can see that we have now changed our title and caption from "Smiley Face" to "Changing Face"—because when we have finished we will be able to click on the *Frown* button and get the face to look like the one you see in Fig. 10.7. Clicking the *Smile* button will get the face to smile again.

The code for our class is shown below. There are quite a lot of new concepts and techniques here, so we will discuss it in detail once you have had a look at it.

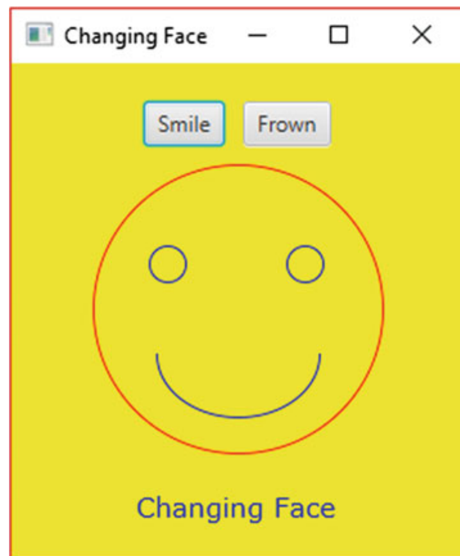


Fig. 10.6 The *ChangingFace* class (still smiling)

ChangingFace

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Background;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.geometry.Pos;

public class ChangingFace extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure the main circle for the face
        Circle face = new Circle(125, 125, 80);
        face.setFill(Color.YELLOW);
        face.setStroke(Color.RED);

        // create and configure the circle for the right eye
        Circle rightEye = new Circle(86, 100, 10);
        rightEye.setFill(Color.YELLOW);
        rightEye.setStroke(Color.BLUE);

        // create and configure the circle for the left eye
        Circle leftEye = new Circle(162, 100, 10);
        leftEye.setFill(Color.YELLOW);
        leftEye.setStroke(Color.BLUE);

        // create and configure a smiling mouth (this is how it will start)
        Arc mouth = new Arc(125, 150, 45, 35, 0, -180);
        mouth.setFill(Color.YELLOW);
        mouth.setStroke(Color.BLUE);
        mouth.setType(ArcType.OPEN);

        // create and configure the text
        Text caption = new Text(68, 240, "Changing Face");
        caption.setFill(Color.BLUE);
        caption.setFont(Font.font ("Verdana", 15));

        // create a group that holds all the features
        Group group = new Group(face, rightEye, leftEye, mouth,  caption);

        // create a button that will make the face smile
        Button smileButton = new Button("Smile");

        // create a button that will make the face frown
        Button frownButton = new Button("Frown");


        // create and configure a horizontal container to hold the buttons
        HBox buttonBox = new HBox(10);
        buttonBox.setAlignment(Pos.CENTER);

        //add the buttons to the horizontal container
        buttonBox.getChildren().addAll(smileButton, frownButton);

        // create and configure a vertical container to hold the button box and the face group
        VBox root = new VBox(10);
```

```
        root.setBackground(Background.EMPTY);
        root.setAlignment(Pos.CENTER);

        //add the button box and the face group to the vertical container
        root.getChildren().addAll(buttonBox, group);

        // create and configure a new scene
        Scene scene = new Scene(root, 250, 275, Color.YELLOW);

        // supply the code that is executed when the smile button is pressed
        smileButton.setOnAction(e -> mouth.setLength(-180));

        // supply the code that is executed when the frown button is pressed
        frownButton.setOnAction(e -> mouth.setLength(180));

        // add the scene to the stage, then set the title
        stage.setScene(scene);
        stage.setTitle("Changing Face");

        // show the stage
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

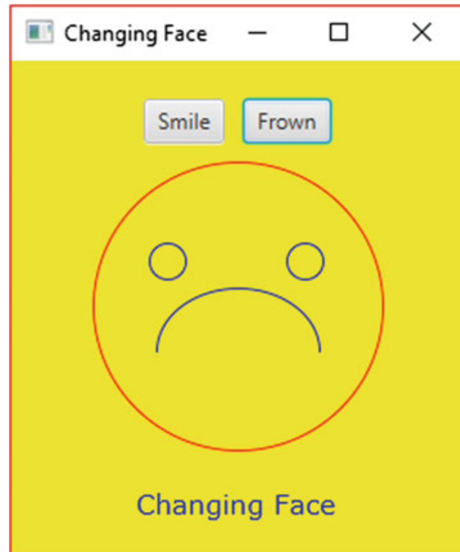


Fig. 10.7 The *ChangingFace* class (frowning)

We have proceeded as before when it comes to creating the face. Once we have done this we have created two instances of the `Button` class. A button is an extremely common feature of graphics programming, and the `Button` class, along with many other similar components is to be found in `javafx.scene.control`. Here is the code for the buttons:

```
Button smileButton = new Button("Smile");

Button frownButton = new Button("Frown");
```

You can see that we have used a version of the constructor that allows us to set the text that appears on the button. You can also use the `setText` method of `Button` for this purpose.

Having created our two buttons, we now go on to create a container to hold them:

```
HBox buttonBox = new HBox(10);
buttonBox.setAlignment(Pos.CENTER);
```

We have created an instance of an `HBox`. As we mentioned earleir, this is a container that arranges the contained nodes horizontally. The constructor we have used takes a parameter that sets the distance between the items, in this case 10 pixels. We have then gone on to use its `setAlignment` method, into which we send a pre-defined constant, an attribute of the `Pos` class which is found in the package `javafx.geometry`. The constant we have chosen is `Pos.CENTER` in order to centre the components that the `HBox` contains. There are a number of other options, and these are demonstrated in Sect. 10.9.1.

Having created our `HBox`, we need to add our buttons to it. We do this by calling a method of `HBox`, called `getChildren`, which returns a list of all the child nodes. This list has two methods for adding the nodes: the `add` method will add a single item, and `addAll` a list of items. As we need to add two buttons, we have used the latter:

```
buttonBox.getChildren().addAll(smileButton, frownButton);
```

So now have an `HBox` containing our buttons and a `Group` containing the shapes that make up our face. We need to organize these so that the face is placed vertically below the buttons, so we use a `VBox` which lines the items up vertically, just as the `HBox` does horizontally. We have given the name `root` to this instance of `VBox`, because this will be the root node that we add to our scene.

```
VBox root = new VBox(10);
root.setBackground(Background.EMPTY);
root.setAlignment(Pos.CENTER);
root.getChildren().addAll(buttonBox, group);
```

You will notice that we have done something else here, which is to add an empty background to the VBox—this is so that the yellow colour of the scene isn't hidden.

Now we can add the HBox containing the buttons, and the group containing the face, to the VBox. We then add this VBox to the scene graphic:

```
root.getChildren().addAll(buttonBox, group);
Scene scene = new Scene(root, 250, 275, Color.YELLOW);
```

We are almost ready to take the final step of adding the scene to the stage.

Almost but not quite! There is one really vital thing we have to do, which is to enable the buttons to respond when they are pressed, and to provide the code that tells the buttons what to do when this happens. At the beginning of this section we explained that a control can be programmed to generate an Event object in response to some action taken by the user. The type of Event that we are interested in here is called an ActionEvent, which is the one that handles a simple mouse-click. We need to add an EventHandler to each button, which means it will generate the ActionEvent as soon as the mouse is clicked. Effectively we are programming our button to "listen out" for a mouse-click.

EventHandler has a method called handle, and it is the code for this method that we need to supply in order that the button knows what to do when the mouse is clicked.

Now, you might think that all this sounds rather complicated—but we are in luck! Java 8 has furnished us with two things that mean our code for doing all this stuff is very simple. The code for doing this for each button is shown below. It contains some new syntax, which we will explain once you have had a look at it:

```
smileButton.setOnAction(e -> mouth.setLength(-180));
frownButton.setOnAction(e -> mouth.setLength(180));
```

You can see that a Button has a method called setOnAction. This an example of what is known as a **convenience method**, a feature of JavaFX. It certainly is convenient because it means that all we have to do in order to add an EventHandler is to supply the code it needs for its handle method. Most controls have these convenience methods, all starting with setOn-. Other examples that you will come across in the second semester are setOnMouseMoved and setOnKeyTyped, as well as many others.

You can see that the code (which looks rather unfamiliar because of the - > notation) is sent directly into the method. This is the other thing that we have been able to do as a result of innovations in Java 8. The code that you see is called a **lambda expression**. Lambda expressions allow us to simply send in some code to a method as an argument, just as we would with a value of a primitive type like **int**, or an object of a class like String.

We will look in detail at lambda expressions in Chap. 13. For now we will just tell you what you need to know in order to code the `setOnAction` method.

In each case we are using the `setLength` method of `Arc` to redraw the mouth. As we have seen, giving this a negative value draws it clockwise, so that the mouth smiles, and giving it a positive value makes the mouth frown. So the instructions for the `smileButton` and `frownButton` respectively are `mouth.setLength(-180)` and `mouth.setLength(180)`.

These instructions are the ones we have to supply to the `handle` method of the `EventHandler`. As we have said, lambda expressions enable us to supply this code by passing it as an argument to the `setOnAction` method. You can see from the above that our lambda expressions look like this:

```
e - > mouth.setLength(-180) //smile
e - > mouth.setLength(180) //frown
```

There are two parts to a lambda expression, one on either side of the `- >` symbol. The code goes on the right of the symbol. On the left we give names to the parameters that the method (in this case `handle`) expects to receive. The `handle` method receives an `ActionEvent`, and we have given this the name `e`. Even though we are not, in this case, going to use this variable, we nonetheless have to give a name.

There is a lot more to lambda expressions. But for now you are only going to use them in connection with a `setOnAction` method, so this is all you need for the moment. One thing we should add, however, is that if there is more than one instruction to our code we have to enclose the code in curly brackets. For example, if we wanted to paint the mouth violet when it smiles, our lambda expression would look like this:

```
smileButton.setOnAction(e -> {
                                mouth.setLength(-180);
                                mouth.setStroke(Color.VIOLET);
                            }
                        );
```

Finally we can add the scene to the stage, set the title and make the stage visible:

```
stage.setScene(scene);
stage.setTitle("Changing Face");

stage.show();
```

10.6 Some More 2D Shapes

Before we move away from 2D graphics, we will draw your attention to a some more shapes that will increase your repertoire.

We have shown a few examples in Fig. 10.8.

You can see that we have experimented with different colours, and with using the `setFill` and `setStroke` methods.

The rectangle that you see in the top left-hand corner was created with the following code:

```
Rectangle rectangle = new Rectangle(50, 50, 50, 100);
```

In this constructor, the first two parameters (all of which are of type **double**) represent the *x* and *y* co-ordinates of the top left hand corner, and the next two represent the width and height of the rectangle respectively.

The line underneath was created using the following constructor:

```
Line line = new Line(50, 180, 80, 250);
```

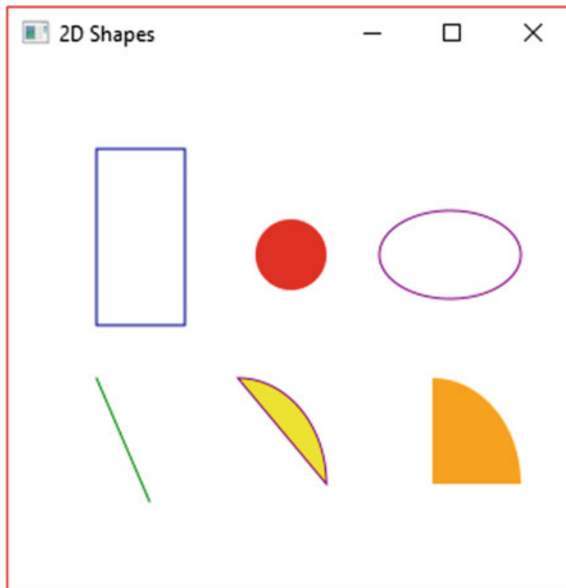


Fig. 10.8 Some more 2D shapes

The first two parameters are the *x* and *y* co-ordinates of the start position, and the last two are the co-ordinates of the end position.

The ellipse that you see in the top right hand corner was drawn simply by creating an `Arc` and drawing the line through an angle of 360°. In our previous examples you saw the effect of choosing `ArcType.OPEN` for our arc type. The two arcs you see on the bottom row show the effect of choosing `ArcType.CHORD` and `ArcType.ROUND` respectively.

All of the above shapes reside in `javafx.scene.shape`. You can check out the many other constructors and methods of these and other shapes on the Oracle™ website.

10.7 An Interactive Graphics Class

Most common applications involve controls (buttons, check boxes, text fields and so on) rather than graphical shapes. The next class—which we have called `PushMe`—is going to have controls that allow the user to input information via a graphics screen. The program isn't all that sophisticated, but it introduces the basic elements that you need to build interactive graphics classes.

This application allows the user to enter some text and then, by clicking on a button, to see the text that was entered displayed below the button. You can see what it looks like in Fig. 10.9.

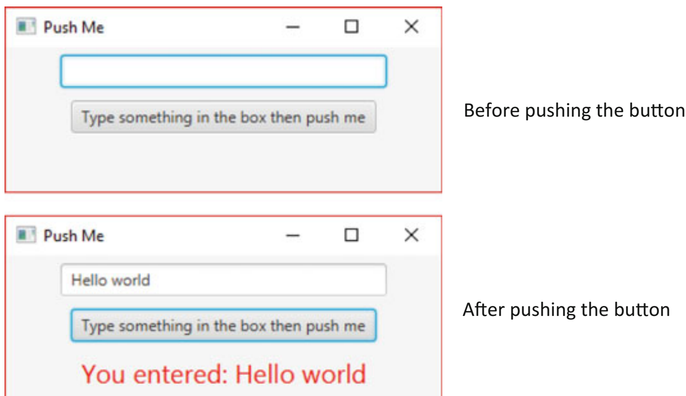


Fig. 10.9 The *PushMe* class

As usual we will show you the code first and discuss it afterwards:

PushMe

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class PushMe extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure a text field for user entry
        TextField pushMeTextField = new TextField();
        pushMeTextField.setMaxWidth(250);

        // create and configure a label to display the output
        Label pushMeLabel= new Label();
        pushMeLabel.setTextFill(Color.RED);
        pushMeLabel.setFont(Font.font("Arial", 20));

        // create and configure a label which will cause the text to be displayed
        Button pushMeButton = new Button();
        pushMeButton.setText("Type something in the box then push me");
        pushMeButton.setOnAction(e -> pushMeLabel.setText("You entered: " + pushMeTextField.getText()));

        // create and configure a VBox to hold our components
        VBox root = new VBox();
        root.setSpacing(10);
        root.setAlignment(Pos.CENTER);

        //add the components to the VBox
        root.getChildren().addAll(pushMeTextField, pushMeButton, pushMeLabel);


        // create a new scene
        Scene scene = new Scene(root, 350, 150);

        //add the scene to the stage, then configure the stage and make it visible
        stage.setScene(scene);
        stage.setTitle("Push Me");
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }

}
```

The box into which we type our text is called a `TextField`. This allows us to type in one line of text:

```
TextField pushMeTextField = new TextField();
pushMeTextField.setMaxWidth(250);
```

You can see that we have set the maximum width of our `TextField` to 250—if we had not done this, it would simply have filled the width of its parent container. You might want to explore a similar class, `TextArea`, that allows you to add several rows of text—you will see an example of this in the next section.

When the button is pressed, the text entered will be displayed underneath the button on a `Label`. As its name suggests, its purpose is simply to display some chosen text. We have created and configured it with the following lines of code:

```
Label pushMeLabel= new Label();
pushMeLabel.setTextFill(Color.RED);
pushMeLabel.setFont(Font.font("Arial", 20));
```

Next we have the code for the Button:

```
Button pushMeButton = new Button();
pushMeButton.setText("Type something in the box then push me");
pushMeButton.setOnAction(e -> pushMeLabel.setText("You entered: " + pushMeTextField.getText()));
```

We have already seen how to create and code a button, so this should be familiar to you. Look carefully a the lambda expression, which is explained in Fig. 10.10.

Having done all this, we create and configure a VBox, add the three components, and then add the VBox to the scene.

```
VBox root = new VBox();
root.setSpacing(10);
root.setAlignment(Pos.CENTER);

root.getChildren().addAll(pushMeTextField, pushMeButton, pushMeLabel);

Scene scene = new Scene(root, 350, 150);
```

Finally we add the scene to the stage, then add a title and make it visible.

```
stage.setScene(scene);
stage.setTitle("Push Me");
stage.show();
```

Use the getText method of TextField to read the current text, then append this to an introductory String

e -> pushMeLabel.setText("You entered: " + pushMeTextField.getText())

Use the setText method of Label to display the message

Fig. 10.10 The lambda expression explained

10.8 A Graphical User Interface (GUI) for the *Oblong* Class

Up till now, when we wanted to write programs that utilize our classes, we have written text-based programs. In most cases, however, you will be wanting to a create graphical user interface (GUI) for your classes. Let's do this for the `Oblong` class that we developed in Chap. 8. The sort of interface we are talking about is shown in Fig. 10.11.

Here is the code for the GUI:

***Oblong*GUI**

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;


public class OblongGUI extends Application
{
    // create an object of the Oblong class as an attribute
    private Oblong testOblong = new Oblong(0, 0);

    @Override
    public void start(Stage stage)
    {
        // create and configure text fields for input
        TextField lengthField = new TextField();
        lengthField.setMaxWidth(50);

        TextField heightField = new TextField();
        heightField.setMaxWidth(50);


        // create and configure a non-editable text area to display the results
        TextArea display = new TextArea();
        display.setEditable(false);
        display.setMinSize(210,50);
        display.setMaxSize(210,50);


        // create and configure Labels for the text fields
        Label lengthLabel = new Label("Length");
        lengthLabel.setTextFill(Color.RED);
        lengthLabel.setFont(Font.font("Arial", 20));

        Label heightLabel= new Label("Height");
        heightLabel.setTextFill(Color.RED);
        heightLabel.setFont(Font.font("Arial", 20));


        // create and configure a button to perform the calculations
        Button calculateButton = new Button();
        calculateButton.setText("Calculate");
        calculateButton.setOnAction( e ->
                   {
                       // check that fields are not empty
                       if(lengthField.getText().isEmpty() || heightField.getText().isEmpty())
                       {
                              display.setText("Length and height must be entered");
                       }
                       else
                       {
                           //convert text input to doubles and set the length and height of the Oblong
                           testOblong.setLength(Double.parseDouble(lengthField.getText()));
                           testOblong.setHeight(Double.parseDouble(heightField.getText()));

                           // use the methods of Oblong to calculate the area and perimeter
                           display.setText("The area is: " + testOblong.calculateArea()
                                                                + "\n" + "The perimeter is: "
                                                                   + testOblong.calculatePerimeter());
                        }
                   }
                                );

        // create and configure an HBox for the labels and text inputs
        HBox inputComponents = new HBox(10);
```

```
        inputComponents.setAlignment(Pos.CENTER);
        inputComponents.getChildren().addAll(lengthLabel, lengthField, heightLabel, heightField);

        // create and configure a vertical container to hold all the components
        VBox root = new VBox(25);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(inputComponents, calculateButton, display);

        // create a new scene and add it to the stage
        Scene scene = new Scene(root, 350, 250);
        stage.setScene(scene);
        stage.setTitle("Oblong GUI");
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

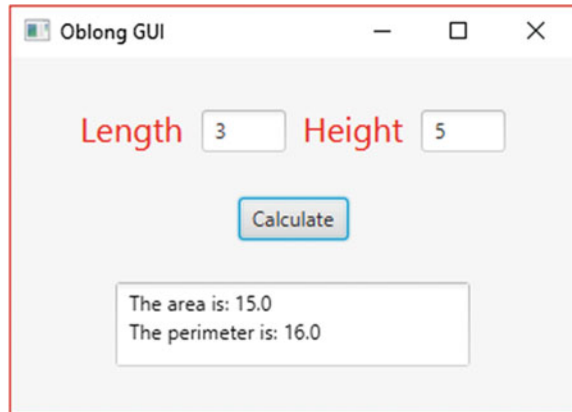


Fig. 10.11 A GUI for the *Oblong* class

In order to connect a GUI to a class, we create an object of that class within the GUI class—and as you can see that is what we have done here. We have declared an attribute, `testOblong`, which we have initialized as a new `Oblong` with a length and height of zero (since the user hasn't entered anything yet):

```
private Oblong testOblong = new Oblong(0,0);
```

After this we declare the graphics components; the only one of these that you have not yet come across is the `TextArea`, which is the large text area that you see in Fig. 10.8, where the area and perimeter of the oblong will be displayed. As you can see, it is a useful component for entering and displaying text, although this time we are using it only to display text, not to enter it. We have declared and configured it as follows:

```
TextArea display = new TextArea();
display.setEditable(false);
display.setMinSize(210,50);
display.setMaxSize(210,50);
```

We have prevented the possibility of entering text by the use of the setEditable method, and we have given it a fixed size by calling both the setMinSize and setMaxsize methods.

The only other thing we need to draw your attention to is the lambda expression that we have sent into the setOnAction method of the calculate button.

```
calculateButton.setOnAction( e ->
        {
             // check that fields are not empty
             if(lengthField.getText().isEmpty() || heightField.getText().isEmpty())
             {
                  display.setText("Length and height must be entered");
             }
             else
             {
                  //convert text input to doubles and set the length and height of the Oblong
                  testOblong.setLength(Double.parseDouble(lengthField.getText()));
                  testOblong.setHeight(Double.parseDouble(heightField.getText()));

                  // use the methods of Oblong to calculate the area and perimeter
                   display.setText("The area is: " + testOblong.calculateArea()
                                                  + "\n" + "The perimeter is: "
                                                        + testOblong.calculatePerimeter());
             }
           }
                      );
```

The first thing that we do here is to check that something has actually been entered. We do this by reading the String that is currently in the TextField by calling its getText method, and then calling the isEmpty method of String.

If either one of the fields is empty then an error message is displayed; otherwise we continue with the task. We could have, if we had wanted to, done some more input validation—for example we could have checked that zeros or negative numbers hadn't been entered. Or, if we wanted to be very strict about the definition of "oblong", we could have checked that the two adjacent sides were not equal. These are left as exercises at the end of the chapter.

If there is no error, we use the setLength and setHeight methods of Oblong to set the length and the height of testOblong to the values entered. However, these methods expect to receive **double**s—but the getText method of TextField, which we use to see what has been entered, returns a String!

We must therefore perform a conversion. To do this we use the parseDouble method of the Double class—one of the wrapper classes you learnt about in Chap. 8. parseDouble takes a String and converts it to a **double**:

```
testOblong.setLength(Double.parseDouble(lengthField.getText()));
testOblong.setHeight(Double.parseDouble(heightField.getText()));
```

It might have occurred to you that if the String did not contain a number, then this would cause a problem. We will show you how we deal with this sort of error in Chap. 14.

You should note that had we wanted to convert the Strings to **int**s, we would have used the parseInt method of the Integer class.

Incidentally, if you want to do this the other way round and convert a **double** or an **int** to a String you can do so simply by concatenating it onto an empty String, as shown in the examples below:

```
String s = "" + 3;
```

or:

```
String s = "" + 3.12;
```

If you take a look at the rest of the code you will see that we have arranged our items by using an HBox to hold the labels and fields for input so they are lined up horizontally, and then used a VBox to line this up vertically with the button and the display area. This is something you should be getting used to by now, so we can move on to the next section where we explain more about how to use these boxes, as well as other containers, each of which lays out the components in a different way.

10.9 Containers and Layouts

You have already seen how much we can achieve just with an HBox and a VBox—we have found these containers to be very versatile, and for simple applications you can do an enormous amount just with these two containers. So we start this section telling you a bit more about what we can do with these, and then we go on to show you some other containers with different layout policies.

10.9.1 More About *HBox* and *VBox*

In Fig. 10.12 you can see twelve VBoxes (although they could have been HBoxes because each one contains only one component), organized in four groups of three. We have drawn a border around each one and coloured the background (we will show you how to do this in a moment). Each box contains a Button—the presence of the border shows the effect of setting the alignment to different values.

You will recall that the only alignment we have seen so far is Pos.CENTER, but there are 11 others that we can choose from. Most are self-explanatory—but the three on the right need a little explanation. Pos.BASELINE_LEFT, Pos.BASELINE_CENTER and Pos.BASELINE_RIGHT place the component in the

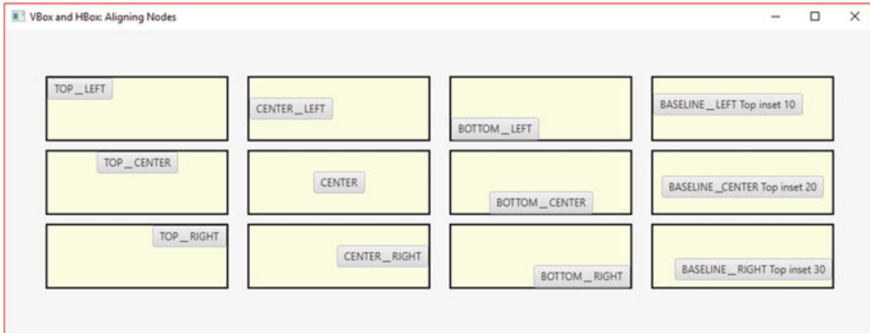


Fig. 10.12 Aligning nodes

lowest position available, and are most relevant to text fields where we want the text to appear at the bottom of a window—as in a chat application, for example. In our diagram we have set our boxes to have a top inset, and the buttons are then positioned accordingly.

To set some insets on a component we use the `setPadding` method, with a statement such as this:

```
box.setPadding(new Insets(10, 20, 10, 20));
```

The parameters to the `Insets` constructor are all **`double`**s, and define the insets for the top, right, bottom and left insets respectively. A single parameter would set all four insets to the same value.

Here `box` could be any component such as a `VBox` or `HBox`, or a `Button`, `Label` or `TextField` for example, as all these inherit the `setPadding` method from a higher level class.

We also promised to show you how to create a border and background colour. To get the black borders that you see in the diagram we did the following:

```
box.setBorder(new Border(new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID,
                                        new CornerRadii(0),new BorderWidths(2))));
```

This does seem to be a rather complicated process, but if you study it you can easily see what's going on. The `setBorder` method requires an object of the class `Border`. To create this we have to send the constructor an object of `Border-Stroke`, which requires four arguments. The argument names should speak for themselves, except perhaps for `ConerRadii`, which determines the roundness of the corners; in this example a value of zero produces square corners. To achieve the background colour we did this:

```
box.setBackground(new Background(new BackgroundFill(Color.LIGHTYELLOW,
                                                     new CornerRadii(0), new Insets(0))));
```

Again, although it looks complicated at first sight, it isn't hard to work out what is actually going on.

All of these border and background classes reside in `javafx.scene.layout`.

We will do some more work on borders and colours in Sect. 10.10.

10.9.2 GridPane

A `GridPane` is a very useful container. As its name suggests, it lays out the components in a matrix of rows and columns, as shown in Fig. 10.13.

The following lines of code would create a `GridPane` object, and configure it to position the components in the centre of each cell, and to leave a 10 pixel vertical gap (the gap between rows) and a 5 pixel horizontal gap (the gap between columns).

```
GridPane pane = new GridPane();
pane.setAlignment(Pos.CENTER);
pane.setVgap(10);
pane.setHgap(5);
```

The really good thing about a `GridPane` is its flexibility—it is sized dynamically as you insert the components, as are the individual cells. For example we could insert a `Button`, `myButton`, to the above `GridPane` object as follows:

```
pane.add(myButton, columnIndex, rowIndex);
```

`columnIndex` and `rowIndex` are **`int`**s—we start counting from zero, so the following line of code would add the button in column 4, row 6:

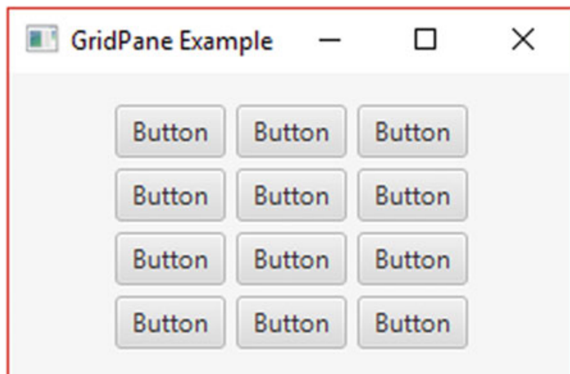


Fig. 10.13 Using a *GridPane*

```
pane.add(myButton, 3, 5);
```

Because of its flexibility, `GridPane` can be very versatile and will allow you to create quite complex presentations—the best thing you can do, as usual, is to try some experiments of your own.

10.9.3 StackPane

A `StackPane`, as its name suggests, stacks components on top of each other. In Fig. 10.14 we have created three different coloured rectangles, each one smaller than the previous one, and added them to the `StackPane` from largest to smallest.

The components are added as before by calling the `getChildren` method inherited by `StackPane`. Here we have chosen to align them in the centre of the pane. You can see that there is a lot of potential here for drawing interesting shapes, and as before you should conduct your own experiments.

There is another very useful way in which we can utilize this container, by creating several items each the same size and stacking them one on top of the other. We can then choose which one is visible, so that the stack behaves like a pack of cards. In this way we can create a series of screens which could, for example, be forms that need to be filled in progressively. We will show you an example of this in Chap. 17.

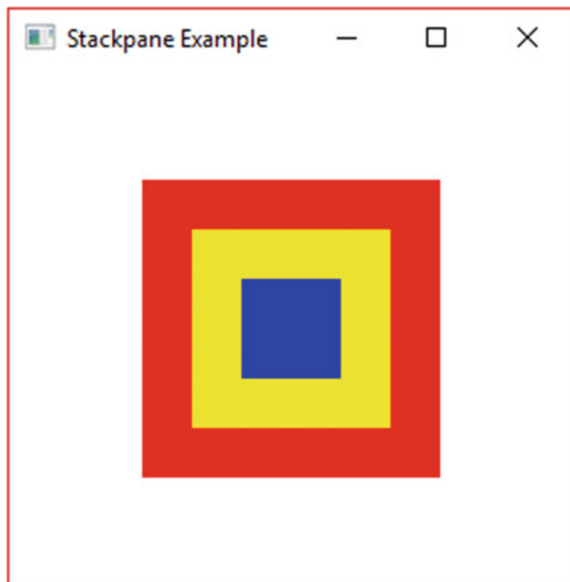


Fig. 10.14 Using a *StackPane*

10.9.4 *FlowPane* and *BorderPane*

As we explained in Sect. 10.2, prior to the existence of JavaFX the principal package for producing graphics in Java was Swing. With Swing, the way that a container arranged its components (its *layout policy*) was determined by associating a particular *layout manager* with it. Two of the most common of these were `FlowLayout` and `BorderLayout`. As you have already seen, JavaFX works by providing different containers, each with its own layout policy—and with the existence of `VBox` and `HBox`, flow layout and border layout policies are not as useful as they were in the days of Swing. However, two containers that produce similar results to these are nonetheless provided in JavaFX—these are `FlowPane` and `BorderPane`.

A `FlowPane` operates by arranging the items in a row, in the order that they were added, starting a new row when necessary. If the window is resized, the components move about accordingly, as shown in Fig. 10.15.

As with the `VBox` and `HBox`, nodes are added to a `FlowPane` object by calling the `getChildren` method and then using `add` or `addAll`.

The `BorderPane` operates by dividing the window into five regions called *Top*, *Bottom*, *Left*, *Right*, *Center*, as shown in Fig. 10.16.

If we use a `BorderPane` the components don't get moved around when the window is resized, as you can see from Fig. 10.17.

To add an item called `myButton`, for example, to the top region of a `BorderPane` named `pane`, we would do the following:

```
pane.setTop(myButton);
```

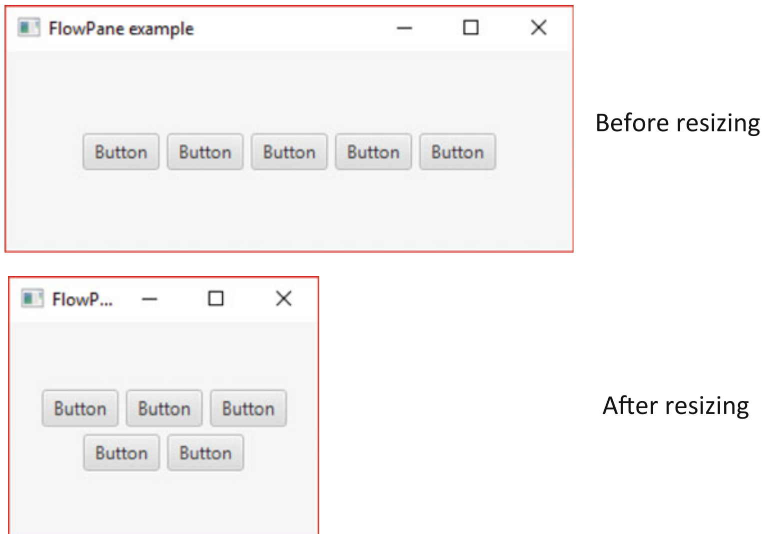


Fig. 10.15 The effect of resizing when using *FlowPane*

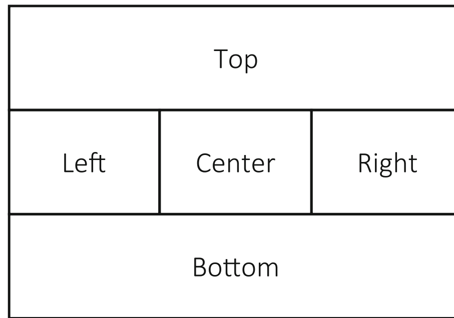


Fig. 10.16 The *BorderPane* layout strategy

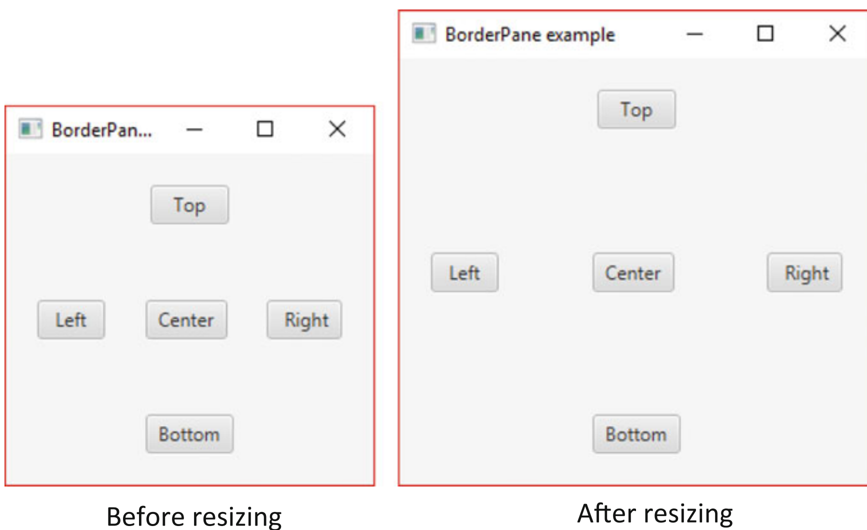


Fig. 10.17 The effect of resizing when using *BorderPane*

Similarly, for the other regions `BorderPane` has methods `setBottom`, `setLeft`, `setRight` and `setCenter`.

10.10 Borders, Fonts and Colours

You have already seen examples of how we can place borders around components and how we can determine the colour of text and background, and define different fonts. In this section we will give you a few more pointers as to how to enhance your applications with these features.

10.10.1 Borders

In Fig. 10.18 we see six buttons all with different border styles.

You will recall from Sect. 10.9.1 that in order to place a border round a component such as a button we use the `setBorder` method—this requires an object of the class `Border`, which in turn is created with an object of the class `BorderStroke`.

So, for example, for the first button (`button1`) we created the following `BorderStroke`:

```
BorderStroke stroke1
    = new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID, new CornerRadii(0), new BorderWidths(6)
```

We then applied this to our button:

```
button1.setBorder(new Border(stroke1));
```

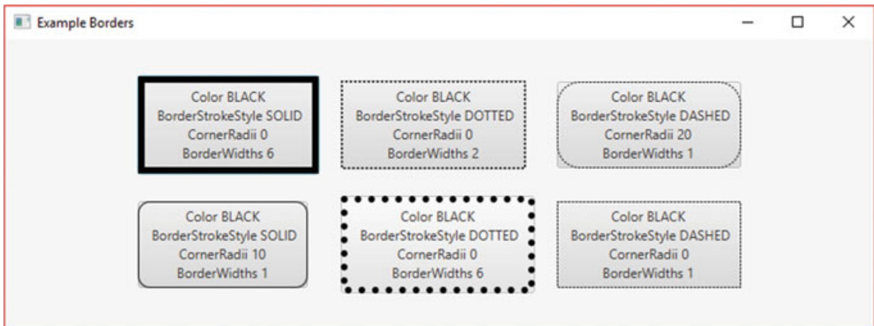


Fig. 10.18 Examples of border styles

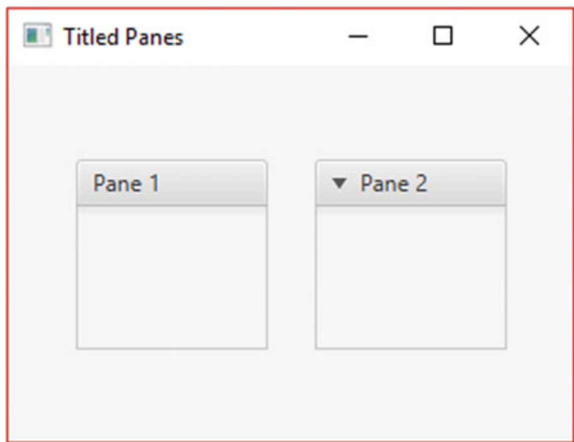


Fig. 10.19 Titled panes

One further effect is to create a titled border as shown in Fig. 10.19.

To do this we use a TitledPane, which is actually a control class, and resides in the package javafx.scene.control. The pane comes with a downward arrow that allows you to collapse the pane, as you see in the second box in Fig. 10.19. If you want a title only for decorative purposes you can turn this feature off as in the first box. In each case a VBox was added to the pane; other nodes could then be added to the VBox.

The code for creating the first pane is as follows:

```
VBox box1 = new VBox();
box1.setMinSize(100, 75);
TitledPane firstPane = new TitledPane("Pane 1", box1);
firstPane.setCollapsible(false);
```

As you can see, the last line turns off the collapsible feature.

10.10.2 Fonts

Figure 10.20 shows some different font examples.

In our example we have applied our fonts to various Text objects. To achieve this we used the setFont method of Text—controls such as Button also have a setFont method.

There are two ways of doing this. Firstly we can use the font method of Font. In our first example, assuming the Text object is called caption1, we would have written:

```
Font font1 = Font.font ("Verdana", FontWeight.BOLD, FontPosture.ITALIC, 12);
caption1.setFont(font1);
```

Fig. 10.20 Font examples

There are a number of different versions of the `font` method which you can look up on the Oracle™ website.

The other way of doing this is to create a new font with one of two constructors. The first requires only the font size (a **`double`**) and uses the default system font. The second requires the name of the font and the size. Our fifth example in Fig. 10.20 was achieved like this:

```
Font font5 = new Font("Calibri", 40);
```

Underlining is done by using the `setUnderline` method of a component.

10.10.3 Colours

You have seen how the JavaFX `Color` class has a great many predefined colours. However we can, if we wish, create our own colours. Those of you who have studied some elementary physics will know that there are three primary colours, red, green and blue[1]; all other colours can be obtained by mixing these in different proportions.

Mixing red, green and blue in equal intensity produces white light; the colour we know as black is in fact the absence of all three. Mixing equal amounts of red and green (and no blue) produces yellow light; red and blue produce a mauvish colour called magenta; and mixing blue and green produces cyan, a sort of turquoise.

The `Color` class has a method called `rgb` which allows us to specify the intensity of each of the primary colours, red, green, blue respectively. The intensity for each colour can range from a minimum of zero to a maximum of 255. So there are 256 possible intensities for each primary colour, and the total number of different colours available to us is therefore $256 \times 256 \times 256$, or 16,777,216.

As an example, you could create the following colour:

```
Color color1 = Color.rgb(200, 100, 50);
```

You could then, for example, set the text colour of a button, `button1`, like this:

```
button1.setTextFill(color1);
```

This particular combination produces a kind of orange—you should experiment with different values.

[1]Don't confuse this with the mixing of coloured paints, where the rules are different. In the case of mixing coloured lights (as on a computer monitor) we are dealing with reflection of light—in the case of paints we are dealing with absorption, so the primary colours, and the rules for mixing, are different. For paints the primary colours are red, blue and yellow.

10.11 Number Formatting

We are now going to tell you about a technique that is not specifically related to JavaFX, but is something you will often want to use in your graphics applications. We frequently need our numerical output to appear in a suitable format—for example with no more than two numbers after the decimal point—or perhaps with *exactly* two numbers after the decimal point. In order to do this we make use of the `DecimalFormat` class that resides in the `java.text` package. We would need the following import statement:

```
import java.text.DecimalFormat;
```

Once you have access to this class you can create `DecimalFormat` objects in your program. These objects can then be used to format decimal numbers for you. The `DecimalFormat` constructor has one parameter, the format string. This string instructs the object on how to format a given decimal number. Some of the important elements of such a string are given in Table 10.1.

In the example in the next section we are going to create the following `DecimalFormat` object:

```
DecimalFormat df = new DecimalFormat("0.0#");
```

Here the `DecimalFormat` object, `df`, is being informed on how to format any decimal numbers that may be given to it, as shown in see Fig. 10.21.

Having created a `DecimalFormat` object, we could then create a `String`, `s`, from a **`double`**, `d`, as follows:

```
String s = df.format(d);
```

Table 10.1 Special `DecimalFormat` characters

Character	Meaning
.	Insert a decimal point
,	Insert a comma
0	Display a single digit
#	Display a single digit or empty if no digit present

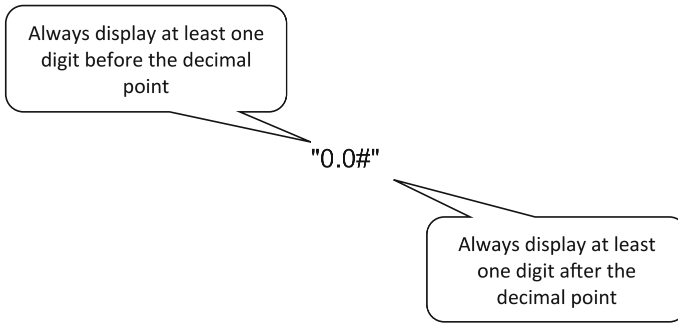


Fig. 10.21 A format String used with the *DecimalFormat* class

The program below shows some examples:

NumberFormatExample

```
import java.text.DecimalFormat;

public class NumberFormatExample
{
   public static void main(String[] args)
   {
      double number = 4376.7863;

      DecimalFormat df1 = new DecimalFormat("###,##0.0#");
      DecimalFormat df2 = new DecimalFormat("###000.00");
      DecimalFormat df3 = new DecimalFormat("00.0");
      DecimalFormat df4 = new DecimalFormat("000000.00000");
      DecimalFormat df5 = new DecimalFormat("000,000.00####");

      System.out.println(df1.format(number));
      System.out.println(df2.format(number));
      System.out.println(df3.format(number));
      System.out.println(df4.format(number));
      System.out.println(df5.format(number));
   }
}
```

The output from the above program is as follows:

```
4,376.79
4376.79
4376.8
004376.78630
004,376.7863
```

In the next chapter you will see how a similar technique can be used to output numbers as a particular currency.

10.12 A Metric Converter

Our final example in this chapter is a practical application that pulls together everything that we've learnt so far about JavaFX. Most of the world uses the metric system; however, if you are in the United Kingdom as we are, then you will still be only halfway there—sometimes using kilograms and kilometres, sometimes pounds and miles. Of course if you are in the USA (and you are not a scientist or an engineer) you will still be using the old imperial values for everything. Some might say it's time that the UK and the USA caught up with the rest of the world! But until that happens this little program, which converts back and forth from metric to imperial, is going to be very handy.

We will be building a `MetricConverter` class. Figure 10.22 shows what we are going to achieve.

The application will have a `VBox` at its root—this `VBox` will hold three `HBoxes`, one for each row that you see in Fig. 10.22. Each row requires two `Buttons` which will perform the calculations in either direction; these two `Buttons` will be held together in a VBox.

The code for the `MetricConverter` class is now presented; it looks quite long, but most of it is just more of what you already know. Take a look at the code and then we will draw your attention to a few points.

MetricConverter

```
import java.text.DecimalFormat;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class MetricConverter extends Application
{
    @Override
    public void start(Stage stage)
    {
       DecimalFormat df = new DecimalFormat("0.0#"); // see discussion in previous section

       // first the components for converting back and forth from inches to centimetres

       // create input fields, and labels to show the units
       TextField cmText = new TextField();
       Label cmLabel = new Label("Cm");
       TextField inchText = new TextField();
       Label inchLabel = new Label("Inches");

       // create buttons to perform the calculations
       Button cmToInchButton = new Button(" ===> ");
       Button inchToCmButton = new Button(" <=== ");
```

```
        // create a VBox to hold the buttons
        VBox inchCmButtons = new VBox();
        inchCmButtons.getChildren().addAll(cmToInchButton, inchToCmButton);

        // create an HBox to hold all the items for the first row
        HBox inchCmPanel = new HBox(10); // compound container
        inchCmPanel.getChildren().addAll(cmText, cmLabel, inchCmButtons, inchText, inchLabel);
        inchCmPanel.setAlignment(Pos.CENTER);

        // next the components for converting back and forth from miles to kilometres

        // create input fields, and labels to show the units
        TextField kmText = new TextField();
        Label kmLabel = new Label("Km");
        TextField mileText = new TextField();
        Label mileLabel = new Label("Miles  "); // extra spaces make all labels the same length

        // create buttons to perform the calculations
        Button kmToMileButton = new Button(" ===> ");
        Button mileToKmButton = new Button(" <=== ");

        // create a VBox to hold the buttons
        VBox mileKmButtons = new VBox();
        mileKmButtons.getChildren().addAll(kmToMileButton, mileToKmButton);

        // create an HBox to hold all the items for the second row
        HBox mileKmPanel = new HBox(10);
        mileKmPanel.getChildren().addAll(kmText, kmLabel, mileKmButtons, mileText, mileLabel);
        mileKmPanel.setAlignment(Pos.CENTER);

        // finally the components for converting back and forth from pounds to kilograms

        // create input fields, and labels to show the units
        TextField kgText = new TextField();
        Label kgLabel = new Label("Kg "); // extra spaces make all labels the same length
        TextField poundText = new TextField();
        Label poundLabel = new Label("Lb        "); // extra spaces make all labels the same length

        // create buttons to perform the calculations
        Button kgToPoundButton = new Button(" ===> ");
        Button poundToKgButton = new Button(" <=== ");

        // create a VBox to hold the buttons
        VBox poundKgButtons = new VBox();
        poundKgButtons.getChildren().addAll(kgToPoundButton, poundToKgButton);

        // create an HBox to hold all the items for the third row
        HBox poundKgPanel = new HBox(10);
        poundKgPanel.getChildren().addAll(kgText, kgLabel, poundKgButtons, poundText, poundLabel);
        poundKgPanel.setAlignment(Pos.CENTER);

        // create a VBox to hold all three rows
        VBox root = new VBox(10);
        root.getChildren().addAll(inchCmPanel, mileKmPanel, poundKgPanel);
        root.setAlignment(Pos.CENTER);

        // write the code for the buttons
        cmToInchButton.setOnAction( e -> {
                                        String s = new String(cmText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d / 2.54;
                                        s = df.format(d);
                                        inchText.setText(s);
                                  }
                           );

        inchToCmButton.setOnAction( e -> {
                                        String s = new String(inchText.getText());
```

```
                                        double d = Double.parseDouble(s);
                                        d = d * 2.54;
                                        s = df.format(d);
                                        cmText.setText(s);
                                    }
                                );

        kmToMileButton.setOnAction( e -> {
                                        String s = new String(kmText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d / 1.609;
                                        s = df.format(d);
                                        mileText.setText(s);
                                    }
                                );

        mileToKmButton.setOnAction( e -> {
                                        String s = new String(mileText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d * 1.609;
                                        s = df.format(d);
                                        kmText.setText(s);
                                    }
                                );

        kgToPoundButton.setOnAction( e -> {
                                        String s = new String(kgText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d * 2.2;
                                        s = df.format(d);
                                        poundText.setText(s);
                                     }
                                );

        poundToKgButton.setOnAction( e -> {
                                        String s = new String(poundText.getText());
                                        double d = Double.parseDouble(s);
                                        d = d / 2.2;
                                        s = df.format(d);
                                        kgText.setText(s);
                                     }
                                );

        // create a new scene
        Scene scene = new Scene(root);

        // add the scene to the stage, then configure the stage and make it visible
        stage.setScene(scene);
        stage.setTitle("Metric Converter");
        stage.setWidth(500);
        stage.setHeight(250);
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

As we have said, there is nothing here that you haven't seen before. But do study the code for the buttons. The code for each one is very similar, differing only in the particular calculation that needs to be made. Look carefully at how we have made use to the `DecimalFormat` class that we described in the previous section. We declared an object of this class at the beginning of the `start` method, and applied it to each of the buttons. The particular format we chose will always display at least one digit after the decimal point once the calculation has been made; you might want to try other formats.

We will be returning to graphics and JavaFX in the second semester. For the time being you already have a very big repertoire with which to build graphical applications. Please do explore the classes available, and try your own programs—everything

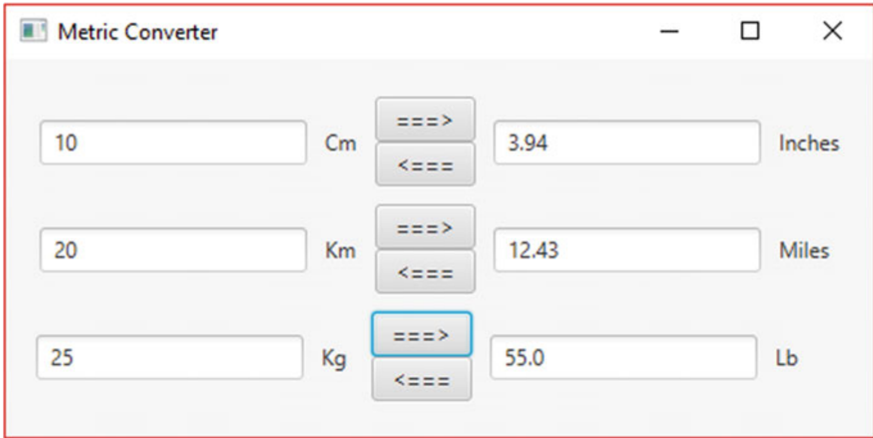


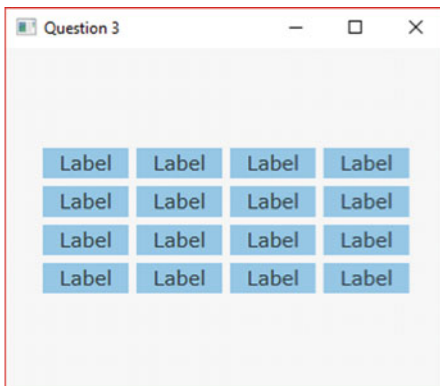
Fig. 10.22 The metric converter

you need is on the Oracle™ site. The more you try things out, the more you will learn and the more you will become familiar with what is available. And of course the more fun you will have!

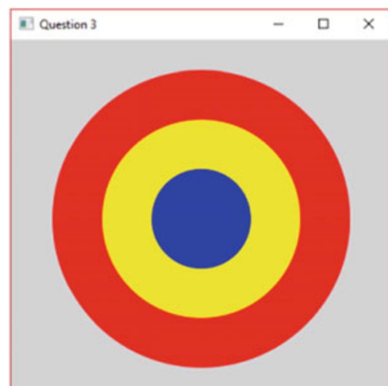
10.13 Self-test Questions

1. Briefly describe the history of graphics programming in Java.
2. What is the name of the three methods that are called when a JavaFX application is launched? what is the purpose of each?
3. Which containers have been used in the following two scene graphics?

(a)



(b)

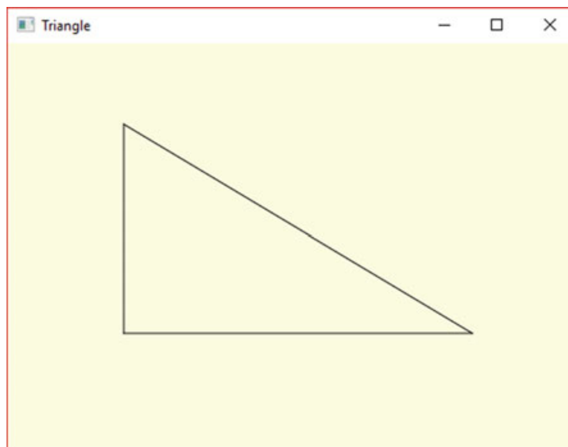


4. Describe how the following containers lay out the nodes that they contain:

   (a) a `VBox`; (b) an `HBox` (c) a `GridPane` (d) a `StackPane`
   (e) a `FlowPane` (f) a `BorderPane`

10.14 Programming Exercises

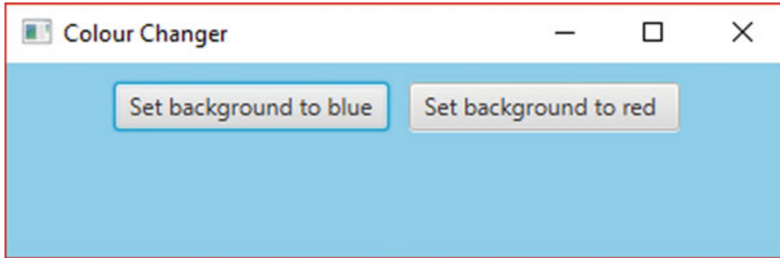
1. Implement a few of the programs that we have developed in this chapter, and experiment with different settings in order to change some the features—for example size, colour, position and so on.
2. Consider some changes or additions you could make to the `PushMe` class. For example, pushing the button could display your text in upper case—or it could say how many letters it contains. Maybe you could add some extra buttons.
3. The application shown below produces a triangle:



   See if you can write the code to produce this triangle using three lines. We suggest the following vertices:

   (100, 70) (100, 250) (400, 250).

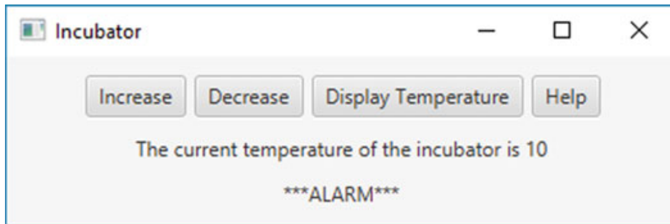
4. Below you see an application called `ColourChanger` which produces the following graphic in which two buttons can be used to change the background colour:



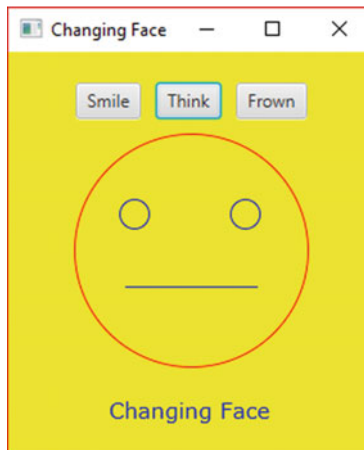
Write the code for this application.

5. Add some additional features to the `MetricConverter`—for example Celsius to Fahrenheit or litres to pints.

6. Look back at the final version of the `Incubator` class that you wrote in programming Exercise 5 of Chap. 8. Now you can create a graphical user interface for it, instead of a text menu. A suggested interface is shown below:



7. Below is a variation on the `ChangingFace` class, which has three possible moods!



Rewrite the original code to produce this new design.

Hint: *The easiest way to achieve the "thinking" mouth is to set the* `radius` *attribute of* `Arc` *to zero.*
A more difficult approach would be to draw a line, but then you would have to create three different mouths, and each time check which was the current mouth, remove that, and add the mouth you require. It is perfectly possible to do this because the list of nodes returned by the `getChildren` *method has methods named* `contains` *and* `remove` *as well as the* `add` *and* `addAll` *methods that you are used to.*

8. Look back at the `OblongGUI` that we developed in Sect. 10.8. Modify the code so that as well as checking that the values have been entered, it also checks that the values entered are not zero, and that the length and height are not equal.

Check for updates

Case Study—Part 1

11

Outcomes:

By the end of this chapter you should be able to:

- *describe each stage of the software development process*;
- ***design** a complete application using UML*;
- ***implement** a detailed UML design in Java*;
- ***document** their code using **Javadoc** comments*;
- *distinguish between **unit testing** and **integration testing***;
- ***test** individual program units by creating suitable drivers*;
- *document their test results professionally using a **test log***.

11.1 Introduction

The process of developing software requires us to carry out several tasks. They can be summarized as follows:

- **analysis and specification**: determining *what* the system is required to do (analysis) and writing it down in a clear and unambiguous manner (specification);
- **design**: making decisions about *how* the system is to be built in order to meet the specification;
- **implementation**: turning the design into an actual program;
- **testing**: ensuring that the system has been implemented correctly to meet the original specification;
- **installation**: delivering and setting up the completed system;
- **operation and maintenance**: running the final system and reviewing it over time—in light of changing requirements.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_11

Rather than completely finishing one task before beginning the next, object-oriented languages like Java encourage systems to be developed a little bit at a time. So, for example, we can build one class and test it (maybe in the presence of the client) before moving onto the next, rather than waiting for the whole system to be developed before testing and involving the client.

In this and the following chapter we will demonstrate this process by developing a case study that will enable you to get an idea of how a commercial system can be developed from scratch; we start with an informal description of the requirements, and then specify and design the system using UML notation and pseudocode where necessary. From there we go on to implement our system in Java. Java applications, such as this, typically consist of many classes working together. When testing for errors we will start with a process of **unit testing** (testing individual classes) followed by **integration testing** (testing classes that together make up an application).

The system that we are going to develop will keep records of the residents of a student hostel. In order not to cloud your understanding, we have simplified things, keeping details of individuals to a minimum, and keeping the functionality fairly basic; you will have the opportunity to improve on what we have done in the practical exercises at the end of the next chapter.

11.2 The Requirements Specification

The local university requires a program to manage one of its student hostels, which contains a number of rooms, each of which can be occupied by a single tenant who pays rent on a monthly basis. The program must keep a list of tenants and their monthly payments. The information held for each tenant will consist of a name, a room number and a list of all the payments a tenant has made (month and amount) for one year. The program must allow the user to add and delete tenants, to display a list of all tenants, to record a payment for a particular tenant, and to display the payment history of a tenant.

11.3 The Design

The two core classes required in this application are `Tenant` (to store the details of a tenant) and `Payment` (to store the details of a payment). We have made a number of design decisions about how the system will be implemented, and these are listed below:

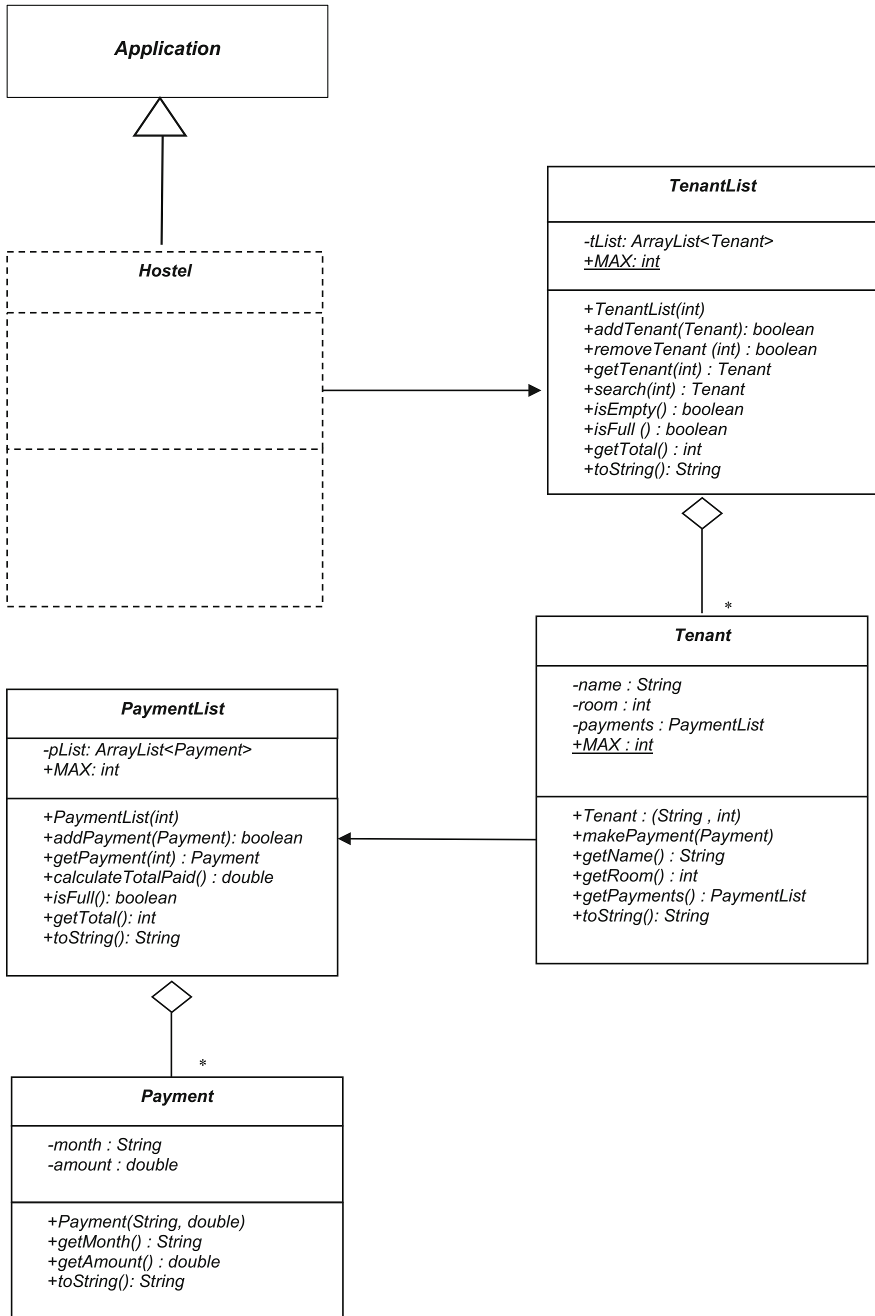


Fig. 11.1 The design of the student hostel system

- instances of the `Tenant` class and instances of the `Payment` class will each be held in a separate collection class, `PaymentList` and `TenantList` respectively;
- the collection classes `PaymentList` and `TenantList` both make use of an `ArrayList`;
- the `Hostel` class, which will hold the `TenantList`, will also act as the graphical interface for the system.

The design of the system is shown in Fig. 11.1. In this design there are two arrows from one class to another. In UML these represent **associations**. An association is a link from objects of one class to objects of another class. For example, a *customer* might have one or more *accounts*; a *student* might have one or more *tutors*. The simplest form of association is a one-to-one relationship whereby a single instance of one class is associated with a single instance of another class—for example a *purchase transaction* and an *invoice*. You have already come across inheritance and aggregation—these are special examples of association.

In our example, the associations represented by the arrows are one-to-one associations—a `Tenant` requires a single instance of a `PaymentList` and a `Hostel` requires a single instance of a `TenantList`.

The `Hostel` class itself has not yet been designed and this will be left until the next chapter where we consider the overall system design and testing; for this reason it has been drawn with a dotted line, but we can see it is a graphics class as it inherits from the JavaFX `Application` class.

In order to implement this application we should start with those classes that do not depend on any other, so that they can be unit tested in isolation. For example, we should not start by implementing the `Tenant` class as it requires the `PaymentList` class to be implemented first. You can see from the associations in Fig. 11.1 that the only class that does not require any other class for its implementation is the `Payment` class.

11.4 Implementing the *Payment* Class

Throughout this case study we will make use of the `Javadoc` style of comments (that we briefly mentioned back in Chap. 1) to document our classes. We will discuss how to read and write `Javadoc` comments in more detail in the next section.

The code for the `Payment` class is shown below.

Payment

```
/** Class used to store details of a single payment in a hostel
 *  @author Charatan and Kans
 *  @version 6th April 2018
 */
public class Payment
{
   private String month;
   private double amount;

   /** Constructor initialises the payment month and the amount paid
    *  @param  monthIn: month of payment
    *  @param  amountIn: amount of payment
    */
   public Payment(String monthIn, double amountIn)
   {
           month = monthIn;
           amount = amountIn;
   }

   /** Reads the month for which payment was made
    *  @return Returns the month for which payment was made
    */
   public String getMonth()
   {
           return month;
   }

   /** Reads the amount paid
    *  @return Returns the amount paid
    */
   public double getAmount()
   {
           return amount;
   }

   @Override
   public String toString()
   {
           return "("  + month + ", " + amount  + ")";
   }

}
```

As you can see, this class is fairly simple and does not require much explanation. Note that we have overridden the `toString` method (hence the `@Override` tag) to provide a convenient way of printing a `Payment` object (as discussed in Chap. 9).

```
@Override
public String toString()
{
       return "(" + month + " : " + amount + ")"; // a convenient way of displaying attributes
}
```

Before incorporating this class into a larger program you would to test if it was working reliably. Eventually, when this class is incorporated into the final program we will have a JavaFX `Hostel` class to run the application, but we need to test this class before an entire suite of classes has been developed. As we said before, testing an individual class in this way is often referred to as *unit testing*.

In order to unit test this class we will need to implement a separate class especially for this purpose. This new class will contain a `main` method and it will act as a **driver** for the original class. A driver is a special program designed to do

nothing except exercise a particular class. If you look back at all our previous examples, this is exactly how we tested individual classes. Initially you should generate an object from the given class. Once an object has been generated we can then test that object by calling its methods. When testing your class by generating objects and calling methods, you will want to display results on the screen, such as the data stored within your object. We could access this data by calling the appropriate `get` methods:

```
public class PaymentTester
{

   public static void main(String[ ] args)
   {
      Payment p1 = new Payment ("January", 175);

      // code to interrogate object data
      System.out.println("Month: " + p1.getMonth());
      System.out.println("Amount: " + p1.getAmount());
   }
}
```

This will display the expected output:

```
Month: January
Amount: 175.0
```

While having multiple output statements like this might be necessary in the final application, it is a rather cumbersome way of retrieving information from an object during the testing phase—when you will not be so concerned with the format of the output. This is where we can make use of the `toString` method we provided in the `Payment` class to display all attributes of the class in one print statement. Here is a modified tester:

PaymentTester

```
// a very simple driver program that makes use of the toString method

public class PaymentTester
{

   public static void main(String[ ] args)
   {
      Payment p1 = new Payment ("January", 175); // create object to test

      System.out.println(p1); // this will call the toString method in our Payment class
   }
}
```

Running this program will produce the following result:

`(January : 175.0)`

You can see the `Payment` object is displayed in the format given in our `toString` method. From this example, you can see how useful the `toString` method is. Now let's move on to the more interesting parts of this system. This

system requires us to develop two kinds of list, a PaymentList and a TenantList. Both make use of ArrayList to store the respective collections. A TenantList requires a PaymentList class to be developed first, so let's start by looking at this PaymentList class.

11.5 The *PaymentList* Class

The design of the PaymentList class is similar to a collection class that we showed you in Chap. 8—the Bank class. Both classes make use of an ArrayList to store a collection of objects. The main difference between the two classes is that in the Bank class the contained type was BankAccount, whereas in the PaymentList class the contained type is Payment. Also, we have included a constant MAX, to allow us to record the maximum number of payments we would like to record in our list. As MAX is a constant it has been given public visibility (+). Figure 11.2 provides a reminder of the design of the PaymentList class.

As you can see, as well as a constructor, there are methods to add a new payment to the list, to check if the list is full and to count the total number of payments made so far. There are also methods to get a payment based on a position number and to calculate the total payments made so far. Finally, once again we have included a toString method for ease of testing. Take a look at the code for the PaymentList class below before we discuss it—once again we are using the Javadoc style of comments which will be fully explained in the next section.

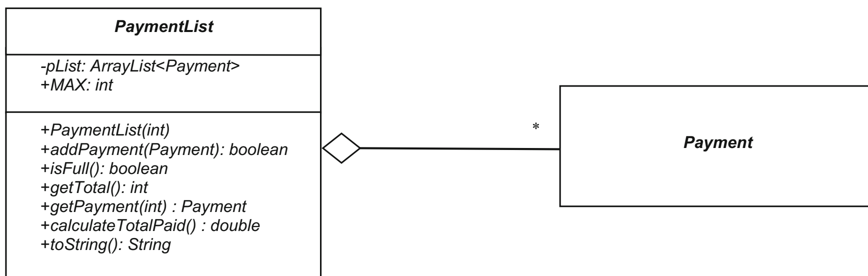


Fig. 11.2 The design of the PaymentList collection class

PaymentList

```
import java.util.ArrayList;

/** Collection class to hold a list of Payment objects
 *  @author Charatan and Kans
 *  @version 4th April 2018
 */
public class PaymentList
{
   // attributes
   private ArrayList<Payment> pList;
   public final int MAX;

   /** Constructor initialises the empty payment list and sets the maximum list size
    *  @param   maxIn: The maximum number of payments in the list
    */
    public PaymentList(int maxIn)
    {
           pList = new ArrayList<>();
           MAX = maxIn;
    }

   /** Checks if the payment list is full
    *  @return Returns true if the list is full and false otherwise
    */
    public boolean isFull()
    {
           return pList.size()== MAX;
    }

   /** Gets the total number of payments
    *  @return Returns the total number of payments currently in the list
    */
    public int getTotal()
    {
           return pList.size();
    }

   /** Adds a new payment to the end of the list
     *  @param  pIn: The payment to add
     *  @return Returns true if the object was added successfully and false otherwise
     */
    public boolean addPayment(Payment pIn)
    {
       if(!isFull())
       {
             pList.add(pIn);
             return true;
       }
       else
       {
             return false;
       }
    }

   /** Reads the payment at the given position in the list
     *  @param     positionIn: The logical position of the payment in the list
     *  @return    Returns the payment at the given logical position in the list
     *             or null if no payment at that logical position
     */
    public Payment getPayment(int positionIn)
    {
       //check for valid logical position
       if (positionIn <1 || positionIn > getTotal())
       {
             // no object found at given position
             return null;
       }
       else
       {
             // take one off logical position to get ArrayList position
            return pList.get(positionIn - 1);
       }
    }

    /** Calculates the total payments made by the tenant
     *  @return  Returns the total value of payments recorded
     */
    public double calculateTotalPaid()
    {
       double totalPaid = 0; // initialize totalPaid
       // loop through all payments
       for (Payment p: pList)
       {
             // add current payment to running total
             totalPaid = totalPaid + p.getAmount();
       }
       return totalPaid;
    }

   @Override
    public String toString()
    {
           return pList.toString();
    }
}
```

Firstly you can see that we have imported the `ArrayList` class from the `java.util` package:

```
import java.util.ArrayList;
```

We make use of the `ArrayList` class to store a collection of payments in the `pList` attribute. In addition to this attribute we have a **`public`** constant value `MAX` to record the maximum number of payments that we can record:

```
// attributes
private ArrayList<Payment> pList;
public final int MAX;
```

The constructor, as well as initialising the `ArrayList` attribute `pList`, sets the value for `MAX` via a parameter (`maxIn`) sent to the constructor:

```
/** Constructor initialises the empty payment list and sets the maximum list size
 *  @param   maxIn: The maximum number of payments in the list
 */
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       MAX = maxIn;
}
```

Now let's take a look at the remaining methods of this class. Firstly, we have an `isFull` method. The `pList` will be full when its size is equal to the value of our constant `MAX`:

```
public boolean isFull()
{
    return pList.size()== MAX; // use the size method of ArrayList
}
```

You can see that the `size` method of `ArrayList` has been used to check the number of items currently in the `pList`.

The `addPayment` method makes use of `isFull` to check if there is space in our `pList` before adding a new `Payment` object. A **`boolean`** value is returned to indicate success or failure:

```
public boolean addPayment(Payment pIn)
{
   if(!isFull()) // ok to add Payment
   {
      pList.add(pIn);
      return true;
   }
   else
   {
      return false; // Payment not added to a full list
   }
}
```

The `size` method of `ArrayList` that we met earlier is used again to implement the `getTotal` method, which returns the total number of payments made so far:

```
public int getTotal()
{
    return pList.size();
}
```

The `getPayment` method makes use of `getTotal` to check that the validity of the parameter `positionIn`. The `positionIn` parameter is the logical position of a payment in the list, which should be a number between 1 and the total number of items in the list. If an invalid position is sent the **`null`** value is returned, otherwise the payment at the associated `ArrayList` position is returned:

```
public Payment getPayment(int positionIn)
{
    //check for invalid logical position
    if (positionIn <1 || positionIn > getTotal())
    {
        // no object to return at the position
        return null;
    }
    else
    {
        // take one off logical position to get ArrayList position
        return pList.get(positionIn - 1);
    }
}
```

Next, the `calculateTotalPaid` method computes the sum of all payments in the list. This `calculateTotalPaid` method uses a standard algorithm for computing sums from a list of items. We met such an algorithm in Sect. 6.8.2 of this book. This algorithm can be expressed in pseudocode as follows:

```
SET totalPaid TO 0
LOOP FROM first item in list TO last item in list
BEGIN
    SET totalPaid TO totalPaid + amount of current payment
END
return totalPaid
```

Since the loop in this algorithm is just reading the items in the payment list, it can be implemented in Java with the use of an enhanced **`for`** loop:

```
public double calculateTotalPaid()
{
    double totalPaid = 0; // initialize totalPaid
    // loop through all payments
    for (Payment p: pList)
    {
        // add current payment to running total
        totalPaid = totalPaid + p.getAmount();
    }
    return totalPaid; // return sum
}
```

Finally, we have overridden the `toString` method again to allow the payment list to be displayed as a single `String`. This may seem like quite a challenging task when dealing with a collection of objects. Maybe we need to loop through the

list and join all the payments together to form a single String? Luckily, we do not have to go to such lengths as the ArrayList class has a toString method built in! So all we need to do is call the toString method of the ArrayList attribute here:

```
@Override
public String toString()
{
   return pList.toString(); // call toString of ArrayList

}
```

If we assume we have three payment objects in the payment list, say (“Jan” : 310), (“Feb” : 280) and (“March” : 310), the toString method of ArrayList would return a String that looks as follows:

[(“Jan” : 310), (“Feb” : 280), (“March” : 310)]

As you can see, the toString method of ArrayList calls the toString method of the contained items, separates them by commas and encloses the whole list in a pair of square brackets.

We will make use of this toString method during the testing of the PaymentList class, but first let’s have a closer look at the Javadoc comments that we have included in the PaymentList class and the Javadoc tool itself.

11.5.1 Javadoc

Oracle’s Java Development Kit contains a tool, Javadoc, which allows you to generate documentation for classes in the form of HTML files. In order to use this tool you must comment your classes in the Javadoc style. As we mentioned in Chap. 1, Javadoc comments must begin with /** and end with */. Javadoc comments can also contain ‘tags’. Tags are special formatting markers that allow you to record information such as the author of a piece of code. Table 11.1 gives some commonly used tags in Javadoc comments.

The @author and @version tags are used in the Javadoc comments for the class as a whole. You can see examples of these tags at the top of the PaymentList class:

```
/** Collection class to hold a list of Payment objects
 *  @author Charatan and Kans
 *  @version 4th April 2018
 */
public class PaymentList
{
    // attributes and methods go here
}
```

When Javadoc comments run over several lines, as in the example above, it is common (though not necessary) to begin each line with a leading asterisk.

Table 11.1 Some Javadoc tags

Tag	Information
@author	The name(s) of the code author(s)
@version	A version number for the code (often a date is used here)
@param	The name of a parameter and its description
@return	A description of the return value of a method

The @param and @return tags can be used in the Javadoc comments preceding each method. The @param tag is used to name and describe the purpose of a given parameter. The @return tag is used to describe the value returned by a method. Here for example are the Javadoc comments for the addPayment method, which makes use of both the @param and @return tags.

```
/** Adds a new payment to the end of the list
 *  @param  pIn: The payment to add
 *  @return Returns true if the object was added successfully and false otherwise
 */
public boolean addPayment(Payment pIn)
{
   if(!isFull())
   {
      pList.add(pIn);
      return true;
   }
   else
   {
      return false;
   }
}
```

The @param tag has been used to provide a comment on the parameter (pIn) and the @return tag has been used to comment the role of the **boolean** value returned by this method. Using Javadoc comments in this way provides a comprehensive explanation of the functionality of a class. Take a look at the remaining Javadoc comments used in the PaymentList class for more examples of these tags.

The Javadoc HTML documentation files themselves can then be generated either from the command line using the **javadoc** command:

javadoc PaymentList.java

or invoked directly by your IDE. Figure 11.3 gives part of the documentation generated as a result of the PaymentList class Javadoc comments.

Comments, such as the Javadoc comments we added into the PaymentList class, provide one technique for documenting the code that you write. Documenting your code is important as it assists in the maintenance of that code, should it need to be modified in the future. Well documented code is also easier to fix if errors arise during development. As well as commenting your code, you should ensure that the code is well laid out so that it becomes easier to read and follow; more about this in a moment.

Class PaymentList

java.lang.Object
PaymentList

```
public class PaymentList
extends java.lang.Object
```

Collection class to hold a list of Payment objects

Field Summary

Fields

Modifier and Type	Field and Description
int	MAX

Constructor Summary

Constructors

Constructor and Description
PaymentList(int maxIn) Constructor initialises the empty payment list and sets the maximum list size

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	addPayment(Payment pIn) Adds a new payment to the end of the list
double	calculateTotalPaid() Calculates the total payments made by the tenant
Payment	getPayment(int positionIn) Reads the payment at the given position in the list
int	getTotal() Gets the total number of payments
boolean	isFull() Checks if the payment list is full
java.lang.String	toString()

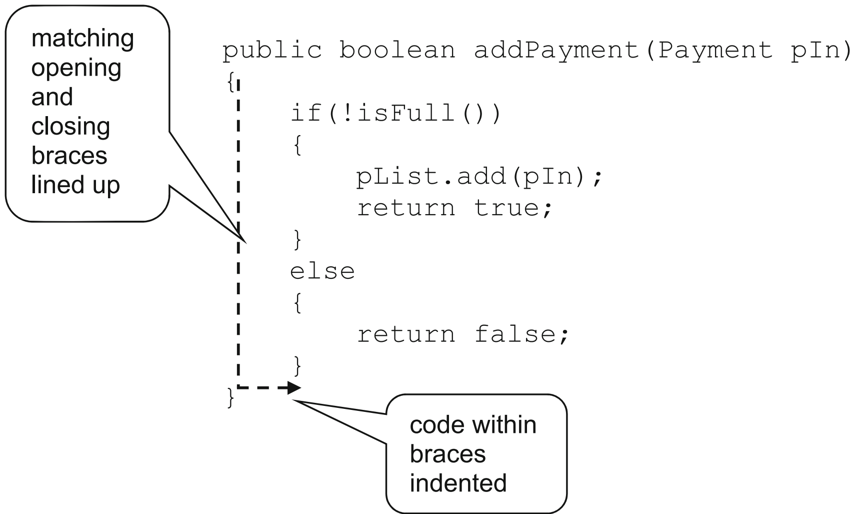
Fig. 11.3 `Javadoc` documentation generated for the `PaymentList` class

11.5.2 Code Layout

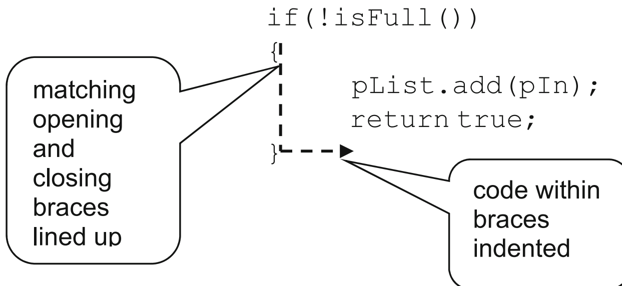
Consistent and clear indentation is important to improve the readability of your programs. Look at the example programs that we have presented to you and notice the care we have taken with our indentation. We are following two simple rules all the time:

- keep braces lined up under the structure to which they belong;
- indent, by one level, all code that belongs within those braces.

For example, look again at the `addPayment` method of the `PaymentList` class:



Notice how these rules are applied again with the braces of the inner **`if...else`** statements:



11.6 Testing the *PaymentList* Class

As we have said before, it is always important to test classes in order to ensure that they are functioning correctly before moving on to the rest of the development. Whereas the testing of the `Payment` class was an example of *unit testing*, testing this `PaymentList` class is an example of *integration testing* as it requires the `PaymentList` class working in conjunction with the `Payment` class.

To test the `PaymentList` class we need a driver that not only creates a `PaymentList` object, but also creates payments to add to this list.

We have quite a few methods to test in our `PaymentList` class, so we need to spend some time considering how we will go about testing these methods. For example, it would make sense to limit the size of our `PaymentList` so that we

can quickly fill up our list to check the isFull method. Here is one possible test strategy:

1. limit the size of the PaymentList to a relatively small number (say 4) using the PaymentList constructor;
2. add two payments to this list, say ("Jan", 310) and ("Feb", 280) using the addPayment method;
3. display the list (using the toString method) to check the items have been added successfully;
4. check to see if the isFull method returns **false**;
5. add two more payments to this list, say ("March", 310) and ("April", 300) using the addPayment method;
6. display the list (using the toString method) to check the items have been added successfully;
7. check to see if the isFull method returns **true**;
8. get details of one of the payments made (say the second payment) using the getPayment method.
9. attempt to retrieve a payment at an invalid position (say 5);
10. display the total number of payments made so far (using the getTotal method);
11. display the total of the payments made so far (using the calculateTotalPaid method);
12. attempt to add another payment to this full list using the addPayment method.

Notice that the test strategy should ensure that all methods of the class are tested and all possible routes through a method are tested. So, for example, as well as ensuring that the isFull method is called ensure we provide a scenario where the method should return **true** and provide a scenario where the method should return **false**.

Once a strategy is chosen, the test results should be logged in a **test log**. A test log is a document that records the testing that took place during system development. Each row of the test log associates an *input* with an *expected output*. If the output is not as expected, reasons for this error have to be identified and recorded in the log.

Figure 11.4 illustrates a suitable test log to document the testing strategy we developed above.

Test logs such as this should be devised *before* the driver itself (and may even be developed before the class we are testing has been developed). The test log can then be used to prompt the development of the driver. As you can see by looking at the test in Fig. 11.4, we assume that the driver is a menu driven program.

TEST LOG			
Purpose: To test the PaymentList class			
Run Number:	**Date:**		
Action	**Expected Output**	**Pass/ Fail**	**Reason for failure**
-	Prompt for size of list		
Enter 4	Display menu of options		
Select ADD option	Prompt for Payment details		
Enter "Jan", 310	Display menu of options		
Select ADD option	Prompt for Payment to add		
Enter "Feb", 280	Display menu of options		
Select DISPLAY option	Message [(Jan : 310.0), (Feb : 280.0)] Display menu of options		
Select IS FULL option	Message "list is NOT full" Display menu of options		
Select ADD option	Prompt for Payment to add		
Enter "March", 310	Display menu of options		
Select ADD option	Prompt for Payment to add		
Enter "April", 300	Display menu of options		
Select DISPLAY option	Message [(Jan : 310.0), (Feb : 280.0), (March : 310.0), (April : 300.0)] Display menu of options		
Select IS FULL option	Message "list is full" Display menu of options		
Select GET PAYMENT option	Prompt for position to retrieve		
Enter 2	Message (Feb : 280.0) Display menu of options		
Select GET PAYMENT option	Prompt for position to retrieve		
Enter 5	Message INVALID PAYMENT NUMBER Display menu of options		
Select GET TOTAL option	Message 4 Display menu of options		
Select CALCULATE TOTAL PAID option	Message 1200.0 Display menu of options		
Select ADD option	Prompt for Payment to add		
Enter "May", 310	Message Error – list full message Display menu of options		
Select EXIT option	Program terminates		

Fig. 11.4 Test log for the `PaymentList` class

The `PaymentListTester` program is one possible driver we could develop in order to process the actions given in this test log:

PaymentListTester

```
public class PaymentListTester
{

    public static void main(String[] args)
    {
       char choice;
       int size;
       PaymentList list; // declare PaymentList object to test

       // get size of list
       System.out.print("Size of list? ");
       size = EasyScanner.nextInt();
       list = new PaymentList(size); // create object to test
       // menu
       do
       {
            // display options
            System.out.println();
            System.out.println("[1] ADD");
            System.out.println("[2] DISPLAY");
            System.out.println("[3] IS FULL");
            System.out.println("[4] GET PAYMENT");
            System.out.println("[5] GET TOTAL");
            System.out.println("[6] CALCULATE TOTAL PAID");
            System.out.println("[7] Quit");
            System.out.println();
            System.out.print("Enter a choice [1-7]: ");
            // get choice
            choice = EasyScanner.nextChar();
            System.out.println();
            // process choice
            switch(choice)
            {
                case '1': option1(list); break;
                case '2': option2(list); break;
                case '3': option3(list); break;
                case '4': option4(list); break;
                case '5': option5(list); break;
                case '6': option6(list); break;
                case '7': System.out.println("TESTING COMPLETE"); break;
                default: System.out.print("1-7 only");
            }
       } while (choice != '7');
    }

    // ADD
    static void option1(PaymentList listIn)
    {
        // prompt for payment details
        System.out.print("Enter Month: ");
        String month = EasyScanner.nextString();
        System.out.print("Enter Amount: ");
        double amount = EasyScanner.nextDouble();
        // create new Payment object from input
        Payment p = new Payment(month, amount);
        // attempt to add payment to list
        boolean ok = listIn.addPayment(p); // value of false sent back if unable to add
        if (!ok)// check if item was not added
        {
            System.out.println("ERROR: list full");
        }
    }

    // DISPLAY
    static void option2(PaymentList listIn)
    {
        System.out.println("ITEMS ENTERED");
        System.out.println(listIn); // calls toString method of PaymentList
    }

    // IS FULL
    static void option3(PaymentList listIn)
    {
        if (listIn.isFull())
        {
            System.out.println("list is full");
        }
        else
        {
            System.out.println("list is NOT full");
        }
    }

    // GET PAYMENT
    static void option4(PaymentList listIn)
    {
        // prompt for and receive payment number
        System.out.print("Enter payment number to retrieve: ");
```

```
        int num = EasyScanner.nextInt();
        // retrieve Payment object form list
        Payment p = listIn.getPayment(num); // returns null if invalid position
        if (p != null)// check if Payment retrieved
        {
            System.out.println(p); // calls toString method of Payment
        }
        else
        {
            System.out.println("INVALID PAYMENT NUMBER"); // invalid position error
        }
    }

    // GET TOTAL
    static void option5(PaymentList listIn)
    {
        System.out.print("TOTAL NUMBER OF PAYMENTS ENTERED: ");
        System.out.println(listIn.getTotal());
    }

    // GET TOTAL PAID
    static void option6(PaymentList listIn)
    {
        System.out.print("TOTAL OF PAYMENTS MADE SO FAR: ");
        System.out.println(listIn.calculateTotalPaid());
    }
}
```

We are now in a position to run the driver and check the actions documented in the test log:

```
Size of list? 4
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 1
Enter Month: Jan
Enter Amount: 310
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 1
Enter Month: Feb
Enter Amount: 280
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
```

```
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 2
ITEMS ENTERED
[(Jan : 310.0), (Feb : 280.0)]
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 3
list is NOT full
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 1
Enter Month: March
Enter Amount: 310
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 1
Enter Month: April
Enter Amount: 300
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
```

```
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 2
ITEMS ENTERED
[(Jan : 310.0), (Feb : 280.0), (March : 310.0), (April :
  300.0)]
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 3
list is full
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 4
Enter payment number to retrieve: 2
(Feb : 280.0)
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 4
Enter payment number to retrieve: 5
INVALID PAYMENT NUMBER
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
```

```
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 5
TOTAL NUMBER OF PAYMENTS ENTERED: 4
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 6
TOTAL OF PAYMENTS MADE SO FAR: 1200.0
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
  [7] Quit
Enter a choice [1-7]: 1
Enter Month: May
Enter Amount: 310
ERROR: list full
[1] ADD
[2] DISPLAY
[3] IS FULL
[4] GET PAYMENT
[5] GET TOTAL
[6] CALCULATE TOTAL PAID
[7] Quit
Enter a choice [1-7]: 7
TESTING COMPLETE
```

You have seen menu driven tester programmes such as this before so we will not discuss it in any detail. Note that our `EasyScanner` class has been used in this tester for ease of keyboard input. For example:

```
// get size of list
System.out.print("Size of list? ");
size = EasyScanner.nextInt(); // EasyScanner used to simplify keyboard input
```

Also, notice how the display method of our `PaymentListTester` (option 2 on the menu) displays the `PaymentList` object using the `toString` method we discussed earlier:

```
// DISPLAY
static void option2(PaymentList listIn)
{
    System.out.println("ITEMS ENTERED");
    System.out.println(listIn); // calls toString method of PaymentList
}
```

The output from this method gives us the `ArrayList` values in the format we discussed earlier, for example:

ITEMS ENTERED
[(Jan : 310.0), (Feb : 280.0), (March : 310.0), (April : 300.0)]

If unexpected results are produced during testing, you should stop and identify the cause of the error in the class that you are testing. Both the cause of the error and how the error was fixed should be documented in the test log. The driver can then be run again with a fresh test log and this process should continue until *all* results are delivered as predicted. In this case, however, the results were as expected, so we can now move on to developing the rest of our system. We have two more classes to look at `Tenant` and `TenantList`. Before we look at the `TenantList` class we need to implement the `Tenant` class.

11.7 Implementing the *Tenant* Class

As you can see from the UML diagram of Fig. 11.1, the `Tenant` class contains four attributes:

- `name`;
- `room`;
- `payments`;
- `MAX`.

The first two of these represent the name and the room of the tenant respectively. The third attribute, `payments`, is to be implemented as a `PaymentList` object and the last attribute, `MAX`, is to be implemented as a **`static`** class attribute. The `MAX` attribute will also be implemented as a *constant* as we are assuming that tenants make a *fixed* number of payments in a year (twelve—one for each month). Since class constants cannot be modified, it makes sense to allow them to be declared as **`public`**. Below is the code for the `Tenant` class.

Tenant

```
/** Class used to record the details of a tenant
 *  @author Charatan and Kans
 *  @version 6th April 2018
 */
public class Tenant
{
   private String name;
   private int room;
   private PaymentList payments;
   public static final int MAX = 12;

  /** Constructor initialises the name and room number of the tenant
    *  and sets the payments made to the empty list
    *  @param  nameIn: name of tenant
    *  @param  roomIn: room number of tenant
    */
   public Tenant(String nameIn, int roomIn)
   {
           name = nameIn;
           room = roomIn;
           payments = new PaymentList(MAX);
   }

  /** Records a payment for the tenant
   *  @param paymentIn: payment made by tenant
   */
   public void makePayment(Payment paymentIn)
   {
           payments.addPayment(paymentIn); // call PaymentList method
   }

  /** Reads the name of the tenant
   *  @return Returns the name of the tenant
   */
   public String getName()
   {
           return name;
   }

  /** Reads the room of the tenant
   *  @return Returns the room of the tenant
   */
   public int getRoom()
   {
           return room;
   }

  /** Reads the payments of the tenant
   *  @return Returns the payments made by the tenant
   */
   public PaymentList getPayments()
   {
           return payments;
   }

   @Override
   public String toString()
   {
           return name+", "+room +", "+payments;
   }
}
```

The Javadoc comments should be sufficient documentation for you to follow the code in this class. Note how the Javadoc comments for the constructor includes two @param tags, as the constructor has two parameters. Also, it is worth noting that the payments attribute, being of type PaymentList, can respond to any of the PaymentList methods we discussed in Sect. 11.5. The makePayment method illustrates this by calling the addPayment method of PaymentList:

```
public void makePayment(Payment paymentIn)
{
    payments.addPayment(paymentIn); // call PaymentList method
}
```

We will leave the testing of this class and the next TenantList class as exercises for you as end of chapter programming exercises.

11.8 Implementing the *TenantList* Class

The TenantList class is a collection class to hold our Tenant objects. Once again we use an ArrayList to store this collection and have a MAX constant to fix an upper limit on the number of tenants our hostel can accommodate. A reminder of the design of the TenantList class is given in Fig. 11.5.

Most of the methods of the TenantList class should be familiar to you from the PaymentList collection classed that we discussed earlier. We will just take a closer look at two methods that were not mirrored in the PaymentList class, namely the remove and search methods. Let's start with the search method. Here is a reminder of its UML interface:

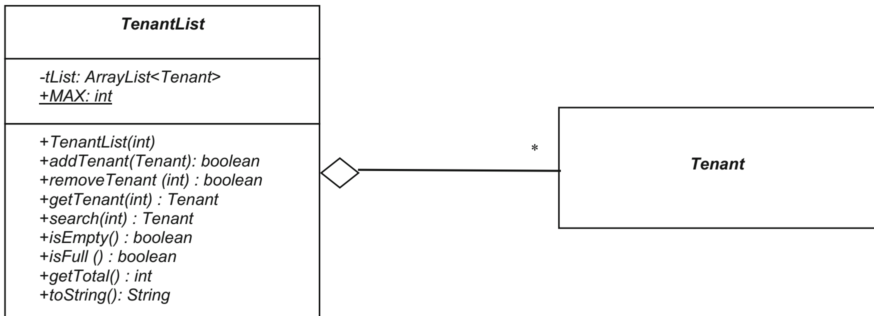


Fig. 11.5 The design of the TenantList collection class

search (int): Tenant

The integer parameter represents the room number of the tenant that this method is searching for. The tenant returned is the tenant living in that particular room; if no tenant is found in that room then **null** is returned. Here is a suitable algorithm for finding a tenant, expressed in pseudocode:

```
LOOP FROM first tenant in list TO last tenant in list
BEGIN
   IF current tenant's room number = room to locate
   BEGIN
      return current tenant
   END
END
return null
```

This is similar to the algorithm we looked at for searching an array in Sect. 8.8.1 except that we return the given item in the list rather than its index. This algorithm requires a loop to search through the tenants in the list and an enhanced **for** loop is a good way to do this:

```
public Tenant search(int roomIn)
{
   for(Tenant currentTenant: tList)// enhanced for loop to serach through the list of tenants
   {
      // find tenant with given room number
      if(currentTenant.getRoom() == roomIn)
      {
         return currentTenant;
      }
   }
   return null; // no tenant found with given room number
}
```

Now let's look at the `removeTenant` method. The UML interface for this method is as follows:

removeTenant(int): boolean

Here the integer parameter represents the room number of the tenant that is to be removed from the list and the **boolean** return value indicates whether or not such a tenant has been removed successfully.

The previous `search` method can be used here to determine if a tenant exists in that particular room (a value of **null** will be returned if no such tenant exists). If such a tenant does exist it can be removed from the list using the `remove` method

of `ArrayList` and a **`boolean`** value of **`true`** can be returned, otherwise a **`boolean`** value of **`false`** can be returned. Here is the code:

```
public boolean removeTenant(int roomIn)
{
   Tenant findT = search(roomIn); // call search method
   if (findT != null) // check tenant is found at given room
   {
      tList.remove(findT); // remove given tenant
      return true;
   }
   else
   {
      return false; // no tenant in given room
   }
}
```

The complete code for the `TenantList` class is now presented below. The `Javadoc` comments provided should now provide sufficient explanation of each part of this class.

TenantList

```
import java.util.ArrayList;

/** Collection class to hold a list of tenants
 *  @author Charatan and Kans
 *  @version 6th April 2018
 */
public class TenantList
{
    private ArrayList<Tenant> tList;
    public final int MAX;

    /** Constructor initialises the empty tenant list and sets the maximum list size
     *  @param   maxIn The maximum number of tenants in the list
     */
    public TenantList(int maxIn)
    {
        tList = new ArrayList<>();
        MAX = maxIn;
    }

    /** Adds a new Tenant to the list
     *  @param  tIn The Tenant to add
     *  @return Returns true if the tenant was added successfully and false otherwise
     */
    public boolean addTenant(Tenant tIn)
    {
        if(!isFull())
        {
            tList.add(tIn);
            return true;
```

```
        }
        else
        {
            return false;
        }
    }

    /** Removes the tenant in the given room number
     *  @param roomIn The room number to of the tenant to remove
     *  @return Returns true if the tenant is removed successfully or false otherwise
     */
    public boolean removeTenant(int roomIn)
    {
        Tenant findT = search(roomIn); // call search method
        if (findT != null) // check tenant is found at given room
        {
            tList.remove(findT); // remove given tenant
            return true;
        }
        else
        {
            return false; // no tenant in given room
        }
    }

    /** Searches for the tenant in the given room number
     *  @param roomIn The room number to search for
     *  @return Returns the tenant in the given room or null if no tenant in the given room
     */
    public Tenant search(int roomIn)
    {
        for(Tenant currentTenant: tList)
        {
            // find tenant with given room number
            if(currentTenant.getRoom() == roomIn)
            {
                return currentTenant;
            }
        }
        return null; // no tenant found with given room number
    }

    /** Reads the tenant at the given position in the list
     *  @param      positionIn The logical position of the tenant in the list
     *  @return     Returns the tenant at the given logical position in the list
     *              or null if no tenant at that logical position
     */
    public Tenant getTenant(int positionIn)
    {
        if (positionIn<1 || positionIn>getTotal()) // check for valid position
        {
            return null; // no object found at given position
        }
        else
        {
            // remove one frm logical poition to get ArrayList position
            return tList.get(positionIn -1);
        }
    }


   /** Reports on whether or not the list is empty
     *  @return Returns true if the list is empty and false otherwise
     */
    public boolean isEmpty()
    {
        return tList.isEmpty();
    }

    /** Reports on whether or not the list is full
     *  @return Returns true if the list is full and false otherwise
     */
    public boolean isFull()
    {
        return tList.size()== MAX;
    }

    /** Gets the total number of tenants
     *  @return Returns the total number of tenants currently in the list
     */
    public int getTotal()
    {
        return tList.size();
    }

    @Override
    public String toString()
    {
        return tList.toString();
    }
}
```

All that remains for us to do to complete our case study in the next chapter is to design, implement and test the `Hostel` class which will not only keep track of the tenants but will also act as the graphical user interface for the system.

11.9 Self-test Questions

1. Describe the class associations given in the UML design of Fig. 11.1.
2. Produce suitable `Javadoc` comments for the `Oblong` class from Chap. 8.
3. The test log of Sect. 11.5.3 did not include checks for the `getItem` and `getTotal` methods of the `PaymentList` class. It also did not include a check that attempts to add to a full list and remove from an empty list would fail. Modify the test log to include these checks.
4. Develop test logs for testing the `Tenant` and `TenantList` classes.
5. Identify the benefits of adding a `toString` method into your classes and then write a suitable `toString` method for the `Bank` class from Chap. 8.

11.10 Programming Exercises

You will need to copy the entire suite of classes that make up the student hostel system and the Bank and BankApplication classes from the website.

1. Modify and then run the driver given in Program 11.2 in light of the changes made to the test log in self-test question 3 above.
2. Develop suitable drivers to test the `Tenant` and `TenantList` classes.
3. Use the test logs you developed in self-test question 5 and the drivers you developed in Exercise 2 above to test the `Tenant` and `TenantList` classes.
4. Incorporate the `toString` method into the `Bank` class from Chap. 8 you developed in self-test question 4 above then modify and run the `BankApplication` tester program from the same chapter to test make use of this `toString` method.

Check for updates

Case Study—Part 2

12

Outcomes:

By the end of this chapter you should be able to:

- *design an attractive graphical user interface*;
- *use pseudocode to design event handling routines*;
- *implement the design in Java using a variety of JavaFX components*;
- *devise a testing strategy for a complete application and carry out the necessary steps to implement that strategy.*

12.1 Introduction

In the previous chapter we designed and developed the core classes required to implement the functionality of our *Student Hostel System*. We now go on develop a graphical user interface for this application.

12.2 Keeping Permanent Records

In practice, an application such as the *Student Hostel System* would not be much use if we had no way of keeping permanent records—in other words, of saving a file to disk. However, reading and writing files is something that you will not learn until your second semester. So, in the meantime, in order to make it possible to keep a permanent record of your data, we have created a special class for you to use; we have called this class `TenantFileHandler`. This class (along with the rest of the files from this case study) can be found on the accompanying website.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_12

The `TenantFileHandler` class has two **`static`** methods: the first, `saveRecords`, needs to be sent two parameters, an integer value indicating the number of rooms in the hostel, and a `TenantList`, which is a reference to the list to be saved; the second, `readRecords`, requires only a reference to a `TenantList` so that it knows where to store the information that is read from the file.

The `readRecords` method will be called when the application starts (so this method call will therefore be coded into the `start` method of the JavaFX application), and the `saveRecords` method will be called when we finish the application (and will therefore be coded into the event-handler of a “Save and Quit” button). The user can of course exit without saving by clicking the cross-hairs, just in case, for any reason, the user should want to abandon any changes.

12.3 Design of the *Hostel* Class

In Fig. 11.1 of the previous chapter we presented the `Hostel` class as part of the Student Hostel application design, but did not give any design details for this class. Let’s consider this `Hostel` class now. Figure 12.1 is a reminder of the important classes in the UML diagram and also includes some key attributes and methods.

As you can see, the `Hostel` class requires a single instance of the `TenantList` collection class we developed in Chap. 11. In Fig. 12.1 this instance is recorded as the private `list` attribute in the `Hostel` class.

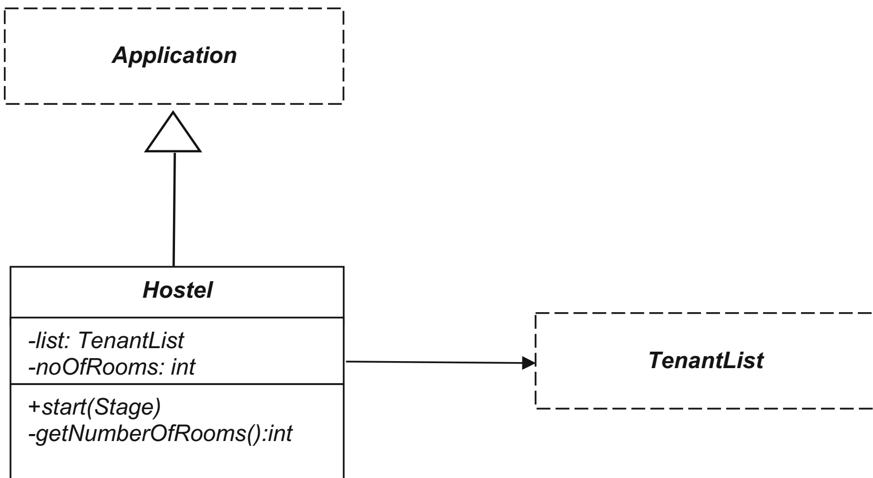


Fig. 12.1 The initial design of the *Hostel* class

Our Student Hostel application will have a JavaFX interface, so the Hostel class also needs to inherit from the JavaFX Application class. Consequently, we need to provide a start method to add components to the Stage and process the event-handling routines. This method will also initialise the tenant list and read any data into this list from a file (using the aforementioned TenantFileHandler class that we have provided).

In order to initialise the tenant list we have added a noOfRooms integer attribute to record how many rooms to limit the hostel to. We have also added a private method, getNumberOfRooms, to request this room limit from the user. We have included a main method to launch the application (although, as we explained in Chap. 10, this is not always necessary for JavaFX applications).

From this initial class design we have the following outline of our Hostel class:

```
import javafx.application.Application;
import javafx.stage.Stage;

public class Hostel extends Application
{
    // attributes
    private TenantList list;
    private int noOfRooms;

    // methods

    @Override
    public void start(Stage stage)
    {
        noOfRooms = getNumberOfRooms(); // call private method
        // initialise tenant list
        list  = new TenantList(noOfRooms);
        TenantFileHandler.readRecords(list);
        // code to layout components, process event handling routines and initialise the list here
    }

    /**
     * Method to request number of hostel rooms from the user
     * @return number of rooms
     */
    private int getNumberOfRooms( )
    {
        System.out.print("How many rooms?: ");
        int num = EasyScanner.nextInt();
        return num;
    }


    public static void main(String[] args)
    {
        launch(args);
    }
}
```

You can see, in the start method, we have called the **private** getNumberOfRooms method to request and return the number of rooms to limit the tenant list to. The getNumberOfRooms method uses a standard text window for output and the keyboard for input. The readRecords method of the TenantFileHandler class is then used in the start method to load into the tenant list any pre-existing Tenant records saved to file.

The remaining code within this class will relate to the GUI for this application, so let's consider the design of the GUI now.

12.4 Design of the GUI

There will be two aspects to the design of the graphical interface. Firstly, we need to design the visual side of things; then we need to design the algorithms for our event-handling routines so that the buttons do the jobs we want them to, like adding or displaying tenants.

Let's start with the visual design. We need to choose which graphics components we are going to use and how to lay them out. One way to do this is to make a preliminary sketch such as the one shown in Fig. 12.2.

It should be clear which `JavaFX` components we will be using here. We have a selection of buttons on our GUI (Fig. 12.3).

The remaining components are `Labels` ("Hostel Application", "Room" "Name", "Month" and "Amount") and `TextFields` (for the five single-row boxes) or `TextAreas` (for the two multiple-row boxes).

In the examples you saw in Chap. 10, you saw that we created our visual components within our `start` method. This made sense as the algorithms for our button event handlers (which needed access to these components) were also contained within the `start` method. In this application, however, we would expect to have much more complicated algorithms for our button event handlers, so we will structure things a little differently. In this `Hostel` class we will implement our event handlers in a series of **`private`** methods (one for each button).

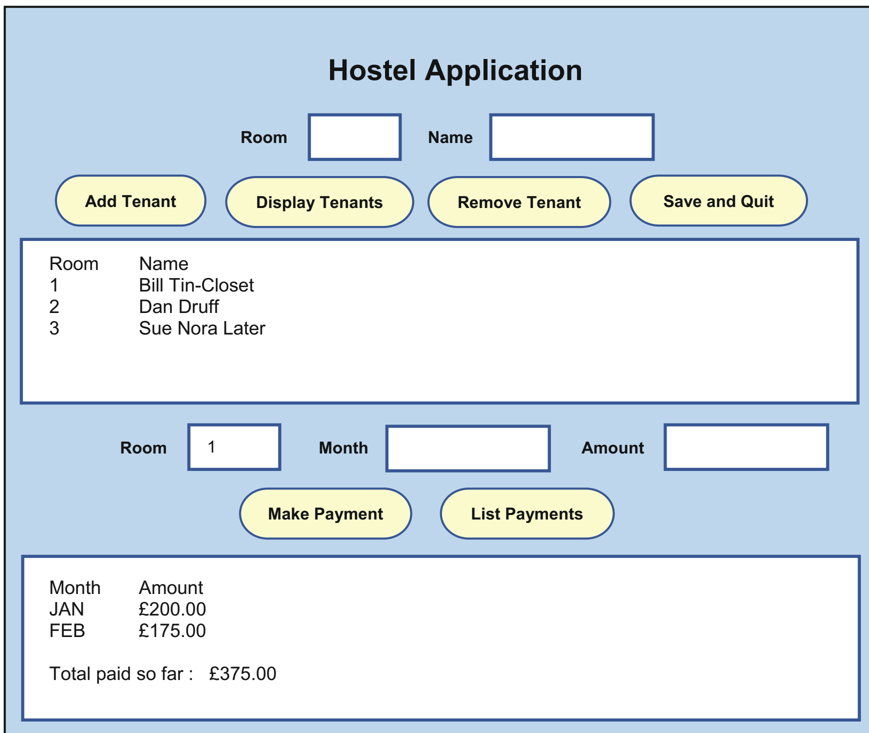


Fig. 12.2 Preliminary design of the *Hostel* GUI

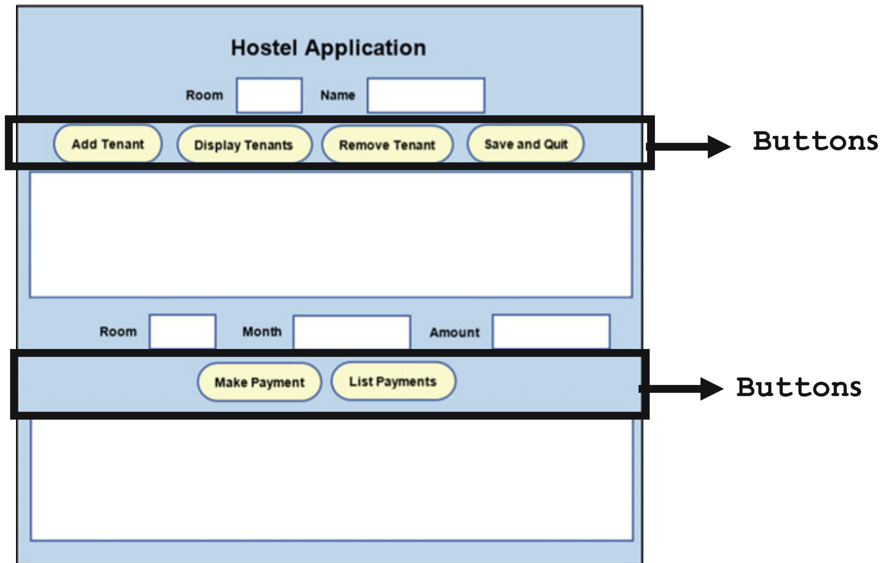


Fig. 12.3 Some graphical components in the *Hostel* GUI

One implication of implementing the event handling code in separate methods is that it makes more sense now to make our visual components accessible *throughout* the class rather than just in the `start` method. To do that we make these visual components *attributes* of the `Hostel` class. In the code snippet below we create these component attributes in the order they appear in our GUI design of Fig. 12.2 (from top to bottom and left to right):

```
public class Hostel extends Application
{
    // previous attributes here

   // visual components declared as attributes of the class
   private Label headingLabel = new Label("Hostel Application");
   private Label roomLabel1 = new Label("Room");
   private TextField roomField1 =  new TextField();
   private Label nameLabel = new Label("Name");
   private TextField nameField =  new TextField();
   private Button addButton = new Button("Add Tenant");
   private Button displayButton =  new  Button("Display Tenants");
   private Button removeButton  = new Button("Remove Tenant");
   private Button saveAndQuitButton  = new Button("Save and Quit");
   private TextArea displayArea1  = new TextArea();
   private Label roomLabel2 = new Label("Room");
   private TextField roomField2  = new TextField();
   private Label monthLabel = new Label("Month");
   private TextField monthField  = new TextField();
   private Label amountLabel = new Label("Amount");
   private TextField amountField =  new TextField();
   private Button paymentButton  = new Button("Make Payment");
   private Button listButton  = new Button("List Payments");
   private TextArea displayArea2 =  new TextArea();

   // code for methods goes here
}
```

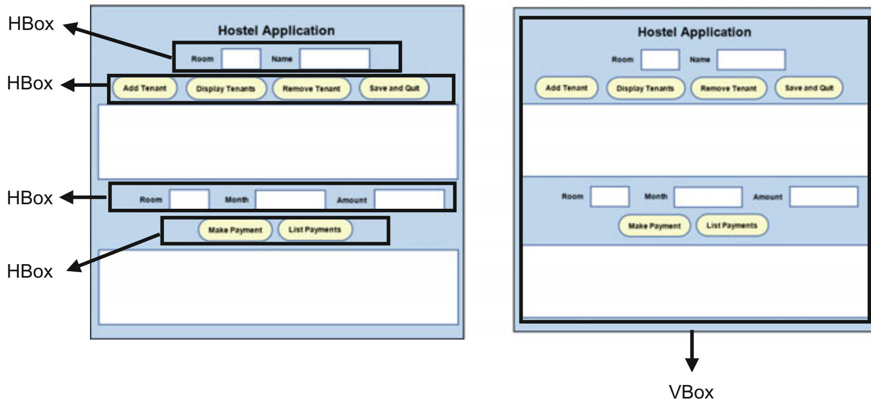


Fig. 12.4 Organizing the *Hostel* GUI with HBoxes and a VBox

Now let's turn to the layout of these components. We will use a mixture of `HBox`es and a `VBox` to organise our layout (see Fig. 12.4).

As always, we use the `start` method to organise our layout:

```
public void start(Stage stage)
{
   // previous code here

   // create four HBoxes
   HBox roomDetails = new HBox (10);
   HBox tenantButtons = new HBox(10);
   HBox paymentDetails = new HBox(10);
   HBox paymentButtons = new HBox(10);
   // add components to HBoxes
   roomDetails.getChildren().addAll(roomLabel1, roomField1, nameLabel, nameField);
   tenantButtons.getChildren().addAll(addButton, displayButton, removeButton, saveAndQuitButton);
   paymentDetails.getChildren().addAll(  roomLabel2, roomField2, monthLabel, monthField,
                                         amountLabel, amountField);
   paymentButtons.getChildren().addAll(paymentButton, listButton);
   // create VBox
   VBox root = new VBox(10);
   // add components to VBox
   root.getChildren().addAll(  headingLabel, roomDetails, tenantButtons, displayArea1,
                               paymentDetails, paymentButtons, displayArea2);
   // add the VBox to the Scene
   Scene scene = new Scene(root, Color.LIGHTBLUE);


   // rest of start method here
}
```

We have created the four `HBox`es given in Fig. 12.4 and then added the relevant components, we then do the same for the `VBox`. Finally, we add the `VBox` to the `Scene`.

Eventually we will also customise the look of our visual components (by setting fonts and borders for example) when we present the complete code for this class. But now let's turn our attention to designing the event-handlers for the buttons on our GUI.

12.5 Designing the Event-Handlers

As you saw in Fig. 12.2, there are six buttons that need to be coded so that they respond in the correct way when pressed:

- the “Add Tenant” button;
- the “Display Tenants” button;
- the “Remove Tenant” button;
- the “Save and Quit” button;
- the “Make Payment” button;
- the “List Payments” button.

As always, we will use the `setOnAction` method of each button to process these button clicks, but (as we said in the previous section) we will place the code for the event-handlers in separate **`private`** methods and call these methods from our lambda expressions:

```
public void start(Stage stage)
{
   // previous code here

   // call private methods for button event handlers
   addButton.setOnAction(e -> addHandler());
   displayButton.setOnAction(e -> displayHandler() );
   removeButton.setOnAction( e -> removeHandler());
   paymentButton.setOnAction( e -> paymentHandler());
   listButton.setOnAction( e -> listHandler());
   saveAndQuitButton.setOnAction( e -> saveAndQuitHandler());

   // rest of start method here
}

// private event handler methods here
```

We have summarized below the task that each button’s event-handler method must perform, and then gone on to design our algorithms using pseudocode.

The Add Tenant Button

The purpose of this button is to add a new `Tenant` to the list. The values entered in `roomField1` and `nameField` must be validated; first of all, they must not be blank; second, the room number must not be greater than the number of rooms available (or less than 1!); finally, the room must not be occupied. If all this is okay, then the new tenant is added (we will make use of the `addTenant` method of `TenantList` to do this) and a message should be displayed in `displayArea1`. We can express this in pseudocode as follows:

```
read roomField1
read nameField
IF roomField1 blank OR nameField blank
   display blank field error in displayArea1
ELSE IF roomField1 value < 1 OR roomField1 value > noOfRooms
   display invalid room number error in displayArea1
ELSE IF tenant found in room
   display room occupied error in displayArea1
ELSE
BEGIN
   add tenant
   blank roomField
   blank nameField
   display message to confirm success in displayArea1
END
```

The Display Tenants Button

Pressing this button will display the full list of tenants (room number and name) in `displayArea1`.

If all the rooms are vacant a suitable message should be displayed; otherwise the list of tenants' rooms and names should appear under appropriate headings as can be seen in Fig. 12.2. This can be expressed in pseudocode as follows:

```
IF list is empty
    display rooms empty error in displayArea1
ELSE
BEGIN
    display header in displayArea1
    LOOP FROM first item TO last item in list
    BEGIN
        append tenant room and name to displayArea1
    END
END
```

The Remove Tenant Button

Clicking on this button will remove the tenant whose room number has been entered in `roomField1`.

As with the *Add Tenant* button, the room number entered must be validated; if the number is a valid one then the tenant is removed from the list (we will make use of the `remove` method of `TenantList` to do this) and a confirmation message is displayed. The pseudocode for this event-handler is given as follows:

```
read roomField1
IF roomField1 blank
    display blank field error in displayArea1
ELSE IF roomField1 value < 1 OR roomField1 value > noOfRooms
    display invalid room number error in displayArea1
ELSE IF no tenant found in room
    display room empty error in displayArea1
ELSE
BEGIN
    remove tenant from list
    display message to confirm success in displayArea1
END
```

The List Payment Button

This button records payments made by an individual tenant whose room number is entered in `roomField2`. The values entered in `roomField2`, `monthField` and `amountField` must be validated to ensure that none of the fields are blank, that the room number is a valid one and, if so, that it is currently occupied.

If everything is okay then a new payment record is added to that tenant's list of payments (we will make use of the `makePayment` method of `PaymentList` to do this) and a confirmation message is displayed in `displayArea2`. This design is expressed in pseudocode as follows:

```
read roomField2
read monthField
read amountField
IF roomField2 blank OR monthField blank OR amountField blank
    display fields empty error in displayArea2
ELSE IF roomField2 value < 1 OR roomField2 value > noOfRooms
    display invalid room number error in displayArea2
ELSE IF no tenant found in room
    display room empty error in displayArea2
ELSE
BEGIN
    create payment from amountField value and monthField value
    add payment into list
    display message to confirm success in displayArea2
END
```

The List Payments Button

Pressing this button causes a list of payments (month and amount) made by the tenant whose room number is entered in `roomField2` to be displayed in `displayArea2`.

After validating the values entered, each record in the tenant's payment list is displayed. Finally, the total amount paid by that tenant is displayed (we will make use of the `calculateTotalPaid` method of `PaymentList` to do this). The pseudocode is given as follows:

```
read roomField2
IF roomField2 blank
display room field empty error in displayArea2
ELSE IF roomField2 value < 1 OR roomField2 value > noOfRooms
    display invalid room number error in displayArea2
ELSE IF no tenant found in room
    display room empty error in displayArea2
ELSE
BEGIN
    find tenant in given room
    get payments of tenant
    IF payments = 0
        display no payments error in displayArea2
    ELSE
    BEGIN
        display header in displayArea2
        LOOP FROM first payment TO last payment
        BEGIN
            append amount and month to displayArea2
        END
        display total paid in displayArea2
        blank monthField
        blank amountField
    END
END
```

The Save and Quit Button

Pressing this button causes all the records to be saved to a file (here we make use of the `saveRecords` method of the `TenantFileHandler` class that we talked about in Sect. 12.2); it then closes the application, terminating the program.

It will only contain a few lines of code and we have therefore not written pseudocode for it.

12.6 Implementing the *Hostel* Class

The complete code for the `Hostel` class now appears below. When you see the code, you should notice that we have utilized the `NumberFormat` class (which is to be found in the `java.text` package) to print the amounts in the local currency. Also note the use of two constants, `WIDTH` and `HEIGHT`, to help size our visual components and the `parseInt` method of the `Integer` class to convert the room values, entered as text, into integer values. We have also enhanced our visual components by making use of borders and backgrounds as discussed in Chap. 10.

Study the code and the comments carefully (in particular compare the event-handling code to the pseudocode we presented in the previous section) to make sure you understand it and we will explain the new concepts to you after that.

Hostel

```
import java.text.NumberFormat;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.Border;
import javafx.scene.layout.BorderStroke;
import javafx.scene.layout.BorderStrokeStyle;
import javafx.scene.layout.BorderWidths;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javafx.scene.control.TextInputDialog;

/**GUI for the Hostel application
  *@author Charatan and Kans
  *@version 7th April 2018
  */
public class Hostel extends Application
{
    // the attributes

    private int noOfRooms;
    private TenantList list;
    // WIDTH and HEIGHT of GUI stored as constants
    private final int WIDTH = 800;
    private final int HEIGHT = 500;
    // visual components
    private Label headingLabel = new Label("Hostel Application");
    private Label roomLabel1 = new Label("Room");
    private TextField roomField1 =  new TextField();
    private Label nameLabel = new Label("Name");
    private TextField nameField =  new TextField();
    private Button addButton = new Button("Add Tenant");
```

```
private Button displayButton =  new  Button("Display Tenants");
private Button removeButton  = new Button("Remove Tenant");
private Button saveAndQuitButton  = new Button("Save and Quit");
private TextArea displayArea1  = new TextArea();
private Label roomLabel2 = new Label("Room");
private TextField roomField2  = new TextField();
private Label monthLabel = new Label("Month");
private TextField monthField  = new TextField();
private Label amountLabel = new Label("Amount");
private TextField amountField =  new TextField();
private Button paymentButton  = new Button("Make Payment");
private Button listButton  = new Button("List Payments");
private TextArea displayArea2 =  new TextArea();



@Override
/** Initialises the screen
 *  @param stage:   The scene's stage
 */
public void start(Stage stage)
{
   noOfRooms = getNumberOfRooms(); // call private method
   // initialise tenant list
   list  = new TenantList(noOfRooms);
   TenantFileHandler.readRecords(list);

   // create four HBoxes
   HBox roomDetails = new HBox (10);
   HBox tenantButtons = new HBox(10);
   HBox paymentDetails = new HBox(10);
   HBox paymentButtons = new HBox(10);
   // add components to HBoxes
   roomDetails.getChildren().addAll(roomLabel1, roomField1, nameLabel, nameField);
   tenantButtons.getChildren().addAll(   addButton, displayButton, removeButton,
                                         saveAndQuitButton);

   paymentDetails.getChildren().addAll(  roomLabel2, roomField2, monthLabel, monthField,
                                         amountLabel, amountField);
   paymentButtons.getChildren().addAll(paymentButton, listButton);
   // create VBox
   VBox root = new VBox(10);
   // add all components to VBox
   root.getChildren().addAll( headingLabel, roomDetails, tenantButtons, displayArea1,
                              paymentDetails, paymentButtons, displayArea2);
   // create the scene
   Scene scene = new Scene(root, Color.LIGHTBLUE);

   // set font of heading
   Font font = new Font("Calibri", 40);
   headingLabel.setFont(font);

   // set alignment of HBoxes
   roomDetails.setAlignment(Pos.CENTER);
   tenantButtons.setAlignment(Pos.CENTER);
   paymentDetails.setAlignment(Pos.CENTER);
   paymentButtons.setAlignment(Pos.CENTER);
   // set alignment of VBox
   root.setAlignment(Pos.CENTER);

   // set minimum and maximum width of components
   roomField1.setMaxWidth(50);
   roomField2.setMaxWidth(50);

   roomDetails.setMinWidth(WIDTH);
   roomDetails.setMaxWidth(WIDTH);

   tenantButtons.setMinWidth(WIDTH);
   tenantButtons.setMaxWidth(WIDTH);

   paymentDetails.setMinWidth(WIDTH);
   paymentDetails.setMaxWidth(WIDTH);

   paymentButtons.setMinWidth(WIDTH);
   paymentButtons.setMaxWidth(WIDTH);

   root.setMinSize(WIDTH, HEIGHT);
   root.setMaxSize(WIDTH, HEIGHT);

   displayArea1.setMaxSize(WIDTH - 80, HEIGHT/5);
   displayArea2.setMaxSize(WIDTH - 80, HEIGHT/5);

   stage.setWidth(WIDTH);
   stage.setHeight(HEIGHT);
```

```
    // customise the visual components

    // customise the VBox border and background
    BorderStroke style = new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID,
                                          new CornerRadii(0), new BorderWidths(2) );
    root.setBorder(new Border (style));
    root.setBackground(Background.EMPTY);

    // customise buttons
    addButton.setBackground(new Background(new BackgroundFill(Color.LIGHTYELLOW,
                            new CornerRadii(10), Insets.EMPTY)));
    displayButton.setBackground(   new Background(new BackgroundFill(Color.LIGHTYELLOW,
                                   new CornerRadii(10), Insets.EMPTY)));
    removeButton.setBackground(new Background(new BackgroundFill(Color.LIGHTYELLOW,
                               new CornerRadii(10), Insets.EMPTY)));
    saveAndQuitButton.setBackground(   new Background(new BackgroundFill(Color.LIGHTYELLOW,
                                       new CornerRadii(10), Insets.EMPTY)));
    paymentButton.setBackground(   new Background(new BackgroundFill(Color.LIGHTYELLOW,
                                   new CornerRadii(10), Insets.EMPTY)));
    listButton.setBackground(   new Background(new BackgroundFill(Color.LIGHTYELLOW,
                               new CornerRadii(10), Insets.EMPTY)));

    // call private methods for button event handlers
    addButton.setOnAction(e -> addHandler());
    displayButton.setOnAction(e -> displayHandler() );
    removeButton.setOnAction( e -> removeHandler());
    paymentButton.setOnAction( e -> paymentHandler());
    listButton.setOnAction( e -> listHandler());
    saveAndQuitButton.setOnAction( e -> saveAndQuitHandler());

    // configure the stage and make the stage visible
    stage.setScene(scene);
    stage.setTitle("Hostel Applicaton");
    stage.setResizable(false); // see discussion below
    stage.show();

 }

 /**
  * Method to request number of hostel rooms from the user
  * @return number of rooms
  */
 private int getNumberOfRooms()
 {
     TextInputDialog dialog = new TextInputDialog();
     dialog.setHeaderText("How many rooms?");
     dialog.setTitle("Room Information Request");

     String response = dialog.showAndWait().get();
     return Integer.parseInt(response);
 }

// event handler methods

 private void addHandler()
 {
     String roomEntered =  roomField1.getText();
     String nameEntered =  nameField.getText();
     // check for errors
     if(roomEntered.length()== 0 || nameEntered.length()== 0)
     {
         displayArea1.setText ("Room number and name must be entered");
     }
     else if(Integer.parseInt(roomEntered)< 1 || Integer.parseInt(roomEntered)>noOfRooms)
     {
         displayArea1.setText ("There are only "  + noOfRooms  + " rooms");
     }
     else if(list.search(Integer.parseInt(roomEntered)) !=  null)
     {
         displayArea1.setText("Room number " +  Integer.parseInt(roomEntered)  + " is occupied");
     }
     else  // ok to add a Tenant
     {
         Tenant t =  new Tenant(nameEntered,Integer.parseInt(roomEntered));
         list.addTenant(t);
         roomField1.setText("");
         nameField.setText("");
         displayArea1.setText("New tenant in room "     +  roomEntered +  " successfully added");
     }
 }
```

```
public void displayHandler()
{
    int i;
    if(list.isEmpty()) // no rooms to display
    {
        displayArea1.setText("All rooms are empty");
    }
    else // display rooms
    {
        displayArea1.setText("Room" +  "\t" +  "Name" +  "\n");
        for(i = 1; i <=  list.getTotal(); i++ )
        {
            displayArea1.appendText(list.getTenant(i).getRoom()
                                    + "\t\t"
                                    + list.getTenant(i).getName() + "\n");
        }
    }
}

private void removeHandler()
{
    String roomEntered =  roomField1.getText();
    // check for errors
    if(roomEntered.length()== 0)
    {
        displayArea1.setText("Room number must be entered");
    }
    else if(Integer.parseInt(roomEntered) < 1 || Integer.parseInt(roomEntered)>noOfRooms)
    {
        displayArea1.setText("Invalid room number");
    }
    else if(list.search(Integer.parseInt(roomEntered))== null)
    {
        displayArea1.setText("Room number " +  roomEntered +  " is empty");
    }
    else // ok to remove Tenant
    {
        list.removeTenant(Integer.parseInt(roomEntered));
        displayArea1.setText("Tenant removed from room " +  Integer.parseInt(roomEntered));
    }
}

private void paymentHandler()
{
    String roomEntered = roomField2.getText();
    String monthEntered = monthField.getText();
    String amountEntered = amountField.getText();
    // check for errors
    if(roomEntered.length()== 0 || monthEntered.length()== 0 || amountEntered.length()== 0)
    {
        displayArea2.setText("Room number, month and amount must all be entered");
    }
    else if(Integer.parseInt(roomEntered) < 1 || Integer.parseInt(roomEntered)>noOfRooms)
    {
        displayArea2.setText("Invalid room number");
    }
    else if(list.search(Integer.parseInt(roomEntered)) == null)
    {
        displayArea2.setText("Room number " +  roomEntered +  " is empty");
    }
    else // ok to process payment
    {
        Payment p =  new Payment(monthEntered,Double.parseDouble(amountEntered));
        list.search(Integer.parseInt(roomEntered)).makePayment(p);
        displayArea2.setText("Payment recorded");
    }
}

private void listHandler()
{
    int i;
    String roomEntered =  roomField2.getText();
    // check for errors
    if(roomEntered.length()== 0)
    {
        displayArea2.setText("Room number must be entered");
    }
    else if(Integer.parseInt(roomEntered) < 1 || Integer.parseInt(roomEntered) > noOfRooms)
    {
        displayArea2.setText("Invalid room number");
    }
    else if(list.search(Integer.parseInt(roomEntered)) == null)
    {
        displayArea2.setText("Room number " + Integer.parseInt(roomEntered) + " is empty");
    }
```

```
        else // ok to list payments
        {
            Tenant t =  list.search(Integer.parseInt(roomEntered));
            PaymentList p  = t.getPayments();
            if(t.getPayments().getTotal() == 0)
            {
                displayArea2.setText("No payments made for this tenant");
            }
            else
            {
                /* The NumberFormat class is similar to the DecimalFormat class that we used
                   previously.
                   The getCurrencyInstance method of this class reads the system values to find out
               which country we are in, then uses the correct currency symbol */
                NumberFormat nf =  NumberFormat.getCurrencyInstance();
                String s;
                displayArea2.setText("Month" +  "\t\t" +  "Amount" +  "\n");
                for(i =  1; i <=  p.getTotal(); i++  )
                {
                    s =  nf.format(p.getPayment(i).getAmount());
                    displayArea2.appendText("" + p.getPayment(i).getMonth() +  "\t\t\t" + s + "\n");
                }
                displayArea2.appendText("\n" + "Total paid so far :   " +
                                                nf.format(p.calculateTotalPaid()));
                monthField.setText("");
                amountField.setText("");
            }
        }
    }

    private void saveAndQuitHandler()
    {
       TenantFileHandler.saveRecords(noOfRooms,list);
       Platform.exit();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Before we complete our examination of the Hostel application we just draw your attention to a few new features.

Firstly, we ask the user for the number of rooms in the hostel by calling the helper method getNumberOfRooms:

```
    private int getNumberOfRooms()
    {
        TextInputDialog dialog = new TextInputDialog();
        dialog.setHeaderText("How many rooms?");
        dialog.setTitle("Room Information Request");

        String response = dialog.showAndWait().get();
        return Integer.parseInt(response);
    }
```

This makes use of a class that we will not actually come across until Chap. 17—the TextInputDialog class. This class provides a really useful way to get information from the user during the running of a JavaFX application. However, as it involves a few advanced concepts we won't deal with it in detail until semester two. For now, if you want to use it, you can just copy what we have done here. The result is shown in Fig. 12.5.

Next, as mentioned above, one new feature we made use of is the NumberFormat class. This class is similar to DecimalFormat that you met in Chap. 10, except that it is designed specifically to convert decimal numbers and convert them into local currency formats. The getCurrencyInstance picks up

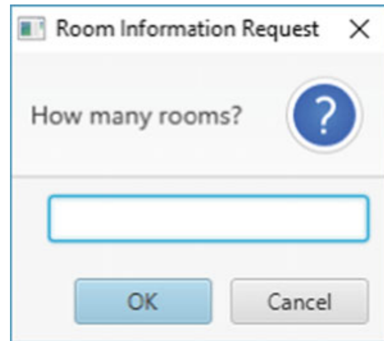


Fig. 12.5 Getting the number of rooms by using a text input dialog

the correct location by interrogating the system and it then returns an appropriate format object:

```
// generate a NumberFormat object
NumberFormat nf =  NumberFormat.getCurrencyInstance();
```

This object's `format` method can then be used to take decimal numbers and format them as local currency values. So the following expression:

```
nf.format(p.calculateTotalPaid())
```

would take a decimal number and, as we are in the United Kingdom, would format the number to two decimal places with a pound sterling symbol (£).

Finally, notice how we configure the stage before making it visible:

```
stage.setScene(scene);
stage.setTitle("Hostel Applicaton");
stage.setResizable(false); // stop the user resizing the stage window
stage.show();
```

You can see we have used a new `stage` method here called `setResizable`. This method takes a **`boolean`** parameter and giving it a value of **`false`** ensures the user cannot resize the window. We thought that would be a good idea as we have gone to such lengths in the code to size our window and the components within it (using our `WIDTH` and `HEIGHT` constants)! One implication of a non-resizable window is that the maximise icon of the window will be greyed out. The final running JavaFX GUI can be seen now in Fig. 12.6.

Before concluding this case study we shall consider how to test the application to ensure that it conforms to the original specification.

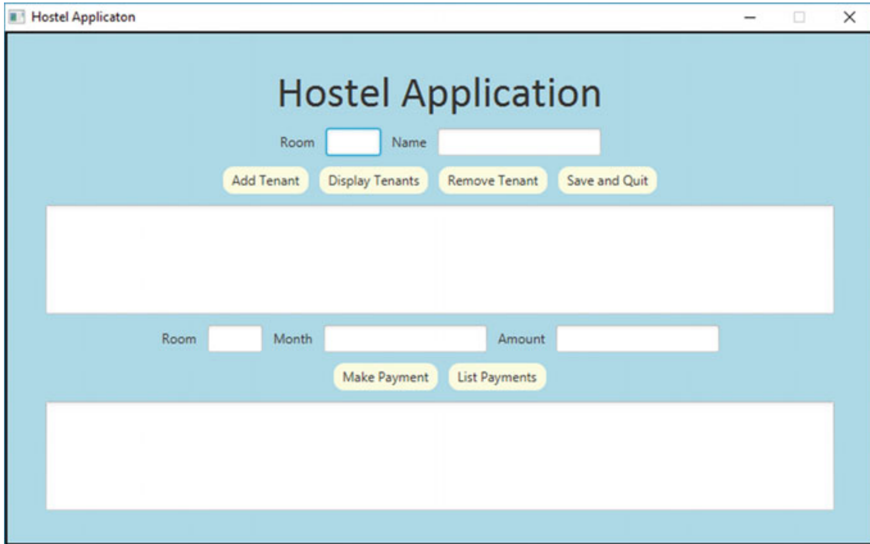


Fig. 12.6 The *Hostel* GUI running in a non-resizable window

12.7 Testing the System

If you look back at the `Hostel` class you can see that much of the event-handling code is related to the validation of data entered from the graphical interface. Much of the testing for such a system will, therefore, be geared around ensuring such validation is effective.

Amongst the types of validation we need to test is the display of suitable error messages when input text fields are left blank, or when inappropriate data has been entered into these text fields. Of course, as well as input validation, we also need to test the basic functionality of the system. Figure 12.7 is one possible test log that may be developed for the purpose of testing the `Hostel` class.

We include a few sample screen shots produced from running the Student Hostel Application against this test log in Figs. 12.8, 12.9 and 12.10. We will leave the complete task of running *Student Hostel Application* against the test log as a programming exercise at the end of this chapter.

TEST LOG			
Purpose: To test the Hostel class			
Run Number:	Date:		
Action	Expected Output	Pass/ Fail	Reason for failure
Request for how many rooms			
Enter 5	Fire up the JavaFX GUI		
Display tenants	"Empty list" message		
Add tenant: Patel, Room Number blank	"Blank field" message		
Add tenant: blank, Room Number 1	"Blank field" message		
Add tenant: Patel, Room Number 1	Confirmation message		
Add tenant: Jones, Room Number 6	Error message: There are only 5 rooms		
Add tenant: Jones, Room Number 1	Error Message: Room 1 is occupied		
Add tenant: Jones, Room Number 2	Confirmation Message		
Display tenants	ROOM NAME		
	1 Patel		
	2 Jones		
List payments, Room Number 1	"Empty list" message		
Make payment: Room blank, Month			
January, Amount 100	"Blank field" message		
Make Payment: Room 1, Month			
blank, Amount 100	"Blank field" message		
Make payment: Room 1, Month			
January, Amount blank	"Blank field" message		
Make payment: Room 1, Month			
January, Amount 100	Confirmation message		
Make payment: Room 1, Month			
February, Amount 200	Confirmation message		
List payments: Room Number blank	"Blank field" message		
List payments, Room Number 1	MONTH AMOUNT		
	January £100		
	February £200		
	Total paid so far £300		
List payments: Room Number 2	"Empty list" message		
List payments: Room Number 5	"Room Empty" message		
Remove tenant: Room Number blank	"Blank field" message		
Remove tenant: Room Number 1	Confirmation Message		
Display tenants	2 Jones		
List payments: Room Number 1	"Room Empty" message		

Fig. 12.7 A test log to ensure the reliability of the *Hostel* class

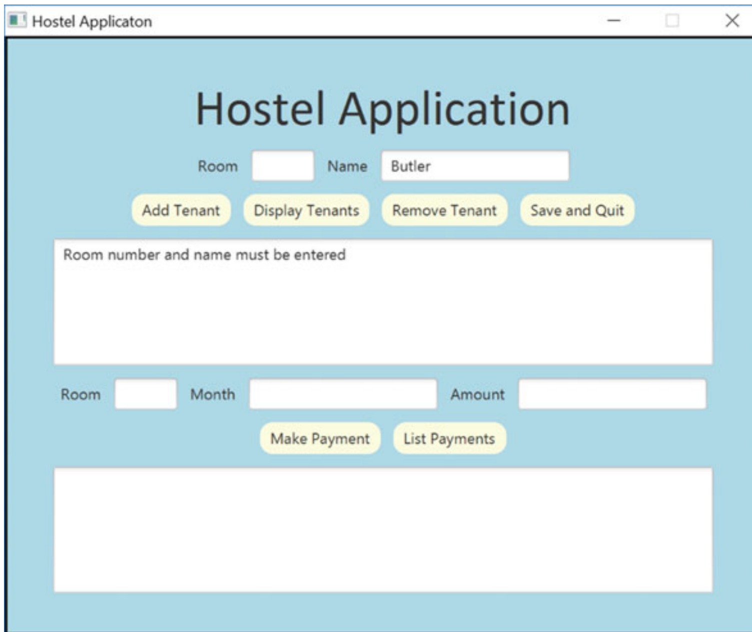


Fig. 12.8 Error messages are produced in *displayArea1*. In this case an attempt is made to add a tenant without filling in the *roomField*

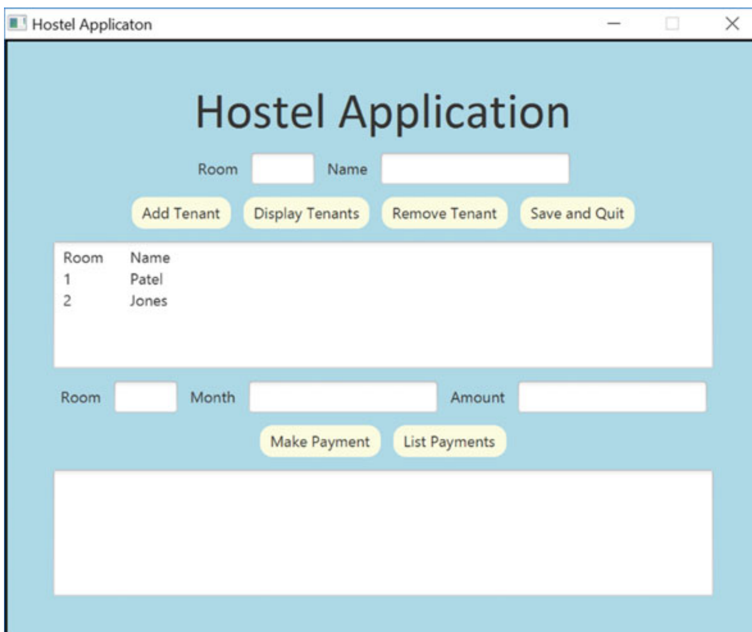


Fig. 12.9 The *displayArea1* is also used to display a list of tenants entered

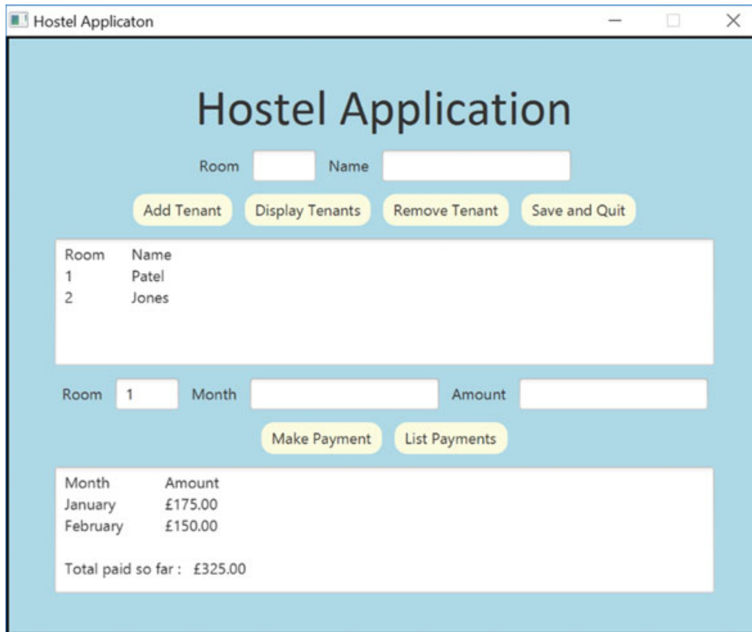


Fig. 12.10 Details of the payments for room 1 are displayed in *displayArea2* when the *List Payments button* is pressed

12.8 What Next?

Congratulations—you have now completed your first semester of programming; we hope you have enjoyed it. Many of you will be going on to at least one more semester of software development and programming—so what lies ahead?

Well, you have probably realized that there are still a few gaps in your knowledge and that some of the stuff that you have learnt can be developed further to give you the power to write multi-functional programs. Think, for example, about the case study we developed in this chapter; you will need to learn how to write the code that stores the information permanently on a disk; also, the JavaFX user interface could be made to look a bit more attractive; and it would be helpful if we had a collection class other than an `ArrayList` that allows us to access an object via a simple key (such as a room number) rather than having to search every item in the collection.

And there is lots more; the standard Java packages provide classes for many different purposes; there is more to learn about interfaces; about dealing with errors and exceptions; about network programming; about accessing information stored in a database and about how to write programs that can perform a number of tasks at

the same time. As well finding out more about the standard Java packages, there is more to learn about how to deploy your own programs as packages.

Does all this sound exciting? We think so—and we hope that you enjoy your next semester as much as we have enjoyed helping you through this one.

12.9 Self-test Questions

1. Referring to the examples in this chapter and the previous chapter, explain the difference between *unit testing* and *integration testing*.
2. Use pseudocode to design the event handling routine for a `search` button. Clicking on the button should display the name of the tenant in the room entered in the `roomField` text box. The name is to be displayed in `displayArea1`. If no tenant is present in the given room, an error message should be displayed in `displayArea1`.
3. Modify the screen design in Fig. 12.1 to include the `search` button discussed in the question above.
4. How else might you improve the application developed in this case study?

12.10 Programming Exercises

You will need to copy the entire suite of classes that make up the student hostel system from the accompanying website.

1. Run the `Hostel` application against the test log given in Fig. 12.7.
2. Modify the `Hostel` class by adding the `search` button you considered in self-test questions 2 and 3 above.
3. Make any additions to the `Hostel` class that you considered in self-test question 4 above. For example you might want to include additional validation to ensure that negative money values are never accepted for payments.

Part II
Semester Two

Check for updates

13 Interfaces and Lambda Expressions

Outcomes:

By the end of this chapter you should be able to:

- *explain what is meant by the term **interface** in Java*;
- *create and use their own interfaces*;
- *implement **inner** classes and **anonymous** classes*;
- *explain the purpose and the importance of **lambda expressions** in Java*;
- *describe the syntax of lambda expressions, and utilize these expressions in a variety of contexts*;
- *create and use **generic** classes and interfaces*;
- *create generic classes and interfaces containing **upper bounded parameters***;
- *utilize **wildcards** in conjunction with generic types*;
- *describe how a programming language can support **polymorphic types***;
- *summarize the ways in which polymorphism can be achieved in Java.*

13.1 Introduction

Welcome back to the second semester of our programming course. We spent the first semester laying the foundations you would need to develop programs in Java. During that time you came a long way. You learnt about the idea of variables, control structures, methods and arrays, and then went on to develop your own classes and extend these classes using inheritance. Finally you developed applications consisting of many classes working closely together and interacting with users via attractive graphical interfaces. Along the way you also learnt about the UML notation and testing strategies. At the beginning of that semester you probably didn't expect to come as far as you have. Well, the second semester might look equally challenging but, with some help from us along the way, you can look forward to new and more advanced challenges.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_13

In the first semester, until we developed our case study, the applications we created were fairly simple, consisting for the most part of only one or two classes. In reality, applications that are developed for commercial and industrial use comprise a large number of classes, and are developed not by one person, but by a team. Members of the team will develop different modules which can later be integrated to form a single application. When groups of programmers work together in this way, they often have to agree upon how they will develop their own individual classes in order for them to be successfully integrated later. In this chapter we will see how these agreements can be formalized in Java programs by making use of a special kind of class called an **interface**. We will then move on to the subject of lambda expressions, which we introduced in Chap. 10. The introduction of lambda expressions in the release of Java 8 was a very significant addition to the language —and in this chapter you will see just how useful they can be.

13.2 An Example

It is a very common occurrence that the attributes of a class should be allowed to take only a particular set of values. Think, for example, of the `BankAccount` class that we developed in the last semester. It is likely that the account number should be restricted to numbers that contain, say, precisely eight digits. Similarly, a `Customer` class might require that the customer number comprises a letter followed by four digits. In some cases there are constraints that exist not because we choose to impose them, but because they occur "naturally"—in the `Oblong` class, for example, it would make no sense if an object of this class were to have a length or height of zero or less.

In such cases, every effort must be made when developing the class to prevent invalid values being assigned to the attributes. Constructors and other methods should be designed so that they flag up errors if such an attempt is made—and one of the advantages of object-oriented programming languages is precisely that they allow us to restrict access to the attributes in this way.

The above remarks notwithstanding, it is the case that in industrial sized projects, classes will be very complex and will contain a great many methods. It is therefore possible that a developer will overlook a constraint at some point, and allow objects to be created that break the rules. It might therefore be useful if, for testing purposes, every object could contain a method, which we could call `check`, that could be used to check the integrity of the object.

In a particular project, people could be writing test routines independently of the people developing the modules, and these routines will be calling the `check` method. We need, therefore, to be able to *guarantee* that every object contains such a `check` method.

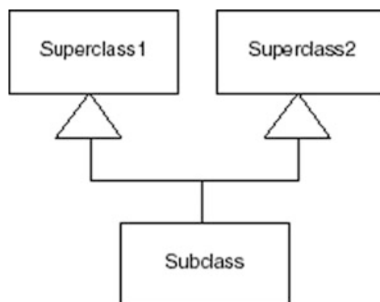


Fig. 13.1 Multiple inheritance—not allowed in Java

You learnt in Chap. 9 that the way to guarantee that a class has a particular method is for that class to inherit from a class containing an **abstract** method—when a class contains an **abstract** method, then any subclass of that class is *forced* to override that method—if it does not do so, a compiler error occurs.

In our example, we need to ensure that our classes all have a check method that tests an object's integrity, so one way to do this would be to write a class as follows:

Checkable

```
public abstract class Checkable
{
        public abstract boolean check();
}
```

Now all our classes could extend Checkable and would compile successfully only if they each had their own check method.

While this would work, it does present us with a bit of a problem. What would happen, for example, if our Oblong class were going to be part of a graphical application and needed to extend another class such as Application? This would be problematic, because, in Java, a class is not allowed to inherit from more than one superclass. Inheriting from more than one class is known as **multiple inheritance** and is illustrated in Fig. 13.1.

One reason that multiple inheritance is disallowed is that it can easily lead to ambiguity, and hence to programming errors. Imagine for example that the two superclasses shown in Fig. 13.1 both contained a method of the same name—which version of the method would be inherited by the subclass?

Luckily, there is a way around this, because Java allows a kind of *lightweight inheritance*, made possible by a construct known as an **interface**.

13.3 Interfaces

An interface is a special class that contains **abstract** methods. When we want a class to inherit the methods of an interface we use the word **implements**, rather than **extends**. Just as with inheritance, once a class implements an interface it has the same type as that interface, as well as being of the type of its own class. So, for

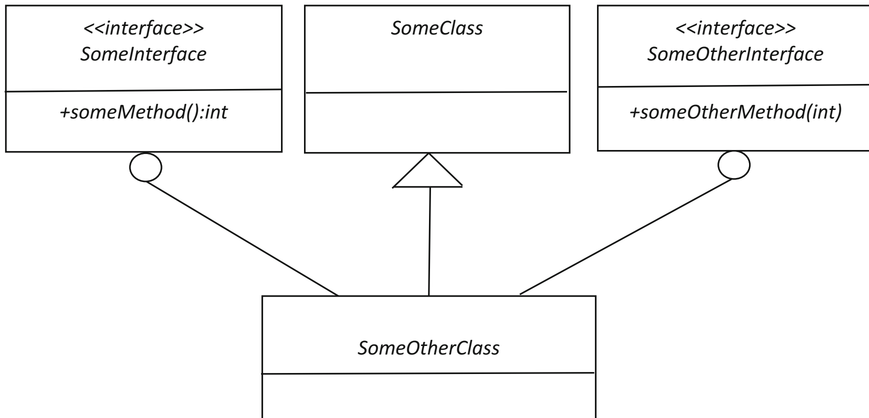


Fig. 13.2 A class can inherit from only one superclass, but can implement many interfaces

example, if a class implements `EventHandler`, then any object of that class is a *kind of* `EventHandler`—in other words it is of type `EventHandler`, as well as being of the type of its own class.

The Java libraries, particularly those associated with graphical applications, contain a great many interfaces. And it is perfectly possible for us to write our own interfaces as we will do in a moment when we turn our `Checkable` class into an interface.

Figure 13.2 shows the UML notation for the implementation of interfaces—you can see that interfaces are marked with the <<interface>> tag, and that a circle is used to indicate a class implementing an interface.

As can be seen in Fig. 13.2, while it is possible to *inherit* from only one class, it is perfectly possible to implement any number of interfaces. In this case `SomeOtherClass` **`extends`** `SomeClass` and **`implements`** `SomeInterface` and `SomeOtherInterface`—because the methods are **`abstract`** they are not actually coded in the interface and this means that the problems with multiple inheritance that we described earlier do not arise.

Prior to the release of Java 8 an interface could contain *only* **`abstract`** methods—but now an interface can also contain **`static`** methods and **`default`** methods. Regular classes have always been permitted to have **`static`** methods (as you have seen before in Chap. 8)—now they can be included in interfaces too. A **`default`** method is a new concept specifically to be used in interfaces, if required. A **`default`** method is a regular method with a complete implementation and so is automatically inherited by all classes that implement the interface (though the implementing class may override this implementation if it chooses). Adding **`default`** methods into interfaces means that additions can be made to an interface without every class that implements the previous version of that interface having to change (as would be the case if we added new **`abstract`** methods to an interface).

We will meet interfaces with **static** and **default** methods in later chapters, but in this chapter we will focus on interfaces with **abstract** methods only.

As with inheritance, a class is *obliged* to override all the abstract methods of the interfaces that it implements. By implementing an interface we are *guaranteeing* that the subclass will have certain methods.

So, in cases where we need a class such as `Checkable` in which all the methods are **abstract** (apart from any **static** methods) we don't create a class—instead we create an interface.

Let's turn our `Checkable` class into an interface. The code looks like this:

The *Checkable* interface

```
public interface Checkable
{
      public boolean check();
}
```

Notice the word **interface** instead of **class**—and notice also that we don't have to declare our method as **abstract**, because by definition all non-**static** methods of an interface are **abstract**.

Let's make the `Oblong` class from Chap. 8 checkable by defining a sub-class that implements the `Checkable` interface. We will need to override the constructor, and of course to code the `check` method. The class will now look like this:

CheckableOblong

```
public class CheckableOblong extends Oblong implements Checkable
{
    //override the constructor
    public CheckableOblong(double lengthIn, double heightIn)
    {
        super(lengthIn, heightIn);
    }

    @Override
    public boolean check() // the check method of Checkable must be overridden
    {
        // the length and height must both be greater than zero
        return getLength() > 0 && getHeight() > 0;
    }
}
```

You can see that the class now implements our `Checkable` interface:

```
public class CheckableOblong extends Oblong implements Checkable
```

The `check` method, which the class is forced to override (note the use of the `@Override` annotation), returns a value of **true** if the attributes are both greater than zero, and **false** otherwise:

```
@Override
public boolean check() // the check method of Checkable must be overridden
{
       return length > 0 && height > 0;
}
```

Other classes can implement the `Checkable` interface in a similar way. Do you remember the `BankAccount` class that we developed in Chapt. 8? One of the attributes was the account number. In reality, an account number would need to obey certain rules—the most common one in the UK is that the account number should contain digits only and that it should comprise exactly eight digits. Let's create a `CheckableBankAccount` class which checks to see if this rule is upheld.

CheckableBankAccount

```
public class CheckableBankAccount extends BankAccount implements Checkable
{
    // override the constructor
    public CheckableBankAccount(String numberIn, String nameIn)
    {
        super(numberIn, nameIn);
    }

    @Override
    public boolean check()
    {
        // check that the account number is exactly 8 characters long
        if(getAccountNumber().length() != 8)
        {
            return false;
        }

        // check that the account number contains only digits
        for(int i=0; i <= 7; i++)
        {
            if(!Character.isDigit(getAccountNumber().charAt(i)))
            {
                return false;
            }
        }
        return true;
    }
}
```

You can see here how, in the `check` method, we check firstly that the string contains exactly eight characters, and then check if every character is a digit by making use of the `isDigit` method of the `Character` class.

In the `Checker` class below we create five objects—two `Checkableblong` objects and three `CheckableBankAccount` objects. In each case the first object is valid, but the others break the rules that we have set for these two classes.

You should notice that in each case the object—whether it is an oblong or a bank account—is of type `Checkable` (as well as being a type of the particular class). This is because both `CheckableOblong` and `CheckableBankAccount` implement the `Checkable` interface.

Checker

```
public class Checker
{
  public static void main(String[] args)
  {
    // create two oblongs
    CheckableOblong oblong1 = new CheckableOblong(10, 8); // valid
    CheckableOblong oblong2 = new CheckableOblong(0, 8); // invalid: first argument is zero

    // create three bank accounts

    CheckableBankAccount account1 = new CheckableBankAccount("12345678", "Smith"); // valid
    CheckableBankAccount account2 = new CheckableBankAccount("S1234567", "Patel"); // invalid: account number
                                                                                    // must contain digits only
    CheckableBankAccount account3 = new CheckableBankAccount("1234567", "Adewale"); // invalid: account number
                                                                          // must be 8 characters long


    // send objects to the checkValidity method
    System.out.println("oblong1 is " + checkValidity(oblong1));
    System.out.println("oblong2 is " + checkValidity(oblong2));

    System.out.println("account1 is " + checkValidity(account1));
    System.out.println("account2 is " + checkValidity(account2));
    System.out.println("account3 is " + checkValidity(account3));
  }

  private static String checkValidity(Checkable objectIn) // receives any Checkable object
  {
    if(objectIn.check())  // call the check method
    {
       return "valid";
    }
    else
    {
       return "invalid";
    }
  }
}
```

As you can see, we send the five objects in turn into a method called `checkValidity` which calls the object's `check` method and returns a `String`—either "valid" or "invalid"; we append this to an initial string and the whole message is displayed.

The `checkValidity` method accepts a parameter of type `Checkable`—and of course both the `CheckableOblong` objects and the three `CheckableBankAccount` objects are of type `Checkable` because they both implement the `Checkable` interface.

As expected, the output from the program is as follows:

`oblong1 is valid`
`oblong2 is invalid`
`account1 is valid`
`account2 is invalid`
`account3 is invalid`

Implementing an interface is rather like making a contract with the user of a class—it *guarantees* that the class will have a particular method or methods. In the above case, a developer will know that any object that implements `Checkable` will have a `check` method. This enables the developer to write methods such as `checkValidity` that expect to receive an object of type `Checkable`, in the certain knowledge that the object—whether it is a `CheckableOblong`, `CheckableBankAccount`, or any other class that implements this interface—will have a method called `check`.

13.4 Inner Classes

It's quite easy to imagine a situation in which a large application is being developed that would make extensive use of a class such as `CheckableBankAcount`. However there are other instances when such a class might be needed only once, and it would seem like a lot of bother to create a new class just for that purpose. In such a case it is possible to write the class "on the fly", by making use of an **inner class**—a class written inside another class.

The program below does this with the `CheckableOblong` class.

InnerClassDemo

```
public class InnerClassDemo
{
    public static void main(String[] args)
    {
       // inner class
       class CheckableOblong extends Oblong implements Checkable
       {
            public CheckableOblong(double lengthIn, double heightIn)
            {
                 super(lengthIn, heightIn);
            }

            @Override
            public boolean check()
            {
                        return getLength() > 0 && getHeight() > 0;
            }
       }

       Checkable oblong1 = new CheckableOblong(5, 0);  //invalid
       Checkable oblong2 = new CheckableOblong(5, 6);  // valid

       System.out.println("oblong1 is " + checkValidity(oblong1));
       System.out.println("oblong2 is " + checkValidity(oblong2));
    }

    private static String checkValidity(Checkable objectIn)
    {
        if(objectIn.check())
        {
            return "valid";
        }
        else
        {
            return "invalid";
        }
    }
}
```

You can see how we have written the `CheckableOblong` class within our main class.

It is also worth noting that, although we have not done this here, you can refer to attributes of the outer class in the inner class: you can also refer to local variables, but they must be **`final`**.

13.5 Anonymous Classes

Do you remember that when we introduced the `Oblong` class in Chap. 7 we mentioned that a true oblong differs from a rectangle in one respect? An oblong, unlike a rectangle, cannot have equal sides (in other words, a square is not a kind of

oblong). So another test that we might want to perform on an oblong is to check that its sides are not equal. It would seem rather unnecessary to have to write a whole new class for this, and in fact there is a more flexible way of doing it. We can create an **anonymous** class. Have a look at the program below, then we will explain what's going on here.

AnonymousClassDemoVersion1

```
public class AnonymousClassDemoVersion1
{
    public static void main(String[] args)
    {
       // create a test oblong
       Oblong testOblong = new Oblong (8,8);

       /* declare an object of an anonymous class that checks that an oblong's length
          and height are greater than zero */

       Checkable checkableObject1 = new Checkable()
          {
             @Override
             public boolean check()
             {
                   return testOblong.getLength() > 0 && testOblong.getHeight() > 0;
             }
          };

      /* declare an object of an anonymous class that checks that an oblong's length
         and height are not equal */

      Checkable checkableObject2 = new Checkable()
         {
             @Override
             public boolean check()
             {
                   return testOblong.getLength() != testOblong.getHeight();
             }

         };

       // this checks that the sides are greater than zero
       System.out.println("checkableObject1 is " + checkValidity(checkableObject1));

       // this checks that the length and height are not equal
       System.out.println("checkableObject2 is " + checkValidity(checkableObject2));
    }

    private static String checkValidity(Checkable objectIn)
    {
        if(objectIn.check())
        {
            return "valid";
        }
        else
        {
            return "invalid";
        }
    }
}
```

We have declared an `Oblong` of length 8 and height 8—it therefore passes our original test (that both the length and height are greater than zero) but fails the second test (that the length and height are unequal).

Now we have the following:

```
Checkable checkableObject1 = new Checkable()
     {
             @Override
             public boolean check()
             {
                   return testOblong.getLength() > 0 && testOblong.getHeight() > 0;
             }
     };
```

Here we have declared an object, `checkableObject1`, which is of type `Checkable`, but which doesn't belong to any named class. It is an object of an *anonymous* class, as defined between the braces. Effectively we have defined a class "as we go"—all we need for this class is its `check` method (because it is of type `Checkable`). You can see that this method checks, as before, that both length and height are greater than zero. You will notice that we have been able to refer to `testOblong`, which was declared in the outer class.

After this we have gone on to declare another object, `checkableObject2`, also of type `checkable`:

```
Checkable checkableObject2 = new Checkable()
     {
            @Override
            public boolean check()
            {
                  return testOblong.getLength() != testOblong.getHeight();
            }

     };
```

Here, however the code for the `check` method is different. We are now checking for the second of our criteria—namely that the length and height of the oblong are not the same.

Now we can send both these objects to the `checkValidity` method:

```
// this tests that the sides are greater than zero
System.out.println("checkableObject1 is " + checkValidity(checkableObject1));

// this tests that the length and height are not equal
System.out.println("checkableObject2 is " + checkValidity(checkableObject2));
```

As expected, we get the following output:

```
checkableObject1 is valid
checkableObject2 is invalid
```

Before moving on, we should point out that it is not actually necessary to declare a named object as we have done in the previous program. We can declare our object as part of the call to the method we are sending it to. The program that follows demonstrates how this is done.

AnonymousClassDemoVersion2

```
public class AnonymousClassDemoVersion2
{
    public static void main(String[] args)
    {

       // create a test oblong
       Oblong oblong = new Oblong (8,8);

       // this checks that the sides are greater than zero
       System.out.println("oblong is " + checkValidity(new Checkable()
                                    {
                                        @Override
                                        public boolean check()
                                        {
                                            return oblong.getLength() > 0 && oblong.getHeight() > 0;
                                        }
                                    }
                                                        ));

       // this checks that the length and height are not equal
       System.out.println("oblong is " + checkValidity(new Checkable()
                                    {
                                        @Override
                                        public boolean check()
                                        {
                                            return oblong.getLength() != oblong.getHeight();
                                        }
                                    }
                                                       ));


    }

    private static String checkValidity(Checkable objectIn)
    {
        if(objectIn.check())
        {
            return "valid";
        }
        else
        {
            return "invalid";
        }
    }
}
```

As you can see, the first call to `checkValidity` looks like this:

```
System.out.println("oblong is " + checkValidity(new Checkable()
                                     {
                                         @Override
                                         public boolean check()
                                         {
                                             return  oblong.getLength()  >  0  &&  oblong.getHeight()  >
0;
                                         }
                                     }
```

We send in a new `Checkable` object, at the same time defining the anonymous class to which the object belongs. The next call is the same apart from the code for the `check` method.

If you think about it, all we are really interested in here is the `check` method—so effectively what we are doing is sending the code for this method to `checkValidity` by defining an anonymous class.

You will probably agree that although this is a very useful thing to be able to do (because we can change our method each time), it does seem a bit cumbersome, and the code doesn't look very elegant. If only there was a way to simply send a block of code to a method, instead of having to send a whole object!

Well, with the advent of Java 8, that's is exactly what we can do. And you have probably already worked out that the mechanism that we use for this is called a lambda expression. You have already had some experience with such expressions when you learnt about creating graphics programs with JavaFX in Chap. 10. But now it is time to explore lambda expressions in greater depth.

13.6 Lambda Expressions

In the program below, `LambdaDemo`, the anonymous classes of the previous example have been replaced by lambda expressions. As you have already seen, we can use a lambda expression to send the code for a particular method to another method. Here we send the code for the `check` method of `checkable` to the `checkValidity` method. You will notice that the format of the lambda expression used here is a little different to the format we saw in Chap. 10. Have a look at it, then we will explain it.

LambdaDemo

```
public class LambdaDemo
{
    public static void main(String[] args)
    {
       // create a test oblong
       Oblong testOblong = new Oblong (8,8);

       // this checks that the sides are greater than zero
       System.out.println("oblong is " + checkValidity(() ->
                               {
                                  return testOblong.getLength() > 0 && testOblong.getHeight() > 0;
                               }
                          ));

       // this checks that the length and height are not equal
       System.out.println("oblong is " + checkValidity(() ->
                                {
                                  return testOblong.getLength() != testOblong.getHeight();
                                }
                             ));
    }

    private static String checkValidity(Checkable objectIn)
    {
        if(objectIn.check())
        {
            return "valid";
        }
        else
        {
            return "invalid";
        }
    }
}
```

The first lambda expression looks like this:

```
() ->   {
             return testOblong.getLength() > 0 && testOblong.getHeight() > 0;
         }
```

The difference between this and the expressions you saw in Chap. 10 is that instead of a variable name on the left of the arrow, we now have a pair of empty brackets. The reason we have done this is because the `check` method does not require any arguments. There are in fact a number of different formats to lambda expressions and we will tell you about them shortly. Before we do that, we will say a little more about how this all works, and the importance of this feature of Java which was introduced with Java 8.

Lambda expressions can be used to send a block of code to any method that expects to receive a **functional interface** as a parameter. A functional interface is an interface that contains *only one* abstract method. `Checkable` is a functional interface for example, as it contains a single abstract method—`check`. The block of code supplied in the lambda expression—which is the code for the abstract method—might be used just once, as we saw in the above program, or many times as we saw in the examples in Chap. 10 where a button can be pressed whenever the user chooses.

Languages that are based on blocks of code being sent to methods are called *functional languages*—examples being Lisp, Clojure and Scala. And while it is true to say that the introduction of lambda expressions hasn't put Java in the same league as these, it has certainly given Java some of the same capabilities.

The Java APIs provide a great many interfaces, many of which are concerned with graphics programming. Interfaces also play a very important role in the collection classes that you will learn more about in Chap. 15 and in multi-threaded programming which you will study in Chap. 20.

13.6.1 The Syntax of Lambda Expressions

You have already seen how lambda expressions are formed, with the instructions on the right side of the arrow, and the parameters on the left.

As we explained in Chap. 10, the code on the right can be a single statement (without the semi-colon), or a number of statements, enclosed in braces with a semi-colon at the end of each statement.

An example of the first might look like this:

```
() -> System.out.println("Hello")
```

Whereas an example of the second could look like this:

```
() -> {
          System.out.println("Hello");
          System.out.println("Goodbye");
      }
```

So, what about the left-hand side of the arrow? You have seen, as in the above two examples, that if the code for the particular method of the interface does not require parameters, then we simply place empty brackets in front of the arrow.

In Chap. 10, you saw that when a single parameter is required, we give that parameter a name and place it in front of the arrow. So, for example we might have:

```
str -> System.out.println("Hello " + str)
```

We don't have to specify a type for `str`, because the compiler will infer this from the header of the **`abstract`** method. This is another example of *type inference*, which you first encountered in Chap. 7.

If there is more than one parameter, then we would list them in brackets. For example:

```
(x, y) -> {
               int z;
               z = x + y;
               System.out.println("Sum = " + z);
           }
```

On occasion, you might find that for some reason the compiler is unable to infer the types of the variables, and you get a compiler error. To fix this you can simply place the type name in front of the variable name:

```
(int x, int y) -> {
                        int z;
                        z = x + y;

                        System.out.println("Sum = " + z);
                   }
```

There is one thing to watch out for when writing lambda expressions. If your code consists of a single **`return`** statement you will get a compiler error if you use the single line format without the semi-colon. For example, the following would give you an error:

```
x -> return 2 * x
```

There are two ways you can avoid this error. Firstly you should note that if there is a single expression on the right of the arrow, java will evaluate this and return the value. So the above expression could be written as:

```
x -> 2 * x
```

Alternatively you could enclose the statement in braces and write:

```
x -> {
        return 2 * x;
     }
```

13.6.2 Variable Scope

Lambda expressions can access the attributes of the enclosing class. They also have access to any parameters that are passed to a method that encloses the expression and to the local variables of that method. However in the case of parameters and local variables, the lambda expression cannot change the value of these—in other words they must be **final**, or effectively **final**.[1]

13.6.3 Example Programs

In this section we will develop a few simple programs to show the various ways that lambda expressions can be written and utilized.

In each of the programs that follow we will refer to a functional interface called `TestInterface` that will contain one **abstract** method called `test`. We will re-define the header for this method in each of the programs.

For the first of our programs the `test` method will look like this:

```
public void test();
```

The following program uses this version:

LambdaSyntaxDemo1

```
public class LambdaSyntaxDemo1
{
    public static void main(String[] args)
    {
        testMethod(() -> System.out.println("Hello "));
    }

    static void testMethod(TestInterface testObjectIn)
    {
        testObjectIn.test();
    }
}
```

We call a method called `testMethod`, which expects to receive an object of type `TestInterface`. We are able simply to send the code for the test method as a lambda expression, which, since `test` does not require any parameters, has open brackets in front of the arrow. There is only one line of code, so we can manage without any braces.

[1]A variable or parameter is said to be *effectively final* if its value is not changed after its initialization.

The output from this program is simply:

```
Hello
```

The next program shows the change in syntax when more than one line of code is required.

LambdaSyntaxDemo2

```
public class LambdaSyntaxDemo2
{
    public static void main(String[] args)
    {
        testMethod( () -> {
                              System.out.print("Hello ");
                              System.out.println("world");
                           }
                  );
    }

    static void testMethod(TestInterface testObjectIn)
    {
        testObjectIn.test();
    }
}
```

The output from this program is of course:

```
Hello world
```

For our next program we will change the `test` method of `TestInterface` to the following:

```
public String test(String stringIn);
```

The method now receives a parameter of type `String`. It also returns a `String`, and we have used this in the version of `testMethod` in our next program shown below.

LambdaSyntaxDemo3

```
public class LambdaSyntaxDemo3
{
    public static void main(String[] args)
    {
        testMethod( str -> {
                              str = "Hello " + str;
                              return str;
                           }
                  );
    }

    static void testMethod(TestInterface testObjectIn)
    {
        String output = testObjectIn.test("world"); // test now requires a String argument
        System.out.println(output);
    }
}
```

As you can see, because test now requires an input, we have named the argument to this method on the left-hand side of the arrow. Our lambda expression looks like this:

```
str -> {
                str = "Hello " + str;
                return str;
        }
```

`testMethod` now calls `test` with the argument "world". `test` obeys the instructions sent to `testMethod`, and produces the following output:

```
Hello world
```

For our final example we will change the header for the test method to the following:

```
public void test(int firstNumber, int secondNumber);
```

Now `test` accepts two **`int`**s. We have used this version in our next program:

LambdaSyntaxDemo4

```
public class LambdaSyntaxDemo4
{
    public static void main(String[] args)
    {
        testMethod( (x, y) -> System.out.println("The sum is " + (x + y)) );
    }

    static void testMethod(TestInterface testObjectIn)
    {
        testObjectIn.test(10, 5);
    }
}
```

Because `test` now requires two arguments, we place these in brackets in front of the arrow:

```
(x, y) -> System.out.println("The sum is " + (x + y))
```

As we mentioned before, we don't have to specify the types for `x` and `y`, but we could do so as follows, without changing the way the program works:

```
(int x, int y) -> System.out.println("The sum is " + (x + y))
```

testMethod calls test with arguments of 10 and 5, so the output is:

```
The sum is 15
```

Before we leave this section, it is worth making one thing absolutely clear. While lambda expressions enable us to effectively send a block of code, what we are actually doing is sending an object which is a type of functional interface, and the code we send is the code for its abstract method. In LambdaSyntaxDemo4 above, for example, we could have written the instructions in the main method like this:

```
TestInterface t = (x, y) -> System.out.println("The sum is " + (x + y));
testMethod(t);
```

And of course, before we had lambda expressions, we would have to have used an anonymous class:

```
TestInterface t = new TestInterface()
                        {
                                 public void test(int x, int y)
                                 {
                                          System.out.println("The sum is " + (x + y));
                                 }
                         };
testMethod(t);
```

13.6.4 Method References—The Double Colon Operator

It is sometimes the case that a lambda expression does nothing more than reference a method of an existing class. To illustrate this, consider the following interface:

DoubleColonInterface

```
public interface DoubleColonInterface
{
    public void test(String s);
}
```

Now consider the following program that uses this interface:

MethodReference

```
public class MethodReference
{
    public static void main(String[] args)
    {
         testMethod(str -> System.out.println(str));
    }

    static void testMethod(DoubleColonInterface testObjectIn)
    {
         testObjectIn.test("Hello world");
    }
}
```

All that the lambda expression does is to call the `println` method of `System.out`, with whatever parameter is specified when `test` is called. In cases such as this, a notation exists that can simplify the code. This uses a double colon to reference the method, as shown below:

DoubleColonDemo

```
public class DoubleColonDemo
{
    public static void main(String[] args)
    {
        testMethod(System.out::println);
    }

    static void testMethod(DoubleColonInterface testObjectIn)
    {
        testObjectIn.test("Hello world");
    }
}
```

This does exactly the same as the previous lambda expression—it calls the `println` method of `System.out` with argument suppled to the `test` method.

Let's look at one more example. We will change the interface we are using as follows:

DoubleColonInterface

```
public interface DoubleColonInterface
{
    public double test(int i);
}
```

Now look at this program:

DoubleColonDemo2

```
public class DoubleColonDemo2
{
    public static void main(String[] args)
    {
        testMethod(Math::sqrt);
    }

    static void testMethod(DoubleColonInterface testObjectIn)
    {
        System.out.println(testObjectIn.test(25));
    }
}
```

Here we are using the `sqrt` (square root) function of Java's `Math` class. The double colon replaces the following lambda expression:

```
testMethod(i -> Math.sqrt(i));
```

In both cases, of course, the program will output `0.5`.

The double colon notation is particularly useful when we are processing streams, as explained in Chap. 22.

13.7 Generics

Do you remember in Chap. 7 that we introduced the ArrayList class, and briefly explained that this class is an example of a *generic* class? We went on to use this class in our case study in Chaps. 11 and 12.

The topic of generics is a very important one. A **generic** class (or interface) has attributes and methods whose types are not defined within the class, but are left to the user to decide upon when an object of the class is declared. Effectively we are sending a type into a class (or interface) and for this reason we often refer to generic classes and interfaces as **parameterized types**.

This is best illustrated by way of an example. Below we have created a very simple generic class—it has only one attribute, together with a set- and a get-method.

SimpleGenericClass

```
public class SimpleGenericClass<T> // the angle brackets indicate that this is a generic class
{
    private T value;

    public void setValue(T valueIn)
    {
        value = valueIn;
    }

    public T getValue()
    {
        return value;
    }
}
```

The angle brackets after the class name indicate that this is a generic class. The T in these brackets indicates that there will be a single type chosen by the user, and we will refer to this type as T throughout the definition. You can think of it as a place-marker for whatever type is chosen by the user of this class. You will also see in one of the examples that follow that we can indicate more than one type in the brackets—so we could have, for example:

```
public class AnotherGenericClass<T, U, V>
```

In our SimpleGenericClass you can see that the single attribute, value, is declared as being of type T. Also, as we would expect, the set-method has a parameter of type T and the get-method returns an object of type T. You should note that the types have to be objects of a class—primitive types such as **int** or **double** can't be used here, so you would have to use the equivalent wrapper classes such as Integer and Double.

There follows a short program that uses this class.

TestGenericClass

```
public class TestGenericClass
{
    public static void main(String[] args)
    {
        SimpleGenericClass<Double> example1 = new SimpleGenericClass<>();
        SimpleGenericClass<String> example2 = new SimpleGenericClass<>();
        SimpleGenericClass<Oblong> example3 = new SimpleGenericClass<>();

        example1.setValue(10.0);
        example2.setValue("Hello");
        example3.setValue(new Oblong(5, 3));

        System.out.println(example1.getValue());
        System.out.println(example2.getValue());
        System.out.println(example3.getValue().calculateArea());
    }
}
```

You can see that we have declared three objects of type `SimpleGenericClass`, each time choosing a different type for its attribute and methods—in the third case we have used our own `Oblong` class. Notice also that in the first case we have to use the wrapper class `Double`, rather than the primitive type.

When calling the constructor of the class, we have left the angle brackets empty:

```
SimpleGenericClass<Double> example1 = new SimpleGenericClass<>();
```

The empty brackets (sometimes referred to as the diamond) can be used in cases where it is easy for the compiler to work out what type of arguments are required; here, for example, it is apparent from the type declaration. This is another example of type inference.

We have then gone on to use the `setValue` method to give a value to the attribute for each object we created. Note that in the first case the argument of 10.0 (a **`double`**) is automatically type cast to `Double`.

In the last three statements we use `getValue` to return the object. In the first two examples we can display the value without having to call a method of the object (`println` is set up to automatically print the value of a `Double` or a `String`). In the final example we display the area of the `Oblong`, using the `calculateArea` method.

The output from this program is:

```
10.0
Hello
15.0
```

Collection classes and graphics classes make extensive use of generic *interfaces*, rather than classes. We could, for example, define a generic functional interface as follows:

SimpleGenericInterface

```
public interface SimpleGenericInterface<T, U, V>
{
    public T doSomething(U firstValue, V secondValue);
}
```

You can see that an object of type `SimpleGenericInterface` would require three types, which are referred to in the definition as `T`, `U` and `V`. The single **`abstract`** method returns an object of type `T` and receives objects of type `U` and `V`.

Very often, however, we don't need to define our own functional interface, because Java provides us with a number of such interfaces "out of the box". Some of these are listed in Table 13.1. Most often these reside in the `java.util.function` package. Some of these also contain **`static`** methods, which you can look up on the Oracle™ site.

The two programs that follow demonstrate how we can use one of the above interfaces—`Function`. In the first one we do this by declaring an anonymous class, while in the second we do the same thing much more neatly with a lambda expression. In both cases the output is:

```
You entered 10
```

In both programs you can see how we have pattern-matched `Integer`, `String` to the types in the interface definition, `T`, `R`.

Table 13.1 Some common functional interfaces

Functional interface	Abstract method name	Parameter types	Return type
Supplier<T>	get	none	T
Consumer<T>	accept	T	void
BiConsumer<T, U>	accept	T, U	void
Function<T, R>	apply	T	R
BiFunction<T, U, R>	apply	T, U	R
UnaryOperator<T>	apply	T	T
BinaryOperator<T, T>	apply	T, T	T
Predicate<T>	test	T	boolean
BiPredicate<T, U>	test	T, U	boolean

TestGenericInterfaceVersion1

```
import java.util.function.Function;

public class TestGenericInterfaceVersion1
{
    public static void main(String[] args)
    {
        Function<Integer, String> t = new Function <Integer, String>()
                                                {
                                                    @Override
                                                    public String apply(Integer i)
                                                    {
                                                        return "You entered " + i;
                                                    }
                                                 };
        System.out.println(t.apply(10));
    }
}
```

TestGenericInterfaceVersion2

```
import java.util.function.Function;

public class TestGenericInterfaceVersion2
{
    public static void main(String[] args)
    {
        Function<Integer, String> str =  i -> "You entered " + i ;

        System.out.println(str.apply(10));
    }
}
```

Notice that in the program above we do not need to include the word **return** in the lambda expression as the code consists of a single **return** statement.

Before we move on there is one more thing to mention. If one of the types of our generic class or interface refers to the return type of a method, it is possible that we might want that method not to return any value—in other words to be of type **void**. A special class, Void, exists for this purpose. Void is simply a placeholder used to represent the keyword **void**. The method in this case would be defined so that a **null** value is returned. So a generic type such as Task<V> could be instantiated with, for example, Task<Integer>, Task<Double> or Task <Void>. You will see an example of this in Chap. 20.

13.7.1 Bounded Type Parameters

When we develop a generic class we are somewhat limited as to the methods we can provide. For example, if we were hoping that the class were going to hold 3D geometrical shapes, we might want whatever types we were dealing with to have methods such as calculateSurfaceArea, calculateVolume etc. If we tried to reference these in our generic methods we would get a compiler error, because a class which is simply declared as being of type T will not have these methods until we know what T is.

Luckily there is a way around this. We can declare our unknown type with an **upper bound**. This means that we specify that the types have to be a particular type, or a derivative of that type. For example, classes such as Integer, Double

and Float are in fact all subclasses of a superclass called Number. We could specify that the type must be Number, or a sub-type of Number such as Integer, Double, Float etc.

We will develop an example to show you how we do this. Firstly, we are going to define two classes to represent the 3D shapes sphere and cuboid. The JavaFX libraries already contain shapes of this type, and, although it is perfectly possible to have classes in different packages with the same name (more about this in Chap. 19), we will avoid any confusion and call our classes Ball and Brick. Both of these classes are going to implement the following interface:

The *Calculatable* interface

```
public interface Calculatable
{
        public double calculateVolume();
}
```

Our Ball and Brick classes below implement this interface, and use the appropriate formula in each case for calculating the volume.[2]

Ball

```
public class Ball implements Calculatable
{
    private double radius;

    public Ball (double radiusIn)
    {
        radius = radiusIn;
    }

    @Override
    public double calculateVolume()
    {
        // uses the constant PI and the method pow from the java.Math package
        return (4 * Math.PI * Math.pow(radius, 3))/3;
    }
}
```

Brick

```
public class Brick implements Calculatable
{
    private double length;
    private double width;
    private double height;

    public Brick (double lengthIn, int widthIn, int heightIn)
    {
        length = lengthIn;
        width = widthIn;
        height = heightIn;
    }

    @Override
    public double calculateVolume()
    {
        return length * width * height;
    }
}
```

[2]Just in case you have forgotten your high school maths, the formula for calculating the volume of a sphere is $\frac{4}{3}\pi r^3$.

Now we'll develop a class, VolumeComparison, that can be used to compare the volumes of two solids. It will be a generic class that will hold two solid objects that implement Calculatable; our class will compare their volumes using the calculateVolume method. Our intention is that the two items can be any objects that implement Calculatable, and do not have to be specified until the VolumeComparison class is instantiated.

The VolumeComparison class is presented below. You will notice that there is something new in the class header. Study the code and then we will explain what's going on.

VolumeComparison

```
public class VolumeComparison<T extends Calculatable, S extends Calculatable>
{
    T first;
    S second;

    public VolumeComparison(T firstIn, S secondIn)
    {
        first = firstIn;
        second = secondIn;
    }

    public int compareVolume()
    {
        if(first.calculateVolume() < second.calculateVolume())
        {
            return -1;
        }
        else if(first.calculateVolume() > second.calculateVolume())
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
}
```

Take a look at the header:

```
public class VolumeComparison<T extends Calculatable, S extends Calculatable>
```

You can see that each of the type parameters contains the words extends Calculatable. Using extends in this context means that the types can be any subtype of the class or interface that follows—in this case Calculatable. Our types now have an **upper bound**—any types or subtypes of this bound will be accepted.

Because both attributes will hold Calculatable objects, we have been able to use the object's calculateVolume method in the code for the compareVolume method of our class.

The program below, ComparisonTester, tests out our VolumeComparison class. You can see how we have been able to create an object of this class with a Ball and a Brick respectively—and of course we could have chosen to compare any objects that implement our Calculatable interface.

ComparisonTester

```
public class ComparisonTester
{
    public static void main(String[] args)
    {
          Ball ballObject = new Ball(10);
          Brick brickObject = new Brick(10, 10, 10);

          VolumeComparison<Ball, Brick> comparison = new VolumeComparison<>(ballObject, brickObject);

          switch(comparison.compareVolume())
          {
               case -1:  System.out.println("The second object has a larger volume");
                         break;
               case 1:   System.out.println("The first object has a larger volume");
                         break;
               case 0:  System.out.println("The volumes are the same");
          }
    }

}
```

13.7.2 Wildcards

We have developed a program called `WildCardTester` which you see below. It makes use of the `SimpleGenericClass` that we developed earlier in this section, and also utilises the `BankAccount` class and `CheckableBankAccount` class that we developed in Sect. 13.4. The line in bold gives rise to a compiler error.

WildCardTester - Incorrect

```
public class WildcardTester
{
    public static void main(String[] args)
    {

               SimpleGenericClass<BankAccount> object1 = new SimpleGenericClass<>();
               SimpleGenericClass<CheckableBankAccount> object2 = new SimpleGenericClass<>();

               object1.setValue(new BankAccount("12345678", "Smith"));
               object2.setValue(new CheckableBankAccount("87654321", "Jones"));

               helper(object1);
               helper(object2);  // this line causes the compiler error
    }

    static void helper(SimpleGenericClass<BankAccount> objectIn) // this causes an error
    {
        System.out.println(objectIn.getValue().getAccountName());
    }
}
```

Can you see what's wrong here? The helper method expects to receive an object of `SimpleGenericClass<BankAccount>`. So `object1` is fine, but `object2` is of type `SimpleGenericClass<CheckableBankAccount>`. The parameter of the receiving method is very specific—a `SimpleGenericClass<BankAccount>` object is required.

We can fix this with a **wildcard**, which uses the? symbol. This is a mechanism for making the variable less restrictive. The way we do it is shown below:

WildCardTester - Correct

```
public class WildcardTester
{
    public static void main(String[] args)
    {

            SimpleGenericClass<BankAccount> object1 = new SimpleGenericClass<>();
            SimpleGenericClass<CheckableBankAccount> object2 = new SimpleGenericClass<>();

            object1.setValue(new BankAccount("12345678", "Smith"));
            object2.setValue(new CheckableBankAccount("87654321", "Jones"));

            helper(object1);
            helper(object2);
    }

   static void helper(SimpleGenericClass<? extends BankAccount> objectIn) // uses a wildcard
    {
        System.out.println(objectIn.getValue().getAccountName());
    }
}
```

The header for the helper method now looks like this:

```
static void helper(SimpleGenericClass<? extends BankAccount> objectIn)
```

Using `<? extends SomeClass>` means that the generic parameter can hold items of `SomeClass` *or any subtype of* `SomeClass`. So now our program compiles and runs without a problem.

You should note that in this case the keyword `extends` includes both classes and interfaces, as it does with bounded types.

13.8 Other Interfaces Provided with the Java Libraries

Many important interfaces are provided in the Java libraries in addition to those listed in Table 13.1, and many of these are not functional interfaces, as they contain more than one **`abstract`** method. In Chap. 15 you will learn much more about classes such as `ArrayList`, which are known as **collection classes**. As their name suggests, these are the classes designed to hold collections of objects. Collection classes implement generic interfaces such as the `List` interface and the `Map` interface, in order to provide a wealth of classes which allow us to handle different types of collections.

In Chap. 20 you will study multithreaded programs—programs in which more than one task can effectively execute at the same time. In that chapter you will encounter the very important `Runnable` interface.

Of particular interest are the interfaces provided in the JavaFX libraries. In Chap. 10 we introduced you to the idea of *event handling*. In JavaFX, event handling is achieved by means of a generic interface called `EventHandler`. Now, as you saw in Chap. 10, in most cases we don't actually have direct

involvement with this interface, because it is hidden from us by convenience methods such as `setOnAction`. Nonetheless, it is important that we know what is going on behind the scenes, not least because—as you will see in Chap. 16—there are times when we will not be able to use convenience methods.

If you cast your mind back to the `PushMe` class of Chap. 10, you will remember that the code for the `setOnAction` method was as follows:

```
pushMeButton.setOnAction(e -> pushMeLabel.setText("You entered: " + pushMeTextField.getText()));
```

Had we not been able to use a convenience method we would have needed to use a method called `addEventHandler`.

`addEventHandler` requires two parameters, of type `EventType` and `EventHandler` respectively.

The interface `EventHandler` is a functional interface, with a single **`abstract`** method `handle`. This method will contain the instructions that tell the button what to do when the event occurs.

In our case we would declare an object of type `EventHandler` as follows:

```
EventHandler<ActionEvent> handler =
                   e -> pushMeLabel.setText("You entered: " + pushMeText.getText());
```

As you can see, `EventHandler` is a generic interface and requires us to tell it the type of objects it will be dealing with. In this case it will be an `ActionEvent` —this is the event that occurs when a button is pushed. At present there is only one type of `ActionEvent`, which is `ActionEvent.ACTION`, so that will be our first parameter. In Chap. 16 you will see that for other events, such as `MouseEvent`, there are many types available such as `MouseEvent.MOUSE_PRESSED`, `MouseEvent.MOUSE_RELEASED`, `MouseEvent.MOUSE_MOVED` and so on, although in most cases we once again won't have to worry about them, as convenience methods are available for each one.

Notice that we have been able to use a lambda expression to send the code for the `handle` method, as `EventHandler` is a functional interface.

So now we can write the code for adding the `EventHandler` as follows:

```
pushMeButton.addEventHandler(ActionEvent.ACTION, handler);
```

Of course this could have been done in one line, without declaring the `handler` object first, but doing it as we have should have made clear exactly what is happening.

13.9 Polymorphism and Polymorphic Types

In this chapter we have seen how an object can have a number of different types. As well as being the type of its own class, it is also the type of any superclasses in the class hierarchy, and the type of any interfaces that it implements. You will recall that we have used the term *polymorphism* to refer to the phenomenon whereby methods and operators can have the same name, but exhibit different behaviour. A language like Java that allows objects to be of more than one type is said to support **polymorphic types**.

In general, polymorphism is an important feature of object-oriented languages, and it is worth our while spending a little more time summarizing the different ways in which polymorphism can be achieved.

13.9.1 Operator Overloading

We have seen several examples of operators that are overloaded, and that can therefore behave differently depending upon the type of data they are manipulating. The + operator, for example, can be used for the concatenation of strings as well as for addition. The division operator, `/`, can be used for integer division as well as for division of real numbers. The particular function performed is determined by the operands. It should be noted that Java, as opposed to some other languages, does not allow the user to overload operators.

13.9.2 Method Overloading

We are now very familiar with this type of polymorphism, whereby several methods in a class have the same name and are distinguished by their parameter lists. This is particularly useful for defining a number of different constructors, and you will find many examples in the Java libraries where a number of constructors are provided for a particular class. Just one of many examples is the `String` class where, among others, a constructor is provided with no parameters to create an empty string, and a constructor is provided that accepts a string value in order to initialize the new `String` object.

13.9.3 Method Overriding

As we have seen, method overriding is a way of achieving polymorphism by redefining a method of a class within a subclass. Here methods are distinguished not by their parameter lists but by the object with which they are associated. As we have described in this and previous chapters, this is made even more powerful by having the ability to define **`abstract`** methods and therefore guaranteeing the existence of these methods further down the hierarchy.

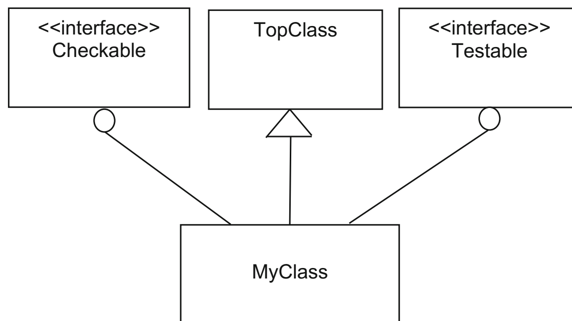
13.9.4 Type Polymorphism

Type polymorphism refers to the technique by which values of different types can be handled in a uniform way. Examples of this are the `System.out.print` and the `System.out.println` methods which are set up to accept objects of many different types—**`int`**, **`char`**, **`double`**, `String` and so on. This is achieved, of course, by defining many different (overloaded) methods. In this way it is possible for a single method to appear to accept an object of any type. This is known as **parametric polymorphism**.

Another way to create a method that accepts more than one type was demonstrated in the `EmployeeTester` program of Sect. 9.6, where a method was set up to accept an object of type `Employee`. We saw there how it was possible to call this method and send in objects of any subclass of `Employee` such as `FullTimeEmployee` or `PartTimeEmployee`. Because objects of a subclass are of the same type as the superclass, a method can effectively receive parameters of more than one type. As we saw, the behaviour of methods of these classes might be different, and hence this can be regarded as a kind of polymorphism—this is known as **subtype polymorphism**.

13.10 Self-test Questions

1. Consider the UML design below of a class called `MyClass`.



   Write the header of the `MyClass` class.

2. Consider the following interface, called `SomeInterface`:

```
public interface SomeInterface
{
          public double SomeMethod(double x);
}
```

The following class is developed as shown below:

```
public class SomeClass implements SomeInterface
{
          private double y;

          public double SomeMethod(int x)
          {
                return 2.5 * x;
          }
}
```

Explain why this class will not compile, and explain how it should be amended in order to rectify the problem.

3. What is meant by a *functional interface*? Explain how lambda expressions are used in conjunction with functional interfaces.
4. A lambda expression is characterized by the arrow symbol `->`. What is placed the left of this symbol, and what is placed to the right?
5. Consider the following interface:

The *Testable* interface

```
public interface Testable
{
      public void test();
}
```

Now consider the following program:

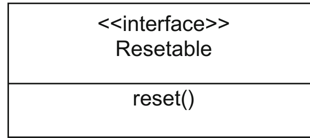
WontCompile

```
public class WontCompile
{
    public static void main(String[] args)
    {
        int x = 2;

        helper( () -> {
                        x = x * 2;
                        System.out.println(x);
                     }
              );
    }

    static void helper(Testable objectIn)
    {
        objectIn.test();
    }
}
```

As its name suggests, this program gives rise to a compiler error. Can you see why?

6. In order to tackle this exercise make sure that the classes `Oblong` and `BankAccount` have been copied from the website and placed in the correct directory for your compiler to access them.

Now consider the following `Resetable` interface:



The `reset` method takes no parameters and returns no value. The intention of the `reset` method is to reset the object data to some basic initial value. For example, an `Oblong` object might be reset by having both sides set to 1.

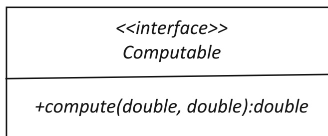
(a) Write the code for the `Resetable` interface.

(b) Write a tester program to check the `Testable` interface. An outline of the tester is given below:

Outline of a *TestResetable* program

```
public class TestResetable
{
        public static void main(String[] args)
        {
           // create an Oblong object and BankAccount Object

           // Make a deposit into the BankAccount object

           /* call the resetObject method with a lambda expression that sets
              the length and height of the Oblong to 1 */

           /* call the resetObject method with a lambda expression that sets
              the balance of the BankAccount object to zero. You can't do this
              directly because there is no setBalance method - but you can
              withdraw the total amount that is in the account */

           /* display the length and height of the oblong (which should now
              be 1) and the balance of the bank account (which should now be
              zero) */
        }

     // write a resetObject method as follows:

      static void resetObject(Resetable objectIn)
      {
               objectIn.reset();
      }
}
```

7. (a) Write the code for an interface, `Computable`, which has a single method, `compute`, that accepts two **`double`**s and returns a **`double`**:



(b) Consider the following program; a lambda expression has been replaced by a comment:

```
public class TestComputable
{
    public static void main(String[] args)
    {
        Computable comp = // lambda expresion here ;
        printResult(comp);
    }

    public static void printResult(Computable compIn)
    {
        System.out.println(compIn.compute(10, 5));
    }
}
```

Replace the comment with a lambda expression that causes the `compute` method to return the sum of the two `doubles` it receives, and then test out your program with some different values in the `printResult` method.

(c) Change the lambda expression so that it performs other calculations such as subtraction or multiplication.

8. Explain, with examples, what is meant by the following terms, and how they are implemented in Java:

   (a) generic class;

   (b) upper bound;

   (c) wildcard.

9. (a) Adapt the code for the `Computable` interface that you developed in question 7 so that it is generic interface, that is not restricted to dealing with **`double`**s only.

   (b) Rewrite the `TestComputable` program from question 7 (b) so that it uses your generic interface with number types other than **`double`**. Don't forget you will have to use the wrapper classes such as `Double` and `Integer`.

   (c) Adapt your generic interface so it is restricted to accepting only `Number` types (remember that classes such as `Integer`, `Float` and `Double` are all sub-types of `Number`).

10. Consider the following two classes:

```
public class HighClass
{
        private int num;
        private String str;

        public HighClass()
        {
                num = 10;
                str = "Hello ";
        }

        public HighClass(int numIn, String strIn)
        {
                num = numIn;
                str = strIn;
        }

        public void display(int mult)
        {
                System.out.println(100 + mult * num);
        }

        public void display(String strIn)
        {
                System.out.println(str + strIn);
        }
}
```

```
public class LowClass
{
        private char ch;

        public LowClass()
        {
                super();
                ch = 'Q';
        }

        public void display(String strIn)
        {
                System.out.println(strIn + ch);
        }
}
```

Give examples from these classes of:

(a) operator overloading;
(b) method overloading;
(c) method overriding;
(d) type polymorphism.

13.11 Programming Exercises

1. Implement a few of the programs from this chapter in order to make sure that you fully understand the concepts involved.

2. A Customer class is being developed for a small business. The suggested code for this class is shown below:

```
public class Customer
{
	private String customerId;
	private String name;
	private double creditLimit;

	public Customer(String IdIn, String nameIn, double limitIn)
	{
		customerId = IdIn;
		name = nameIn;
		creditLimit = limitIn;
	}

	public String getCustomerId()
	{
		return customerId;
	}


	public String getName()
	{
		return name;
	}

	public double getCreditLimit()
	{
		return creditLimit;
	}

	public void setCreditLimit(double limitIn)
	{
		creditLimit = limitIn;
	}
}
```

The owner of the business requires that the customer ID must comprise a single letter followed by exactly 3 numbers.

(a) Create a CheckableCustomer class that inherits from the Customer class and implements the Checkable interface that we developed in Sect. 13.3. The class will override the check method according to the above rule.
(b) Adapt the Checker program from Sect. 13.3 so that it now checks the validity of a CheckableCustomer object, according to the above rule.
(c) Adapt the InnerClassDemo program from Sect. 13.4 so that Check-ableCustomer is now specified as an inner class.
(d) Adapt the AnonymousClassDemoVersion1 and Anony-mousClassDemoVersion2 programs from Sect. 13.5 so that an anonymous class is used instead of an inner class to check the validity of a Customer Object.
(e) Finally, adapt program LambdaDemo program from Sect. 13.6 to use a lambda expression to check the validity of a Customer object.

3. Implement the `Resetable` interface and the `TestResetable` class from self-test question 6.

4. Implement the `Computable` interface and the test classes you developed in self-test question 7.

5. Implement the generic version of the above classes as described in self-test question 9.

6. In Sect. 13.7 the program `TestGenericInterfaceVersion2` demonstrated the use of one of the out-of-the-box interfaces (`Function`) listed in Table 13.1. Design and implement some more programs to demonstrate the use of some of the other functions listed in that table.

Exceptions 14

Outcomes:

By the end of this chapter you should be able to:

- *explain the term* ***exception***;
- *distinguish between* ***checked*** *and* ***unchecked*** *exception classes in Java*;
- *claim an exception using a* ***throws*** *clause*;
- *throw an exception using a* ***throw*** *command*;
- *catch an exception in a* ***try…catch*** *block*;
- *use a* ***finally*** *block or a* ***try-with-resources*** *construct to deal with clean-up issues*;
- *use the* `Optional` *class to avoid* `NullPointerException` *errors*;
- *define and use their own exception classes.*

14.1 Introduction

One way in which to write a program is to assume that everything proceeds smoothly and as expected—users input values at the correct time and of the correct format, files are never corrupt, array indices are always valid and so on. Of course this view of the world is very rarely accurate. In reality, unexpected situations arise that could compromise the correct functioning of your program.

Programs should be written that continue to function even if unexpected situations should arise. So far we have tried to achieve this by carefully constructed **`if`** statements that send back error flags, in the form of **`boolean`** values, when appropriate. However, in some circumstances, these forms of protection against undesirable situations prove inadequate. In such cases the *exception handling* facility of Java must be used.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_14

14.2 Pre-defined Exception Classes in Java

An **exception** is an event that occurs during the life of a program which could cause that program to behave unreliably. You can see that the events we described in the introduction fall into this category. For example, accessing an array with an invalid index could cause that program to terminate.

Each type of event that could lead to an exception is associated with a pre-defined *exception class* in Java. When a given event occurs, the Java run-time environment determines which exception has occurred and an object of the given exception class is generated. This process is known as **throwing** an exception.

These exception classes have been named to reflect the nature of the exception. For example, when an array is accessed with an illegal index an object of the `ArrayIndexOutOfBoundsException` class is thrown.

All exception classes inherit from the base class `Throwable` which is found in the `java.lang` package. These subclasses of `Throwable` are found in various packages and are then further categorized depending upon the type of exception. For example, the exception associated with a given file not being found (`FileNotFoundException`) and the exception associated with an end of file having been reached (`EOFException`) are both types of input/output exceptions (`IOException`), which reside in the `java.io` package. Figure 14.1 illustrates part of this hierarchy.

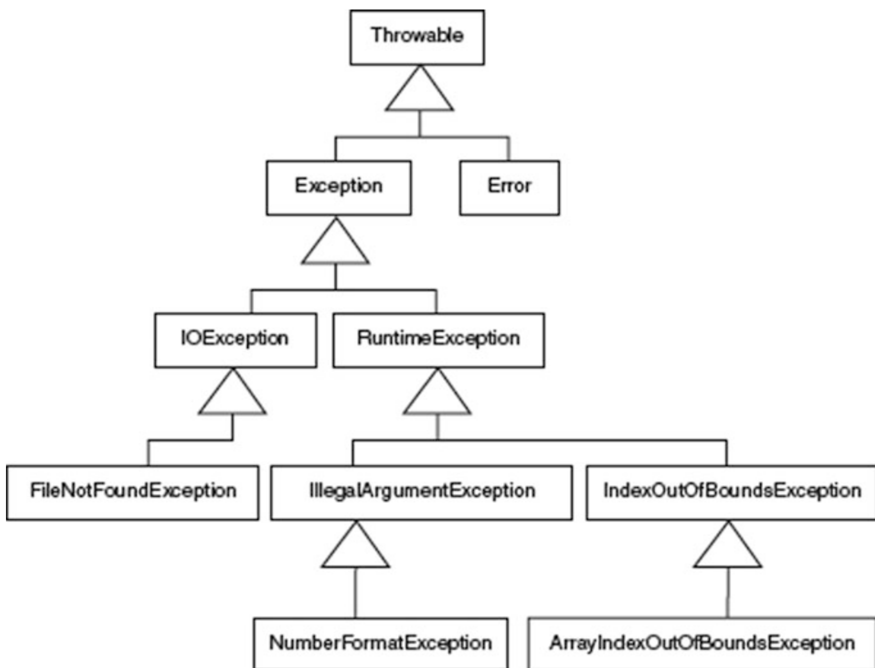


Fig. 14.1 A sample of Java's pre-defined exception class hierarchy

As you can see from Fig. 14.1, there are two immediate subclasses of Throwable: Exception and Error. The Error class describes internal system errors that are very unlikely ever to occur (so called "hard" errors). For example, one subclass of Error is VirtualMachineError where some error in the JVM has been detected. There is little that can be done in the way of recovery from such errors other than to end the program as gracefully as possible. All other exceptions are subclasses of the Exception class and it is these exceptions that programmers deal with in their programs. The Exception class is further subdivided. The two most important subdivisions are shown in Fig. 14.1, IOException and RuntimeException.

The RuntimeException class deals with errors that arise from the logic of a program. For example, a program that converts a string into a number, when the string contains non-numeric characters (NumberFormatException) or accesses an array using an illegal index (ArrayIndexOutOfBoundsException).

The IOException class deals with external errors that could affect the program during input and output. Such errors could include, for example, the keyboard locking, or an external file being corrupted.

Since nearly every Java instruction could result in a RuntimeException error, the Java compiler does not flag such instructions as potentially error-prone. Consequently these types of errors are known as unchecked exceptions. It is left to the programmer to ensure that code is written in such a way as to avoid such exceptions; for example, checking an array index before looking up a value in an array with that index. Should such an exception arise, it will be due to a program error and will not become apparent until runtime.

The Java compiler *does*, however, flag up those instructions that may generate all other types of exception (such as IOException errors) since the programmer has no means of avoiding such errors arising. For example, an instruction to read from a file may cause an exception because the file is corrupt. No amount of program code can prevent this file from being corrupt. The compiler will not only flag such an instruction as potentially error-prone, it will also specify the exact exception that could be thrown. Consequently, these kinds of errors are known as checked exceptions. Programmers have to include code to inform the compiler of how they will deal with checked exceptions generated by a particular instruction, before the compiler will allow the program to compile successfully.

14.3 Handling Exceptions

Consider a simple program that allows the user to enter an aptitude test mark at the keyboard; the program then informs the user if he or she has passed the test and been allowed on a given course. We could use the nextInt method (from either our EasyScanner class, or the original Scanner class) to allow the user to enter this mark. However, in order to show you how exceptions can be dealt with in your programs, we will not take this approach—instead we will devise our own class,

TestException, that will contain a class method called getInteger. Before we do that, here is the outline of the main application:

```
public class TestException
{
    // this method is declared 'static' as it is a class method
    public static int getInteger()
    {
            // code for method goes here
    }
}
```

Now let's look at an outline for the TestException class.

The getInteger method must allow the user to enter an integer at the keyboard and then return that integer. There are many ways we could try and read an integer from the keyboard. As we have said, rather than make use of the nextInt method in the Scanner class, the approach we will take here will be to use a rather low-level method called read in the System.in object. So far we have used the System.out object to display information on the screen, but we have not explored the System.in object. This object is an object of the InputStream class that you will find out more about in Chap. 18.

You will remember from Chap. 2 that each character on the keyboard is represented by a Unicode number. For countries in which the standard western alphabet is used, the lower case letters 'a' through to 'z' are represented by the Unicode values 97 through to 122 inclusive. Special characters also have Unicode values. For example, the carriage return character has a Unicode value 13. The InputStream class provides a read method that is a bit like the next method of the Scanner class, except that it treats the string as a series of Unicode numbers. Each number is considered to be of type byte, so that the string itself is an array of bytes. Figure 14.2 illustrates the effect of the read method when someone enters the word "hello" at the keyboard.

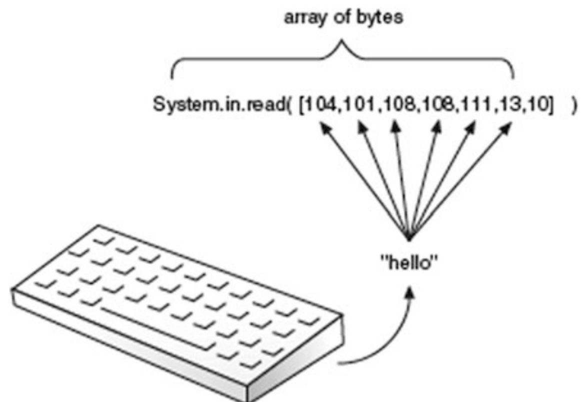


Fig. 14.2 The 'read' method stores characters entered at the keyboard as an array of bytes

Notice that the array of bytes is not returned as a value but instead sent as a parameter. Also note that the new-line character is given a Unicode value of 10.

The `getInteger` method will first have to take this array of bytes and convert it into a `String`. Luckily a version of the `String` constructor returns a `String` object from an array of bytes. We then remove any trailing spaces at the end of the `String`; this can be done with the `String` method `trim` as follows:

```
byte [] buffer = new byte[512]; // declare a large byte array
System.in.read(buffer); // characters entered stored in array
String s = new String (buffer); // make string from byte array
s = s.trim(); // trim string
```

Now, finally, we have to convert this string into an integer. We can use the `parseInt` method of the `Integer` class to do this:

```
int num = Integer.parseInt(s); // converts string to an 'int'
```

Our `TestException` class now looks like this:

```
// this is a first attempt, it will not compile!
public class TestException
{
  public static int getInteger()
   {
            byte [] buffer = new byte[512];
            System.in.read(buffer);
            String s = new String (buffer);
            s = s.trim();
            int num = Integer.parseInt(s);
            return num; // send back the integer value
   }
}
```

Unfortunately, as things stand, this class will *not* compile. The cause of the error is in the `getInteger` method, in particular the way we used the `read` method of `System.in`. Whenever this method is used, the Java compiler *insists* that we be very careful. To understand this better, take a look at the header for this `read` method, taken from the Java language specification, in particular the part we have emboldened:

```
public int read (byte[] b) throws IOException
```

Up until now you have not seen a method header of this form. The words **`throws`** `IOException` are the new bits in this method header. In Java this is known as a method **claiming an exception**.

14.3.1 Claiming an Exception

The term claiming an exception refers to a given method having been marked to indicate that it will pass on an exception object that it might generate. So the term **throws** IOException means that the method *could* cause an input/output exception in your program. The type of error that could take place while data is being read includes the loss of a network connection or a file being corrupted, for example.

Remember, when an exception occurs, an exception object is created. This is an unwanted object that could cause your program to fail or behave unpredictably, and so should be dealt with and not ignored. Rather than dealing with this exception object *within* the read method, the Java people decided it would be better if callers of this method dealt with the exception object in whatever way they felt was suitable. In effect, they *passed the error* on to the caller of the method (see Fig. 14.3).

As the type of exception generated (IOException) is not a subclass of RuntimeException, it is an example of a *checked exception*. In other words, the compiler *insists* that if the read method is used, the programmer deals with this exception in some way, and does not just ignore it as we did originally. That is why we had a compiler error initially. There are always two ways to deal with an exception:

1. deal with the exception within the method;
2. pass on the exception out of the method.

The developers of the read method decided to pass on the exception, so now our getInteger method has to decide what to do with this exception. In a while we will show you how to deal with an exception within a method, but for now we will just make our getInteger method pass on the exception too. We do this by simply adding a **throws** clause to our method:

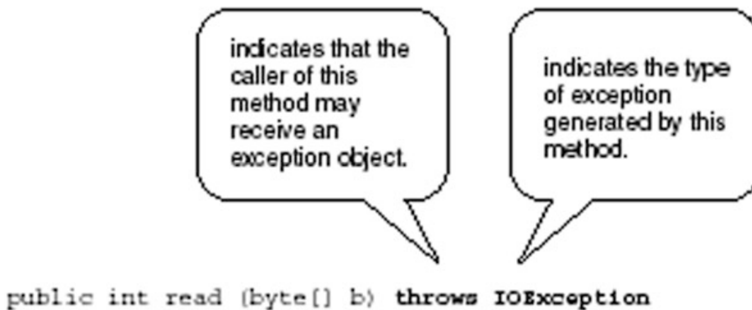


Fig. 14.3 The 'throws' clause can be added to a method header to indicate that the method may generate an exception

```
import java.io.IOException;
public class TestException
{
   // adding this throws clause will allow this method to compile
   public static int getInteger() throws IOException
   {
        // as before
   }
}
```

Notice that, as the `IOException` class is in the `io` package, we now need the following **`import`** statement at the top of this class:

```
import java.io.IOException;
```

Now that the `getInteger` method has claimed the `IOException`, it *will* compile as we have not just *ignored* the exception, we have made a *conscious decision* to pass the exception on to any method that calls this `getInteger` method. Now, let's think about developing our aptitude test program.

```
// something wrong here!
public class AptitudeTest
{
   public static void main (String[] args)
   {
        int score;
        System.out.print("Enter aptitude test score: ");
        score = TestException.getInteger(); // calling class method
        // test score here
   }
}
```

Can you see what the problem with this program is? Well, this program will not compile now as the `main` method makes a call to our `getInteger` method, and this method may now throw an `IOException`! The `main` method now has to deal with this exception and not just ignore it. For the time being, to keep the compiler happy, we will just let the `main` method throw this exception as well. Here is the code:

AptitudeTest

```
import java.io.IOException;
public class AptitudeTest
{
       // this main method will throw out any IOExceptions
       public static void main (String[] args) throws IOException
       {
              int score;
              System.out.print("Enter aptitude test score: ");
              // the 'getInteger' method may throw an IOException
              score = TestException.getInteger();
              if (score >= 50)
              {
                     System.out.println("You have a place on the course!");
              }
              else
              {
                     System.out.println("Sorry, you failed your test");
              }
       }
}
```

Dealing with the exception in the way we have is not a very good idea. We have effectively continually passed on the exception object until it gets thrown out of our program to the operating system. This may cause the program to terminate when such an exception occurs. Before we deal with this problem let us show you a test run. Take a look at it as something very interesting happens.

```
Enter aptitude test score: 12w
java.lang.NumberFormatException: 12w
at java.lang.Integer.parseInt(Integer.java:418)
at java.lang.Integer.parseInt(Integer.java:458)
at TestException.getInteger(TestException.java:10)
at AptitudeTest.main(AptitudeTest.java:11)
```

As you can see, when asked to enter an integer, the user inadvertently added a character into the number (12**w**). This has led to an exception being generated and thrown out of our program. Again, looking at the output generated, you can see that when an exception is generated in this way the Java system gives you quite a lot of information. This information includes the name of the method that threw the exception, the class that the method belongs to, the line numbers in the source files where the error arose, and the type of exception that was thrown. Such information is referred to as the **stack trace** of the exception.

Look at the name of the exception that is thrown. It's not the one we were worried about, `IOException`, but `NumberFormatException`. This exception is raised when trying to convert a string into a number when the string contains non-numeric characters, as we were trying to do in this case within our `getInteger` method:

```
public static int getInteger() throws IOException
{
    // some code here
    int num = Integer.parseInt(s); /* will cause a NumberFormatException
                               if string s contains non-numeric characters*/
}
```

Why didn't the compiler warn us about this when we first used the `parseInt` method in our implementation of `getInteger`? Well, the reason is that the exception that could arise (`NumberFormatException`) is a subclass of `RuntimeException` and so is unchecked!

Notice that run-time exceptions do not need to be claimed in method headers in order for them to be thrown. For example, although the following is valid in Java, it is not *necessary* to claim the `NumberFormatException` in the header.

```
/* multiple exceptions can be claimed in the method header as follows by separating exception
  names with commas. However run-time exceptions do not need to be claimed in this way */

public static int getInteger() throws IOException, NumberFormatException
{
      // some code here
}
```

The way we have dealt with exceptions so far has not been very effective. As you can see from the test run of program 14.1, continually throwing exceptions up to the calling method does not really solve the problem. It may keep the compiler happy, but eventually it means exception objects will escape from your programs and cause them to terminate. Instead, it is better at some point to handle an exception object rather than throw it. In Java this is known as **catching an exception**.

14.3.2 Catching an Exception

One route for an exception object is *out* of the current method and *up to* the calling method. That's the approach we used in the previous section. Another way out for an exception object, however, is into a **catch** block. Once an exception object is trapped in a **catch** block, and that block ends, the exception object is effectively terminated. In order to trap the exception object in a **catch** block you must surround the code that could generate the exception in a **try** block. The syntax for using a **try** and **catch** block is as follows:

```
try
{
    // code that could generate an exception
}
catch (Exception e) // type of exception must be specified as a parameter
{
    // action to be taken when an exception occurs
}
// other instructions could be placed here
```

There are a few things to note before we show you this **try**…**catch** idea in action. First, any number of lines could be placed within the **try** block, and more than one of them could cause an exception. If none of them causes an exception the **catch** block is missed and the lines following the **catch** block are executed. If any one of them causes an exception the program will leave the **try** block and look for a **catch** block that deals with that exception.

Once such a **catch** clause is found, the statements within it will be executed and the program will then *continue* with any statements that follow the **catch** clause—it will *not* return to the code in the **try** clause. Look carefully at the syntax for the **catch** block:

```
catch (Exception e)
{
     // action to be taken when an exception occurs
}
```

Table 14.1 Some methods of the *Exception* class

Method	Description
printStackTrace	Prints (onto the console) a stack trace of the exception
toString	Returns a detailed error message
getMessage	Returns a summary error message

This looks very similar to a method header. You can see that the **catch** block header has one parameter: an object, which we called e, of type Exception. Since *all* exceptions are subclasses of the Exception class, this will catch *any* exception that should arise. However, it is better to replace this exception class with the *specific* class that you are catching so that you can be certain *which* exception you have caught. As there may be more than one exception generated within a method, there may be more than one **catch** block below a **try** block—each dealing with a different exception. When an exception is thrown in a **try** block, the **catch** blocks are inspected in order—the first matching **catch** block is the one that will handle the exception.

Within the **catch** block, programmers can, if they choose, interrogate the exception object using some Exception methods, some of which are listed in Table 14.1.

With this information in mind we can deal with the exceptions in the previous section in a different way. All we have to decide is where to catch the exception object. For now we will leave the getInteger method as it is, and catch offending exception objects in the main method of the AptitudeTest2 program. Take a look at it and then we will discuss it.

AptitudeTest2

```
import java.io.IOException;

public class AptitudeTest2
{
    public static void main (String[] args)
    {
       try
       {
              int score;
              System.out.print("Enter aptitude test score: ");
              // getInteger may throw IOException or NumberFomatException
              score = TestException.getInteger();
              if (score >= 50)
              {
                     System.out.println("You have a place on the course!");
              }
              else
              {
                     System.out.println("Sorry, you failed your test");
              }
       }
       // if something does goes wrong!
       catch (NumberFormatException e)
       {
              System.out.println("You entered an invalid number!");
       }
       catch (IOException e)
       {
              System.out.println(e); // calls toString method
       }
       // even if no exception thrown/caught, this line will be executed
       System.out.println("Goodbye");
    }
}
```

Notice that by catching an offending exception object there is no need to pass that object out of the method by raising that exception in the method header. Since we catch the `IOException` here, the `throws IOException` clause can be removed from the header of `main`. In this program we have chosen to print out an error message if an `IOException` is raised (by implicitly calling the `toString` method), whereas we have chosen to print our own message if a `NumberFormatException` is raised. Now look at a sample test run of the program:

```
Enter aptitude test score: 12w
You entered an invalid number!
Goodbye
```

As you can see the user once again entered an invalid integer, but this time the program did not terminate. Instead the exception was handled with a clear message to the user, after which the program continued to operate normally.

14.4 The 'finally' Clause

From the previous sections you can see that three courses of action may now take place in a **try** block:

1. the instructions within the **try** block are all executed successfully;
2. an exception occurs within the **try** block; the **try** block is exited and a matching **catch** block is found for this exception;
3. an exception occurs within the **try** block; the **try** block is exited but no matching **catch** block is found for this exception, so the exception is thrown from the method.

It may be the case that, no matter which of these courses of action take place, you wish to execute some additional instructions before the method terminates. Often such a scenario arises when you wish to carry out some clean-up code, such as closing a file or a network connection that you have opened in the **try** block. We will see some examples of these scenarios in later chapters. The **finally** clause allows us to do this. The syntax for the **finally** clause is as follows:

```
try
{
      // code that could generate an exception
}
catch (Exception e) /* if one or more 'catch' clauses are specified,
                     they must be given before the 'finally' clause */
{
      // action to be taken when an exception occurs
}
finally
{
      // cleanup code can go here
}

// other instructions could be placed here
```

Notice that when **catch** clauses are specified, the **finally** clause must come directly *after* such clauses. If no such **catch** clauses are specified, the **finally** clause must follow directly after the **try** clause. Now, when the code in the **try** block is executed the following three courses of action can take place:

1. the instructions within the **try** block are all executed successfully; if there are any **catch** blocks specified they are skipped and the code in the **finally** block is executed, followed by any code after the **finally** block;
2. an exception occurs within the **try** block; the **try** block is exited and a matching **catch** block is found for this exception, after which the code in the **finally** block is executed, followed by any code after the **finally** block;
3. an exception occurs within the **try** block; the **try** block is exited but no matching **catch** block is found for this exception so the code in the **finally** block is executed—after which the method terminates and the appropriate exception is thrown by the given method.

We will use a very simple example of closing a `Scanner` resource as a way to demonstrate how the **finally** clause works under the three scenarios outlined above.

We have seen many examples of creating `Scanner` objects. `Scanner` objects point to a resource, in our case this resource has been the keyboard (`System.in`). We would not ordinarily close this resource as this would be closed automatically once the program terminates, but in the `ClosingAResourceUsingFinally` program below we will close it explicitly in a **finally** clause. Take a look at the code and then we will discuss it.

ClosingAResourceUsingFinally

```
import java.util.Scanner;

public class ClosingAResourceUsingFinally
{
    public static void main(String[] args)
    {
       Scanner keyboard = new Scanner (System.in); // open a Scanner resource
       try
       {
            System.out.println("START TRY");
            String[] colours = {"RED","BLUE","GREEN"}; // initialise array
            System.out.print("Which colour? (1,2,3): ");
            String pos = keyboard.next();
            // next line could throw NumberFormatException
            int i = Integer.parseInt(pos);
            // next line could throw ArrayIndexOutOfBoundsException
            System.out.println(colours[i-1]);
            System.out.println("END TRY");
        }
       // include a catch only for ArrayIndexOutOfBoundsException
       catch(ArrayIndexOutOfBoundsException e)
       {
            System.out.println("ENTER CATCH ");
            System.out.println(e);
       }
       // this block will always be executed
       finally
       {
            System.out.println("ENTER FINALLY");
            keyboard.close(); // Scanner resource closed
            System.out.println("Scanner CLOSED");
       }
       System.out.println("Goodbye");

    }
}
```

Most of this code should be self-explanatory. Notice that we have provided a **catch** block for the ArrayIndexOutOfBoundsException but we have not provided a **catch** block for the NumberFormatException, so such an exception would be thrown from main should it arise. Also, notice that we displayed messages to indicate when we are in each of the **try**, **catch** and **finally** blocks. Finally, notice how we close a Scanner resource, in the **finally** block of code, by calling the Scanner object's close method:

```
// this block will always be executed
finally
{
       System.out.println("ENTER FINALLY");
       keyboard.close(); // Scanner resource closed
       System.out.println("Scanner CLOSED");
}
```

Here is one test run:

START TRY
Which colour? (1,2,3): ***2***
BLUE
END TRY
ENTER FINALLY
Scanner CLOSED
Goodbye

Here the user enters a valid colour number, so the **try** block completes successfully. The **catch** block is skipped and the **finally** block is executed, which closes the Scanner resource. Following the **finally** block the last "Goodbye" instruction is executed.

Here is another test run:

START TRY
Which colour? (1,2,3): ***4***
ENTER CATCH
java.lang.ArrayIndexOutOfBoundsException: 3
ENTER FINALLY
Scanner CLOSED
Goodbye

Here the user enters an invalid colour number, the **try** block does not complete as an ArrayIndexOutOfBoundsException is thrown. A matching **catch** block is found for this exception and executed. Upon completion of this **catch** block the program continues with the code in the **finally** block, which closes the Scanner resource. Following the **finally** block the last "Goodbye" instruction is executed.

Here is the last test run:

```
START TRY
Which colour? (1,2,3): 2c
ENTER FINALLY
Scanner CLOSED
Exception in thread "main" java.lang.
NumberFormatException:
For input string: "2c"
at java.lang.NumberFormatException.forInputString
(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:456)
at java.lang.Integer.parseInt(Integer.java:497)
at
ClosingAResourceUsingFinally.main
(ClosingAResourceUsingFinally.java:11)
```

In this case, the user entered an invalid number causing a `NumberFormatException`—so the **try** block did not complete successfully. However, there is no **catch** block provided for this exception. Without a **finally** clause this would have led to program termination *immediately* as the exception escapes from `main`. We have a **finally** clause, however, which closes the `Scanner` resource. The program then terminates with the offending exception (`NumberFormatException`).

These three test runs match the three scenarios we identified for the **try**/**catch**/**finally** blocks earlier. You will notice from the three test runs above that, if the instructions inside the **finally** clause were written as normal below the **catch** clause (without putting them into a **finally** block), the first two test runs would have produced exactly the same result. This is because code following a **catch** block is always executed if no exception is thrown, or if an exception is thrown and a matching **catch** clause is found and executed. Only in scenario three, when an exception is thrown and no matching **catch** is found (perhaps because no **catch** clauses were specified), does the **finally** clause really make a difference to program flow.

You may well come across the third scenario when developing your programs, and we shall see examples in later chapters, so the **finally** clause could be used here for clean-up code. Using the **finally** clause in scenarios one and two is optional.

14.5 The 'Try-with-Resources' Construct

In the previous section we saw how a **finally** clause can be used to close a resource before exiting a program. In the test runs of `ClosingAResourceUsingFinally` we saw that, once the **finally** code is executed, any uncaught exception is reported

(`NumberFormatException` in the test runs of Sect. 14.4). However, you should note that if the code in the **`finally`** clause itself throws an exception it is *this* exception that is thrown from the method rather than the original offending exception. The `close` method of `Scanner`, for example, would itself throw an `IOException` if the system was unable to close the `Scanner` resource. This is not ideal as the original exception is probably more appropriate to report.

One of the more recent developments in Java allows for us to avoid writing **`finally`** clauses to close a resource, but instead adapt the **`try`** clause for this purpose. This is known as **try-with-resources**. The *try-with-resources* clause automatically closes a specified resource (or resources) for us and suppresses any exceptions that might arise from doing so, leaving any original uncaught exceptions free to be reported. The program below re-writes the `ClosingAResourceUsingFinally` program to make use of this try-with-resources clause. Take a look at it and then we will discuss it.

ClosingAResourceUsingTryWithResources

```
import java.util.Scanner;

public class ClosingAResourceUsingTryWithResources
{
   public static void main(String[] args)
   {
      try (Scanner keyboard = new Scanner (System.in)) // open a Scanner resource here
      {
            System.out.println("START TRY");
            String[] colours = {"RED","BLUE","GREEN"}; // initialise array
            System.out.print("Which colour? (1,2,3): ");
            String pos = keyboard.next();
            // next line could throw NumberFormatException
            int i = Integer.parseInt(pos);
            // next line could throw ArrayIndexOutOfBoundsException
            System.out.println(colours[i-1]);
            System.out.println("END TRY");
        }
      // include a catch only for ArrayIndexOutOfBoundsException
      catch(ArrayIndexOutOfBoundsException e)
      {
            System.out.println("ENTER CATCH ");
            System.out.println(e);
      }
      // we have removed the finally clause
      System.out.println("Goodbye");

   }
}
```

You can see that, when using a try-with-resources clause, we include the resource declaration (in this case the creation of a `Scanner` object) in round brackets straight after the try keyword:

```
try (Scanner keyboard = new Scanner (System.in)) // Scanner resource created in brackets
{
      // previous code as before
}
```

This Scanner resource is now treated as an object whose scope is the **try** block.

If code in the **try** block completes, with no exception occurring, the given resource (our Scanner object keyboard in this case) is automatically closed. If an exception does occur during the **try** block, the given resource is still automatically closed, whether or not that exception is caught. Unlike a **finally** block, however, if an exception is thrown from the **try** block but not caught and an exception is then also thrown by the close method of the given resource, the exception thrown by the close method is suppressed and the original uncaught exception from the **try** block is reported.

So, in the example above, if a NumberFormatException occurs that has no **catch** block and closing the Scanner resource also throws an IOException, the IOException is suppressed and the NumberFormatException is thrown.

Note, we can create any object (or objects) within the **try-with-resources** bracket as long its type is a class that implements the Closeable (or AutoCloseable) interfaces. Classes that implement these interfaces all include a close method. Scanner is one example of such a class. We will see further examples Chap. 18 when we study file-handling.

14.6 Null-Pointer Exceptions

One of the most common exception types you will come across when programming is the dreaded NullPointerException. This is an example of an unchecked RuntimeException and is a very common cause of program errors. It occurs when we try to call the method of an object, but the object itself contains a **null** value rather than valid object data. As a very simple example, consider the following instructions that make use of our BankAccount class of Chap. 8:

```
BankAccount acc1 = null; // BankAccount reference set to null
System.out.println(acc1.getAccountName()); // 'getAccountName' will cause a NullPointerException
```

Here the BankAccount object reference, acc1, is set to the **null** value. We should not call any methods on this object as it is not yet referencing a valid BankAccount object. However, we have then made an attempt to call the getAccountNumber method on this object reference and, consequently, the program will throw a NullPointerException.

While we are unlikely to make such obvious errors as the one demonstrated above, **null** values can find their way into your programs in very subtle ways and NullPointerExceptions can then easily arise.

Think back to the Bank collection class of Sect. 8.8.1, for storing a collection of BankAccount objects. Here is a reminder of its getItem method that retrieves a BankAccount object given its account number.

```
// returns the account assciated with a particular account number
public BankAccount getItem(String accountNumberIn)
{
    int index = search(accountNumberIn);
    if(index != -999) // check that account exists
    {
         return list.get(index); // return valid BankAccount object
    }
    else
    {
         return null; // no such account so return null value
    }
}
```

You can see that, if the given account number exists in the list, the associated BankAccount object is returned:

```
if(index != -999) // check that account exists
{
     return list.get(index); // return valid BankAccount object
}
```

If, however, if no such account exists a **null** value is returned to indicate this:

```
else
{
     return null; // no such account so return null value
}
```

Now consider the following simple program that adds two BankAccount objects to the Bank collection class and then asks the user to enter an account number to search for:

NullPointerDemo

```
public class NullPointerDemo
{
       public static void main(String[] args)
       {
              // create Bank collection
              Bank myBank = new Bank();
              // add two BankAccount objects
              myBank.addAccount("001", "Aaron");
              myBank.addAccount("002", "Quentin");
              // request user for account number
              System.out.print("Enter account number to search for: ");
              String account = EasyScanner.nextString(); // we are using our EasyScanner class here
              // retrieve account name accociated with this account number
              // this may throw a NullPointerException!
              System.out.println(myBank.getItem(account).getAccountName());

       }
}
```

Here is a sample program where the user enters a valid account number to search for:

Enter account number to search for: ***001***
Aaron

Once a valid account number is entered, "001" in this case, the appropriate account name ("Aaron") is displayed.

Here is a sample program run with an invalid account number entered:

```
Enter account number to search for: 003
Exception in thread "main" java.lang.NullPointerException
     at NullPointerDemo.main(NullPointerDemo.java:15)
```

As you can see, in this case, having entered an invalid account number ("003") a `NullPointerException` is thrown. Here is the offending instruction:

```
System.out.println(myBank.getItem(account).getAccountName());
```

The problem here is that, as we have seen, the `getItem` method (highlighted in bold) does not return a valid `BankAccount` object but, instead, a **`null`** value when the given account number is not present in the list. Consequently, the attempt to get the account number via the `getAccountNumber` fails with a `NullPointerException` then thrown.

Of course, we can avoid this error by using an **`if`** statement to check for a **`null`** value and only call the `getAccoutNumber` method if we do not have a **`null`** value:

```
if (myBank.getItem(account) != null) // add a check for non-null value
{
       System.out.println(myBank.getItem(account).getAccountName()); // safe to call method
}
```

While this would work, unfortunately very often these **`if`** checks are not included and potential `NullPointerExceptions` can remain. To deal with this problem, Java has recently introduced the `Optional` class to better deal with such `NullPointerExceptions`.

14.7 The *Optional* Class

The `Optional` class is essentially a wrapper class that can contain the value of a given type, or it can contain **`null`**. The best way to see how this `Optional` class works is to use it in an example, so let's re-write the `getItem` method in our `Bank` collection class to make use of this new class.

Firstly, the `Optional` class resides in the `util` folder, so we will need the following **`import`** statement at the top of our `Bank` class:

```
import java.util.Optional;

public class Bank
{
      // code for Bank class goes here
}
```

Now, here is the new `getItem` method; take a look at it and then we will discuss it:

```
//note the Optional return type
public Optional<BankAccount> getItem(String accountNumberIn)
{
        int index = search(accountNumberIn);
        if(index != -999) // check that account exists
        {
            return Optional.of(list.get(index)); // send valid Optional value
        }
        else
        {
            return Optional.empty(); // send empty(null) Optional value
        }
}
```

Firstly, you can see that the return type is no longer just `BankAccount`, but an `Optional` value that *could* contain a `BankAccount`—the generics mechanism is used to fix the contained type:

```
// Optional return type may contain a BankkAccount or null
public Optional<BankAccount> getItem(String accountNumberIn)
{
      // code for getItem here
}
```

We should point out here that changing the `getItem` method in this way would mean that we would also have to change the `deposit` and `withdraw` methods, both of which call `getItem`; we do this by using the `get` method of `Optional`, as explained below.

The `Optional` return type makes it clear to anyone using this class that the value being returned could contain a `BankAccount` *or* it could contain **`null`**, whereas a return type of `BankAccount` would not have made this clear.

Now, when we wish to return a `BankAccount` object we wrap it up inside an `Optional` value using the `Optional` class method `of` as follows:

```
if(index != -999) // check that account exists
{
      return Optional.of(list.get(index)); // send valid Optional value
}
```

To indicate no BankAccount object can be found we wrap up a **null** value inside an Optional value using the Optional class method empty as follows:

```
else
{
       return Optional.empty(); // send empty(null) Optional value
}
```

Programmers are no longer able to treat the returned value as a BankAccount object. It is now an object of type Optional<BankAccount>. So, for example, the following instruction from our NullPointerDemo program will no longer compile:

```
System.out.println(myBank.getItem(account).getAccountName()); //compiler error
```

Since getItem now returns an Optional<BankAccount> object we are no longer able to call a BankAccount method such as getAccountNumber, and attempting to do so raises a compiler error.

So how do we avoid this compiler error? One approach would be to check if a valid value has been returned. We can use the isPresent method of the Optional class, that returns **true** if a valid object is returned and **false** otherwise, on the object returned:

```
if (myBank.getItem(account).isPresent())
{
       // code to retrieve valid object here
}
```

Now can retrieve the BankAccount object before we call the getAccountName method. We can use the get method of the Optional class to do this:

```
if (myBank.getItem(account).isPresent())
{
       // notice use of 'get' method to retrieve the BankAccount object inside Optional
       System.out.println(myBank.getItem(account).get().getAccountName());
}
```

There is also another method we can use called ifPresent. This receives an object of the functional interface Consumer, which you came across in Chap. 13. It is one of the "out-of-the-box" interfaces and was described in Table 13.1. Its abstract method, accept, receives a single parameter (of whatever type is chosen for the generic interface) and its return type is void. We can therefore call ifPresent with a lambda expression. Below, we have re-written the NullPointerDemo program using this syntax; take a look at it and then we will discuss it:

NullPointerDemoWithOptional

```
public class NullPointerDemoWithOptional
{
    public static void main(String[] args)
    {
       // create Bank collection
       Bank myBank = new Bank();  // version with new getItem method
       // add two BankAccount objects
       myBank.addAccount("001", "Aaron");
       myBank.addAccount("002", "Quentin");

       // request user for account number
       System.out.print("Enter account number to search for: ");
       String account = EasyScanner.nextString(); // we are using our EasyScanner class here

       // retrieve account name accociated with this account number using lambda, and display result
       myBank.getItem(account).ifPresent(value -> System.out.println(value.getAccountName()));
    }
}
```

You can see that we have retrieved a `BankAccount` using the new version of `getItem`, and it will therefore be of type `Optional<BankAccount>`. We then call the `ifPresent` method of `Optional` with the correct lambda expression for the abstract method of the `Consumer` interface. The input to this method (which we have called `value`) is the item held in the `Optional` object, which will be either a `BankAccount` or a **`null`** value. The `ifPresent` method executes the lambda expression only if the item held is a valid object—if it is not, it does nothing.

```
myBank.getItem(account).ifPresent(value -> System.out.println(value.getAccountName()));
```

Thus, the program displays the account name if a valid object was retrieved, otherwise nothing is displayed.

You can see how the `Optional` type alerts programmers that the **`null`** value might be returned and ensures they unpack this value before proceeding, thus avoiding `NullPointerExceptions`. It also provides a variety of methods that, potentially, avoid the need for **`null`** value **`if`** checks in your code. There are a variety of other methods contained in the `Optional` class, which you can look up on the Oracle™ site.

14.8 Exceptions in GUI Applications

In Sect. 14.4 we showed you how the `parseInt` method could potentially result in a `NumberFormatException` being thrown. If this were not handled at some point, the exception object would escape out of your program and cause the program to terminate.

However, this isn't the first time that you used the `parseInt` method. You often had to use it when implementing your JavaFX GUI applications. In such applications all user input would normally be retrieved as strings, and then converted to numbers by calling methods such as `parseInt` method and `parseDouble`.

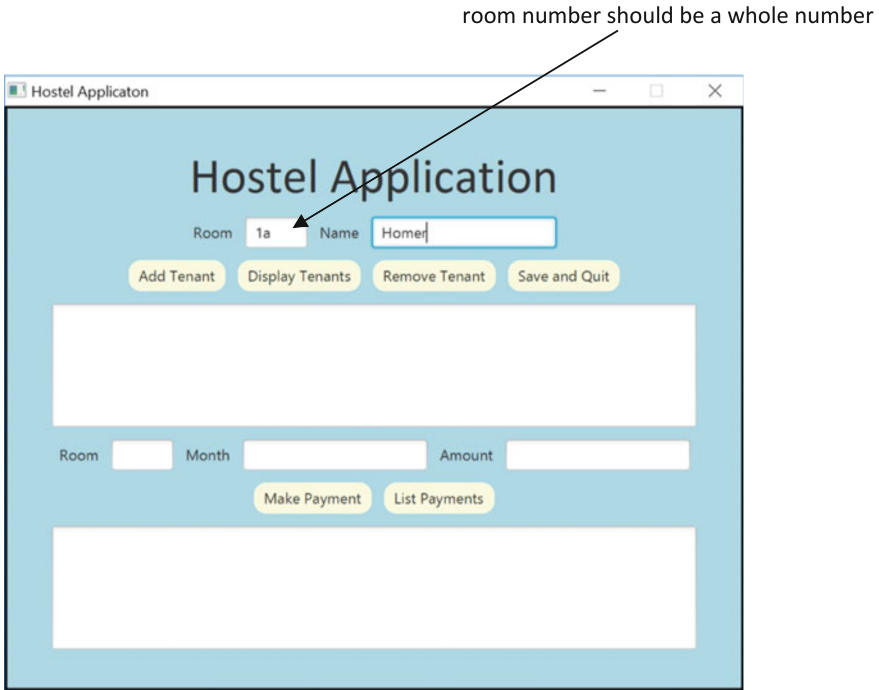


Fig. 14.4 A sample screen shot from the 'Hostel' case study illustrating an invalid room number having been entered

At the time, you never considered handling these exceptions, and your applications never seemed to terminate as a result of invalid data entry. For example, do you remember the `Hostel` case study of Chaps. 11 and 12? Figure 14.4 illustrates a sample screen shot when a user enters an invalid room number.

When such an event occurred within your JavaFX application, the application seemed to continue operating regardless. After our discussion on exceptions this might seem surprising as the text entered is being processed by a `parseInt` method. To remind you, here is a fragment from the `addHandler` method:

```
private void addHandler()
{
         String roomEntered =  roomField1.getText(); // code to read room number
         String nameEntered =  nameField.getText();

       // previous additional code here

       // code to check room number, parseInt could cause an exception!
       if(Integer.parseInt(roomEntered)< 1 || Integer.parseInt(roomEntered)>noOfRooms)
       {
            displayArea1.setText ("There are only "  + noOfRooms  + " rooms");
       }

       // code to addTenant here
}
```

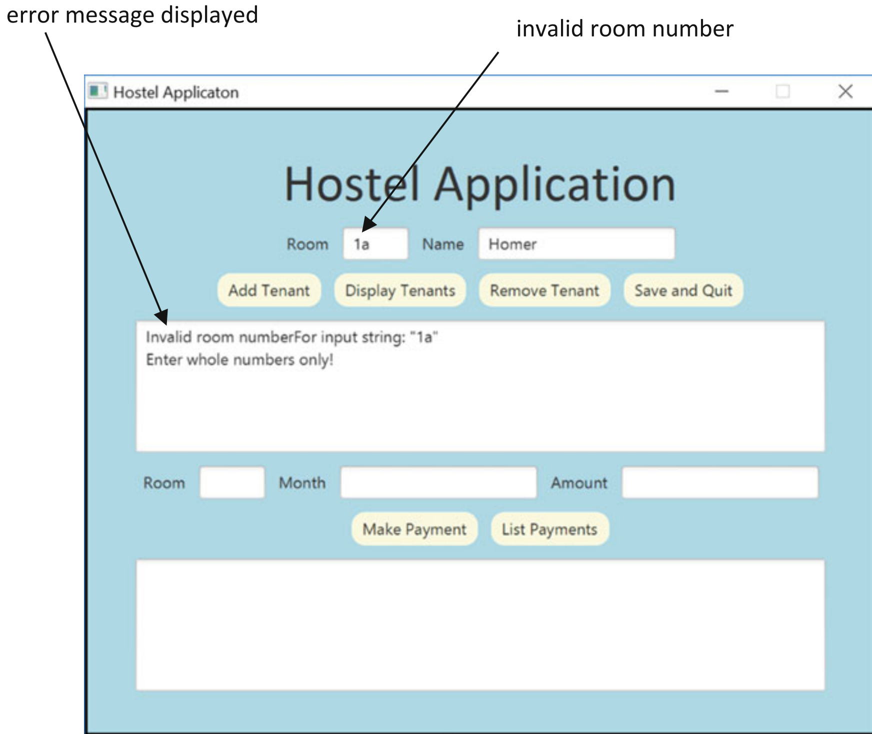


Fig. 14.5 Exceptions can still be dealt with in JavaFX applications

In fact, when an invalid number is entered as illustrated in Fig. 14.4, a `NumberFormatException` occurs—but:

- you will not see details of the exception in your graphics screen since they will always be displayed in the output window of your IDE, or, if you are running your program from the command line, on the black console window (which may be hidden during the running of your application);
- exceptions do not terminate JavaFX applications; however, they may make them behave unpredictably.

So, if you look at the console screen or output window, you will see a list of exceptions that have been thrown during the running of your JavaFX applications—you may be surprised to see how many are actually thrown when you thought your program was operating correctly.

Often, graphical programs will continue to operate normally in the face of exceptions. To ensure this is the case you should still add exception handling code into your JavaFX applications. For example, we could amend the event-handler above as follows:

```
private void addHandler()
{
        try // place previous code in try block
        {
               // previous add handler code here
        }
        catch (NumberFormatException e) // catch exception and diplay error message on GUI
        {
               displayArea1.setText("Invalid room number" + e.getMessage()
                                                       + "\nEnter whole numbers only!");

        }
}
```

Now if we run the application again, with the same input as depicted in Fig. 14.4, we get the response given in Fig. 14.5.

14.9 Using Exceptions in Your Own Classes

So far we have mainly been dealing with how to handle predefined exceptions (as summarised in Fig. 14.1) that are automatically thrown by your Java programs, such a `NullPointerException` being thrown when calling a method of a **`null`** value.

Sometimes, however we wish to generate an error under circumstances when Java would not ordinarily raise an error. For example, think back to the `PaymentList` collection class from the case study in Chap. 11. Here is an outline of that class:

```
import java.util.ArrayList;

public class PaymentList
{
        private ArrayList<Payment> pList;
        public final int MAX;

        /** Constructor initialises the empty payment list and sets the maximum list size
         * @param   maxIn The maximum number of payments in the list
         */

       public PaymentList(int maxIn)
       {
         pList = new ArrayList<>();
         MAX = maxIn;
       }

       // remaining methods go here
```

Here we have an `ArrayList` to hold the payments and a `MAX` value to record the maximum number of payments. There are no Java exceptions we need to be concerned about here. But take a look at the code for the constructor. Can you see a potential problem here?

We are setting the value of `MAX` and this value is sent as an integer. We would want this number to be a positive number, but integers in Java can be positive or negative or zero. If a negative or zero value is sent, Java will still allow us to set

MAX to this value but doing so would create knock on logical problems in our program code.

Ideally, we should avoid setting MAX to a value that is less than 1, and would instead wish to report an error. We could try and use an **if else** statement to check the parameter, maxIn, and then try and report an error if need be:

```
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       if (maxIn < 1) // check if parameter is not positive
       {
              // report error
       }
       else
       {
              MAX = maxIn; // ok to set MAX
       }
}
```

How would we report back this error? One way, which we have used before, would be to return a **boolean** value of **false** to indicate failure. A general problem with reporting errors using **boolean** values is that these values can be ignored. In this case we have another problem—constructors can have no return value! The only way to report back errors from constructors is to throw exceptions.

14.9.1 Throwing Exceptions

We have seen circumstances where our programs automatically throw exceptions, but we have not seen examples of where we throw our own exceptions. Let's take a look at the syntax to do this.

The first thing we need to decide is *which* exception to throw. We could make use of one of the pre-existing Java exception class. Since the error we wish to report is a logical error we could try throwing a RuntimeException. In order to throw an exception object you must:

- write an instruction explicitly to throw the exception using a **throw** command;
- combine this with the **new** command to generate an object of the appropriate exception type by calling the given exception class's constructor.

Below is a modified PaymentList constructor that throws a RuntimeException:

```
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       if (maxIn < 1)
       {
              throw new RuntimeException( ); // force a RuntimeException to be thrown
       }
       else
       {
              MAX = maxIn;
       }
}
```

Notice this version of the RuntimeException constructor requires no parameters:

```
throw new RuntimeException( ); //  no parameters required here
```

Every Java exception class is provided with an alternative version of the constructor that can receive a String parameter, often used to bundle information regarding the error within the exception object.

Let's use this alternative version of the RuntimeException constructor to provide some useful error information:

```
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       if (maxIn < 1)
       {
              throw new RuntimeException("invalid list size set " + maxIn ); // send error info
       }
       else
       {
              MAX = maxIn;
       }
}
```

Now let's look at a program that makes use of this modified PaymenList constructor, catches the RuntimeException if it is thrown and displays this exception object (containing our error message).

RuntimeExceptionDemo

```
public class RuntimeExceptionDemo
{
    public static void main(String[] args)
    {
        try
        {
            System.out.print("Enter size of list: ");
            int size = EasyScanner.nextInt();  // we are using our EasyScanner here
            PaymentList p = new PaymentList(size); // can now throw RuntimeException
        }
        catch(RuntimeException e) // catch exception from PaymentList constructor
        {
            System.out.println(e); // display exception object
        }
        System.out.println("END OF DEMO");

    }
}
```

First, a normal test run:

Enter size of list: 5
END OF DEMO

Now a test run with an invalid list size:

Enter size of list: 0
java.lang.RuntimeException: invalid list size set 0
END OF DEMO

Here you can see an illegal list size of zero triggers the `RuntimeException` and displaying this exception object reveals our error information.

In the example above we had an error that made sense to our program, avoiding using a non-positive value when setting the maximum size of our list, but we have co-opted an existing exception name to report this error (`RuntimeException`).

Using a pre-existing exception type name (such as `RuntimeException`) to report your own program-specific errors has a few problems. Firstly, a clause to catch this exception will also catch any other exception of the given type. However, you might want very different error handling code for your program specific errors. Secondly, the name of the exception type is not a good description of your specific error. It would be better if we could invent a new name (or names) for your program-specific errors. For example, if you were developing a game where pieces moved on a board you might want to have an exception called `InvalidMoveException` rather than use a generic name such as `RuntimeException`.

Luckily, it is very simple to create your own exception classes with names of your choice.

14.9.2 Creating Your Own Exception Classes

You can create your own exception class by inheriting from any predefined exception class. Generally speaking, if you wish your exception class to be unchecked then it should inherit from `RuntimeException` (or one of its sub-classes), whereas if you wish your exception to be checked you can inherit from the general `Exception` class.

In the case of the `PaymentList` class we wish to make the exception thrown by the constructor unchecked, as it is a logical error not an input/output error. So we will define our exception class by inheriting from the `RuntimeException` class. We will call this new exception class `HostelException` so it can be used throughout our Hostel application if need be. Look at the code first and then we will discuss it.

HostelException

```
public class HostelException extends RuntimeException
{
	public HostelException () // constructor without parameter
	{
		super("error in Hostel application");
	}

	public HostelException (String message)// constructor with parameter
	{
		super (message);
	}
}
```

As well as inheriting from some exception class, user-defined exception classes should have two constructors defined within them. One should take no parameters and simply call the constructor of the superclass with a default message of your choosing:

```
public HostelException () // constructor without parameter
{
        super("error in Hostel application"); // default error info
}
```

The other constructor should allow a user-defined message to be supplied with the exception object:

```
public HostelException (String message)// constructor with parameter
{
        super (message); // user defined message can be provided
}
```

The PaymentList constructor can now be modified to make use of this HostelException class as follows:

```
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       if (maxIn < 1)
       {
              throw new HostelException("invalid list size set " + maxIn ); // user-defined error
       }
       else
       {
              MAX = maxIn;
       }
}
```

Finally, we can amend the tester so that the new HostelException is caught:

HostelExceptionDemo

```
public class HostelExceptionDemo
{
    public static void main(String[] args)
    {
        try
        {
            System.out.print("Enter size of list: ");
            int size = EasyScanner.nextInt();
            PaymentList p = new PaymentList(size); // can now throw HostelException
        }
        catch(HostelException e) // catch exception from PaymentList constructor
        {
            System.out.println(e); // display exception object
        }
        catch (Exception e) // note general catch clause added
        {
            System.out.println("Some unforseen error");
            e.printStackTrace();
        }
        System.out.println("END OF DEMO");
    }
}
```

Notice how we added a general **catch** clause to catch any exceptions that we might not yet have considered. In this **catch** clause we have printed the stack trace in this error-handler to determine the exact cause of this unexpected error:

```
catch (Exception e) // catches all uncaught errors
{
       System.out.println("Some unforseen error");
       e.printStackTrace();
}
```

During testing this is always a good strategy. However, you need to ensure that the general **catch** clause is the *last* **catch** clause you specify. For example, something like the following will not compile:

```
try
{
       // some code here
}
catch (Exception e) // catches all uncaught errors
{
       // some code here
}
catch (HostelException e) // will not compile!
{
       // some code here
}
```

The above will not compile as the first general **catch** clause (**catch** Exception) will catch *all* exception types (including HostelException). So, any **catch** clauses below (such as catch HostelException) will never be reached. To write unreachable code in Java causes a compiler error.

14.10 Documenting Exceptions

We finish off this chapter by looking at how to document methods that may throw exceptions, using the Javadoc style of comments discussed in Chap. 11. The @throws tag should be used to document the name of any exceptions that may be thrown by a method. Here for example, is the PaymentList constructor, documented with Javadoc comments:

```
/** Constructor initialises the empty payment list and sets the maximum list size
 * @param   maxIn             The maximum number of payments in the list
 * @throws  HostelException   If the list is sized with a non-positive value
 */
public PaymentList(int maxIn)
{
       pList = new ArrayList<>();
       if (maxIn < 1)
       {
              throw new HostelException("invalid list size set " + maxIn);
       }
       MAX = maxIn;
}
```

Generally speaking, when documenting methods in this way, it is good practice to document *all* exceptions that a method may throw. Multiple `@throws` tags can be used to list multiple exceptions.

14.11 Self-test Questions

1. What is an *exception*?
2. Distinguish between *checked* and *unchecked* exceptions and then identify which of the following exceptions are checked, and which are unchecked:
   - `FileNotFoundException`;
   - `NegativeArraySizeException`;
   - `NullPointerException`;
   - `NumberFormatException`;
   - `IOException`;
   - `Exception`;
   - `ArrayIndexOutOfBoundsException`;
   - `RuntimeException`.
3. Explain the following terms:
   (a) *claiming* an exception;
   (b) *throwing* an exception;
   (c) *catching* an exception.
4. Look at the program below and then answer the questions that follow:

```
public class ExceptionsQ4
{
   public static void main(String[] args)
   {
       int[] someArray = {12,9,3,11};
       int position = getPosition();
       display (someArray, position);
       System.out.println("End of program" );
    }

    static int getPosition()
    {
       System.out.println("Enter array position to display");
       String positionEntered = EasyScanner.nextString(); // requires EasyScanner class
       return Integer.parseInt(positionEntered);
    }

    static void display (int[] arrayIn, int posIn)
    {
         System.out.println("Item at this position is: " + arrayIn[posIn]);
    }
}
```

(a) Will this result in any compiler errors?

(b) Which methods could throw exceptions?

(c) Identify the names of the exceptions that could be thrown and the circumstances under which they could be thrown.

5. What is the purpose of a **finally** clause?

6. When would you use the *try-with-resources* construct?

7. Consider once again the PaymentList class of Chap. 11. In particular here is a reminder of its getItem method that returns a particular payment given a position in the list:

```
/** Reads the payment at the given position in the list
 *  @param      positionIn The logical position of the payment in the list
 *  @return     Returns the payment at the given logical position in the list
 *              or null if no payment at that logical position
 */
public Payment getPayment(int positionIn)
{
       //check for valid logical position
       if (positionIn <1 || positionIn > getTotal())
       {
              // no object found at given position
              return null;
       }
       else
       {
              // take one off logical position to get ArrayList position
              return pList.get(positionIn - 1);
       }
}
```

(a) Explain the purpose of the Optional class.

(b) Re-write the getPayment method using an Optional class.

8. When would it be appropriate to define your own exception class?

14.12 Programming Exercises

1. Implement the program given in self-test question 4 above. Now:

(a) Re-write main so that it catches any exceptions it may now throw by displaying a message on the screen indicating the exception thrown.

(b) At the moment, the "End of program" message may not always be executed. Add an appropriate **finally** clause so that this message is always executed at the end of the program.

(c) Add an additional **catch** clause in main to catch any unaccounted-for exceptions (within this **catch** clause print out the stack trace of the exception).

(d) Create your own exception class InvalidPositionException (make this an unchecked exception).

(e) Re-write the display method so that it throws the `InvalidPositionException`.

(f) Re-write `main` to take account of this amended display method.

(g) Document these exceptions using appropriate `Javadoc` comments.

2. The `Scanner` class has methods `nextInt` and `nextDouble` for reading an **`int`** and a **`double`** value from the keyboard respectively. Both of these methods throw an exception if an appropriate numerical value is not entered.

(a) Write a tester program to find out the name of this exception.

(b) Develop a new version of the `EasyScanner` class, say `EasyScannerPlus`, so that instead of throwing exceptions the methods `nextInt` and `nextDouble` repeatedly display an error message and allow for data re-entry.

(c) Write a tester program to test out the methods of your `EasyScannerPlus` class.

3. Look back at the time table application that you developed in programming Exercise 8 of Chap. 8. Here is a reminder of the original design for the `TimeTable` class:

TimeTable
-times: Booking[][]
+TimeTable(int, int) ***+makeBooking(int, int, Booking) : boolean*** ***+cancelBooking(int, int) : boolean*** ***+isFree(int, int) : boolean*** ***+getBooking(int, int) : Booking*** ***+numberOfDays() : int*** ***+numberOfPeriods() : int***

As you can see several methods return **`boolean`** values. Some of these **`boolean`** values were sent to indicate errors. Now modify this class to make use of exceptions as follows:

(a) Develop a `TimeTableException` class (make this an unchecked exception).

(b) Modify the `TimeTable` class so that the constructor, and the methods `makeBooking`, and `cancelBooking` all throw a `TimeTableException` if an error occurs (this would mean that the methods `makeBooking` and `cancelBooking` no longer need to return **`boolean`** values).

(c) The `getBooking` method currently returns **`null`** if either the given day or period number passed as parameters are invalid. Re-write this method to return an `Optional` value instead.

(d) Modify the tester that you developed for the time table application to take account of the exceptions and `Optional` value you incorporated in parts (b) and (c) above.

4. Look back at the *Hostel* case study of Chaps. 11 and 12 and make use of exceptions and `Optional` types where appropriate. Amend the `Javadoc` documentation for this application to include information on any exceptions you may have included.

Check for updates

15 The Java Collections Framework

Outcomes:

By the end of this chapter you should be able to:

- *use the* ArrayList *class to store a* ***list*** *of objects*;
- *use the* HashSet *class to store a* ***set*** *of objects*;
- *use the* HashMap *class to store objects in a* ***map***;
- *use the enhanced* **for** *loop and a* **forEach** *loop to scan through a collection*;
- *use an* Iterator *object to scan through a collection*;
- *create objects of your own classes, and use them in conjunction with Java's collection classes*;
- *sort elements in a collection using the* Comparable<T> *and* Comparator<T> *interfaces.*

15.1 Introduction

An array is a very useful type in Java but it has its restrictions:

- once an array is created it must be sized, and this size is fixed;
- it contains no useful pre-defined methods.

Think back to the SomeUsefulArrayMethods program of Chap. 6. As this program used an array to hold a collection of elements we had to put an upper limit to the size of this collection. Sometimes, however, an upper limit is not known. Just creating a very big array is very wasteful of memory—and what happens if even this very big array proves to be too small? Another problem, as seen in our SomeUsefulArrayMethods program, was that to carry out any interesting processing (like searching the array) required us to write complex algorithms.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_15

One solution to this problem might be to develop our own collection class that hid the array from the user and contained the code to create a reasonable sized array and, when the array is full, copy this array into a slightly bigger array and use this new array and continue doing this every time the array gets full. This collection class could also include a large group of useful methods to process the array (such as searching and deleting). In order to make this collection class as useful as possible we could incorporate generics (that we met in Chap. 13) to allow this collection class to hold a collection of any type. This collection class could then be used in place of an array to hold and process a collection of objects.

Luckily we do not need to go to such lengths; the Java developers have already done this for us. They have developed a group of generic collection classes that grow as more elements are added to them, and these classes provide lots of useful methods. This group of collection classes are referred to as the **Java Collections Framework** (**JCF**). These collection classes are organized around several collection interfaces that define different types of collections that we might be interested in using. Three important interfaces from this group are:

- `List`;
- `Set`;
- `Map`.

The `List` and `Set` interfaces are themselves specialised types of the super interface named `Collection`, and so share many common methods, whereas the `Map` interface is distinct from `Collection` group of interfaces and so has several methods that are unique to it.

As well as providing a wide group of collection interfaces, the Java Collections Framework also contains many classes that implement these interfaces. In this chapter we will focus on the three key interfaces named above (`List`, `Set` and `Map`) and some of the classes provided in the JCF which implement these interfaces. To find out about all the interfaces and classes in the JCF you can refer to the Oracle™ site.

15.2 The *List* Interface and the *ArrayList* Class

The `List` interface specifies the methods required to process an *ordered list* of objects. Such a list may contain duplicates. Examples of a list of objects include jobs waiting for a printer, emergency calls waiting for an ambulance and the names of players that have won the Wimbledon tennis tournament over the last 10 years. In each case ordering is important, and repetition may also be required. We often think of such a collection as a *sequence* of objects.

There are two implementations provided for the List interface in the JCF. They are ArrayList, and LinkedList. We have already introduced you to the ArrayList in the Bank application in Chap. 8, that stored a collection of BankAccount objects, so let's take a closer look at the ArrayList class now.

15.2.1 Creating an *ArrayList* Collection Object

All classes in the JCF are in the java.util package, so to use the ArrayList class we require the following **import** statement:

```
import java.util.ArrayList;
```

Like all the classes in JCF the ArrayList class is a *generic* collection class. This means it can be used to store objects of *any* type.[1] Let's use an ArrayList to store a queue of jobs waiting for a printer, and let us represent these jobs by a series of Job ID Strings. The ArrayList constructor creates an empty list:

```
// creates an ArrayList object -'printQ'
ArrayList<String> printQ = new ArrayList<> ();
```

Notice the use of generics to set the type of objects stored in our list. In the case of the printQ object, we want each element within this collection to be of type String. As mentioned in Chap. 7, type inference means that we do not need to include the String type in the angle brackets in the call to the constructor on the right, as the type is inferred to be String in this case.

Of course, the generics mechanism can be used to fix *any* object type for the elements within a collection. In Chap. 8, for example, we used an ArrayList to store a collection of BankAccount objects in our Bank class. As another example, if you wished create a list of Oblong objects you could do so as follows:

```
// this will make 'someOblongs' a list of Oblong objects
ArrayList<Oblong> someOblongs = new ArrayList<> ();
```

You should be aware that the type of any object created using this generics mechanism is now the class name *plus* the contained type brackets. So, for example, if we were to write a method that received a list of strings, we would specify it as follows:

[1]As we mentioned in Chap. 7, generic collections cannot store primitive types like **int** and **double**. If primitive types are required then objects of the appropriate wrapper class (Integer, Double and so on) must be used. However, as discussed in Chap. 9, *autoboxing* and *unboxing* automate the process of moving from a primitive type to its associated wrapper.

```
// this method receives an ArrayList<String> object
public void someMethod (ArrayList<String> printQIn)
{
    // some code here
}
```

15.2.2 The Interface Type Versus the Implementation Type

In order to create the object `printQ`, we have used the `ArrayList` class to implement the `List` interface. What if, at some point, we decide to change to the `LinkedList` implementation? Or some other implementation that might be available in the future? If we did this, then all references to the type of this object (such as in the method header of the previous section) would have to be modified to give the new class name.

There is an alternative approach. It is actually considered better programming practice to declare collection objects to be the type of the interface (`List` in this case) rather than the type of the class that implements this collection. So this would be a better way to create our `printQ` object:

```
// the type is given as 'List' not 'ArrayList'
List<String> printQ = new ArrayList<> ();
```

Notice that the interface type still needs to be marked as being a list of `String` objects using the generics mechanism. A method that receives a `printQ` object would now be declared as follows:

```
// this method receives a List<String> object
public void someMethod (List<String> printQIn)
{
    // some code here
}
```

The advantage of this approach is that we can change our choice of implementation in the future (maybe by using `LinkedList` or some other class that might be available that implements the `List` interface), without having to change the type of the object (which will always remain as `List`). This is the approach that we will take.

Remember, all classes in the JCF reside in the `util` package, so to use the `List` interface in your code you now need to add an additional **`import`** statement as follows:

```
import java.util.ArrayList;
import java.util.List;
```

Now, let us look at some `List` methods.

15.2.3 *List* Methods

The `List` interface defines two `add` methods for inserting into a list, one that inserts the item at the end of the list and one that inserts the item at a specified position in the list. Like arrays, `ArrayList` positions begin at zero. We wish to use the `add` method that adds items to the end of the list as we are modelling a queue here. This `add` method requires one parameter, the object to be added into the list:

```
printQ.add("myLetter.doc");
printQ.add("myFoto.jpg");
printQ.add("results.xls");
printQ.add("chapter.doc");
```

Notice that, since we have marked this list as containing `String` objects only, an attempt to add an object of any other type will result in a compiler error:

```
// will not compile as 'printQ' can hold Strings only!
printQ.add(new Oblong(10, 20));
```

All the Java collection types have a `toString` method defined, so we can display the entire list to the screen:

```
System.out.println(printQ); // implicitly calling the toString method
```

When the list is displayed, the values in the list are separated by commas and enclosed in square brackets. So this `println` instruction would display the following list:

[myLetter.doc, myFoto.jpg, results.xls, chapter.doc]

As you can see, the items in the list are kept in the order in which they were inserted using the `add` method above.

As we said earlier, the `add` method is overloaded to allow an item to be inserted into the list at a particular position. When the item is inserted into that position, the item previously at that particular position and all items behind it shuffle along by one place (that is they have their indices incremented by one). This `add` method requires two parameters, the position into which the object should be inserted, and the object itself. For example, let's insert a high priority job at the start of the queue:

```
printQ.add(0, "importantMemo.doc"); // inserts into front of the queue
```

Notice that the index is provided first, then the object. The index must be a valid index within the current list or it may be the index of the back of the queue. An invalid index throws an unchecked `IndexOutOfBoundsException`.

Displaying this list confirms that the given job ("`importantMemo.doc`") has been added to the front of the queue, and all other items shuffled by one place:

[importantMemo.doc, myLetter.doc, myFoto.jpg, results.xls, chapter.doc]

If we wish to overwrite an item in the list, rather than insert a new item into the list, we can use the `set` method. The `set` method requires two parameters, the index of the item being overwritten and the new object to be inserted at that position. Let us change the name of the last job from "`chapter.doc`" to "`newChapter.doc`". This is the fifth item in the list so its index is 4:

```
printQ.set(4, "newChapter.doc");
```

If the index used in the `set` method is invalid an `IndexOutOfBoundsException` is thrown once again. Displaying the new list now gives us the following:

[importantMemo.doc, myLetter.doc, myFoto.jpg, results.xls, newChapter.doc]

`List` provides a `size` method to return the number of items in the list, so we could have renamed the last job in the queue in the following way also:

```
printQ.set(printQ.size()-1, "newChapter.doc"); // last position is size-1
```

The `indexOf` method returns the index of the first occurrence of a given object within the list. It returns −1 if the object is not in the list. For example, the following checks the index position of the job "`myFoto.jpg`" in the list:

```
int index = printQ.indexOf("myFoto.jpg"); // check index of job
if (index != -1) // check object is in list
{
   System.out.println("myFoto.jpg is at index position: " + index);
}
else // when job is not in list
{
   System.out.println("myFoto.jpg not in list");
}
```

This would display the following from our list:

myFoto.jpg is at index position: 2

Items can be removed either by specifying an index or an object. When an item is removed, items behind this item shuffle to the left (i.e. they have their indices decremented by one). As an example, let us remove the "myFoto.jpg" job. If we used its index, the following is required:

```
printQ.remove(2);
```

Once again, an IndexOutOfBoundsException would be thrown if this was not a valid index. This method returns the object that has been removed, which could be checked if necessary. Displaying the list would confirm the item has indeed been removed:

[importantMemo.doc, myLetter.doc, results.xls, newChapter.doc]

Alternatively, we could have removed the item by referring to it directly rather than by its index[2]:

```
printQ.remove("myFoto.jpg");
```

This method returns a **boolean** value to confirm that the item was in the list initially, which again could be checked if necessary.

Behind the scenes you can guess how this remove method works. It looks in the list for the given String object ("myFoto.jpg"). It uses the equals method of the String class to identify a match. Once it finds such a match it shuffles items along so there are no gaps in the list.

Of course, as we have already said, these collection classes can be used to store objects of *any* type—not just Strings. For methods like remove to work properly, the contained object must have a properly defined equals method. We will return to this later in the chapter, when we look at how to use objects of our own classes in conjunction with the classes in the JCF.

The get method allows a particular item to be retrieved from the list via its index position. The following displays the job at the head of the queue:

```
// the first item is at position 0
System.out.println("First job is " + printQ.get(0));
```

[2]If there were more than one occurrence of the object, the first occurrence would be deleted.

This would display the following:

First job is importantMemo.doc

The `contains` method can be used to check whether or not a particular item is present in the list:

```
if (printQ.contains("poem.doc")) // check if value is in list
{
    System.out.println("poem.doc is in the list");
}
else
{
    System.out.println("poem.doc is not in the list");
}
```

Finally, the `isEmpty` method reports on whether or not the list contains any items:

```
if (printQ.isEmpty()) // returns true when list is empty
{
    System.out.println("Print queue is empty");
}
else
{
    System.out.println("Print queue is not empty");
}
```

15.3 The Enhanced *for* Loop and Java Collections

In Chap. 6 we showed you how the enhanced **for** loop can be used to iterate through an entire array. The use of this loop is not restricted to arrays, it can also be used with the `List` (and `Set`) implementations provided in the JCF. For example, here an enhanced **for** loop is used to iterate through the `printQ` list to find and display those jobs that end with a ".doc" extension:

```
for (String item: printQ) // iterate through all items in the 'printQ'list
{
   if(item.endsWith(".doc")) // check extension of the job ID
   {
       System.out.println(item); // display this item
   }
}
```

Notice that the type of each item in the `printQ` list is `String`. Within the loop we use the `String` method `endsWith` to check if the given job ID ends with `String` ".doc". Assuming we had the following `printQ`:

[importantMemo.doc, myLetter.doc, results.xls, newChapter.doc]

the enhanced **for** loop above would produce the following output:

importantMemo.doc
myLetter.doc
newChapter.doc

If we do not wish to iterate through the *entire* list, or if we wish to *modify* the items within a list as we iterate through them, then (as we have said before) the enhanced **for** loop should not be used.

For example, if we wished to display the items in the printQ that are behind the head of the queue, the enhanced **for** loop is not appropriate as we are not processing the *entire* printQ. Instead the following standard **for** loop could be used:

```
// remember second item in list is at index 1!
for (int pos = 1; pos < printQ.size(); pos++)
{
     System.out.println(printQ.get(pos)); // retrieve item in printQ
}
```

Notice how the size method is used to determine the last index in the loop header. Within the loop, the get method is used to look up an item at the given index.

Again, if we assume we have the following printQ:

[importantMemo.doc, myLetter.doc, results.xls, newChapter.doc]

the **for** loop above would produce the following output:

myLetter.doc
results.xls
newChapter.doc

15.4 The *forEach* Loop

There is a very similar loop to the enhanced **for** loop known as the **forEach** loop. The **forEach** loop makes use of the forEach method, which its to be found in classes implementing the Collection interface (such as lists and sets). The forEach method requires an implementation of a Consumer interface, which can be provided via a lambda expression. Table 15.1 provides a reminder of the Consumer interface that we discussed in Chap. 13.

Table 15.1 Reminder of the Consumer interface

Functional interface	Abstract method name	Parameter types	Return type
Consumer<T>	accept	T	void

This implementation is then applied to each element in the underlying collection. For example, lets revisit the code in Sect. 15.3 for searching through our printQ list to display document names that end with ".doc". This time we will use a lambda expression and a **forEach** loop:

```
// using a forEach loop to iterate through a list
printQ.forEach(item ->
        {
            if(item.endsWith(".doc")) // check extension of the job ID
            {
                System.out.println(item); // display this item
            }
        });
```

As you can see, we use a **forEach** loop by calling the forEach method on the list object printQ. The parameter to the lambda expression is a name we give to one value from this collection, item in this case, and to the right of the lambda arrow we define how we process this value (using the same code we used in Sect. 15.3). The **forEach** loop will retrieve all the items within the collection to be processed in this way. You should note that, as with the enhanced **for** loop, the **forEach** loop should not be used to modify the underlying collection.

We will see further examples of the use of this **forEach** loop in this chapter. However, the **forEach** loop is primarily designed to be used with **streams**, a mechanism for processing Java collections that we will cover in Chap. 22.

15.5 The *Set* Interface and the *HashSet* Class

Like List, the Set interface is a specialised version of the general Collection interface. The Set interface defines the methods required to process a collection of objects in which there is no repetition, and ordering is unimportant. Let's consider the following collections and consider if they are suitable for a set:

- a queue of people waiting to see a doctor;
- a list of number-one records for each of the 52 weeks of a particular year;
- car registration numbers allocated parking permits.

The queue of people waiting to see a doctor cannot be considered to be a set as ordering is important here. Order may also be important when recording the list of number-one records in a year. It would also be necessary to allow for repetition—as a record may be number-one for more than one week. So a set is not a good choice

for this collection. The collection of car registration numbers can be considered a set, however, as there will be no duplicates and ordering is unimportant.

Java provides three implementations of this `Set` interface: `HashSet`, `TreeSet` and `LinkedHashSet`. Here we will look at the `HashSet` class. The constructor creates an empty set. We will use the set to store a collection of vehicle registration numbers (as `Strings`):

```
// creates an empty set of String objects
Set<String> regNums = new HashSet<>();
```

Again, notice that we have used the generics mechanism to indicate that this is a set of `String` objects, and we have given the type of this object as the interface `Set<String>`. Now let us look at some `Set` methods.

15.5.1 *Set* Methods

Once a set is created the methods specified in the `Set` interface can be used. The `add` method allows us to insert objects into the set, so let us add a few registration numbers:

```
regNums.add("V53PLS");
regNums.add("X85ADZ");
regNums.add("L22SBG");
regNums.add("W79TRV");
```

We can display the entire set as follows:

```
System.out.println(regNums); // implicitly calling the toString method
```

The set is displayed in the same format as a list, in square brackets and separated by commas:

[W79TRV, X85ADZ, V53PLS, L22SBG]

Notice that, unlike lists, the order in which the items are displayed is not determined by the order in which the items were added. Instead, the set is displayed in the order in which the items are stored internally (and over which we have no control). This will not be a problem as ordering is unimportant in a set.

As with a list, the `size` method returns the number of items in the set:

```
System.out.println("Number of items in set: " + regNums.size() );
```

This would `print` the following onto the screen:

Number of items in set: 4

If we try to add an item that is already in the set, the set remains unchanged. Let us assume the four items above have been added into the set and we now try and add a registration number that is already in the set:

```
regNums.add("X85ADZ"); // this number is already in the set
System.out.println(regNums);
```

When this set is displayed, "X85ADZ" appears only once:

[W79TRV, X85ADZ, V53PLS, L22SBG]

The `add` method returns a **boolean** value to indicate whether or not the given item was successfully added. This value can be checked if required.

```
boolean ok = regNums.add("X85ADZ"); // store boolean return value
if (!ok) //check if add method returned a value of false
{
      System.out.println("item already in the set!");
}
```

The `remove` method deletes an item from the set if it is present. Again, assuming that the four items given above are in the set, we can delete one item as follows:

```
regNums.remove("X85ADZ");
```

If we now display the set, the given registration will have been removed:

[W79TRV, V53PLS, L22SBG]

As with the `add` method, the `remove` method returns a **boolean** value of **false** if the given item to remove was not actually in the set. The `Set` interface also includes `contains` and `isEmpty` methods that work in exactly the same way as their `List` counterparts.

15.5.2 Iterating Through the Elements of a Set

Both the enhanced **for** loop and the **forEach** loop can be used to iterate through all the elements of a set. Let us look at an example.

In the UK, the first letter of the registration number was at one time used to determine the time period when the vehicle came to market. A registration beginning with ‘S’, for example, denoted a vehicle that came to market between August 1998 and February 1999, while a registration beginning with ‘T’ denoted a vehicle that came to market between March 1999 and July 1999.

The following enhanced **for** loop will allow us to iterate through the collection of registration numbers and display all registrations after ‘T’.

```
for (String item: regNums) // iterate through all items in 'regNums'
{
       if (item.charAt(0) > 'T') // check first letter of registration

      {
           System.out.println(item); // display this registration
      }
}
```

Here is the equivalent **forEach** loop:

```
regNums.forEach ( item ->
{
      if (item.charAt(0) > 'T') // check first letter of registration

      {
           System.out.println(item); // display this registration
      }
});
```

Again, notice that the type of every element within our `regNums` set is `String`. Within the loops we use the `String` method `charAt` to check the first letter of registration. Assuming we have the following set of registration numbers:

[W79TRV, V53PLS, L22SBG]

the loops above would both produce the following result:

W79TRV
V53PLS

Let us consider a slightly different scenario now. Instead of simply displaying registration numbers after ‘T’, we now wish to modify the original `regNums` set so that registrations prior or equal to ‘T’ are removed. We could try using an enhanced **for** loop again:

```
// this will compile but is not safe!
for (String item: regNums) // iterate through all items in 'regNums'
{
     if (item.charAt(0) <= 'T') // check first letter of registration
     {
        regNums.remove(item); // remove this registration
     }
}
```

Here we are once again iterating over the elements of the `regNums` set. But this time, within the loop, we are attempting to *remove* the given element from the set. As we said, both the enhanced **`for`** loop and the **`forEach`** loop both should *not* be used to modify or remove elements from the original collection. To do so would not give a compiler error but may lead to run-time exceptions. If we cannot use these loops here, how else can we iterate over the elements in a set? Unlike an array, values in the set cannot be retrieved by an index value. Instead, to access the items in a set, the `iterator` method can be used to obtain an **iterator** object.

15.5.3 *Iterator* Objects

An `Iterator` object allows the items in a collection to be retrieved by providing three methods defined in the `Iterator` interface (see Table 15.2).

To obtain an `Iterator` object from the `regNums` set, the `iterator` method could be called as follows:

```
// the 'iterator' method retrieves an Iterator object from a set
Iterator<String> elements = regNums.iterator();
```

Here we have called the `iterator` method and stored the item returned by this method in a variable we have called elements. The generics mechanism has been used here to indicate that the `Iterator` object will be used to iterate over `String` objects only.

Table 15.2 Methods of the *Iterator* interface

Method	Description	Inputs	Outputs
`hasNext`	Returns **true** if there are more elements in the collection to retrieve and **false** otherwise	None	An item of type **boolean**
`next`	Retrieves one element from the collection	None	An item of the given element type
`remove`	Removes from the collection	None	None
`forEachRemaining`	Performs the given action for each remaining element until all elements have been processed or the action throws an exception	An implementation for the `accept` method of the `Consumer` `<T>` interface	None

Once an `Iterator` object has been created, a **`while`** loop can be used to iterate through the collection, with the `hasNext` method the test of the loop. The body of the loop can then retrieve items with the `next` method and, if required, delete items with the `remove` method. Let us see how this would work with the `regNums` set.

```
/* an Iterator object can be used with a 'while' loop if we wish to iterate over a set and modify
its contents */

// first create an Iterator object as discussed before
Iterator<String> elements = regNums.iterator();
// repeatedly retrieve items as long as there are items to be retrieved
while (elements.hasNext())
{
    String item = elements.next(); // retrieve next element from set
    if (item.charAt(0) <= 'T') // check first letter of registration
    {
        elements.remove(); // call Iterator method to remove registration
    }
}
```

Within the loop we call the `next` method of our `Iterator` object to retrieve the next item within the collection. Since we have specified the `Iterator` object to retrieve `String` objects, we know that this method will return a `String`. We have stored this `String` object in a variable we have called `item`:

```
// the String returned from the Iterator object is stored in a variable
String item = elements.next();
```

It is always a good idea to store the object returned by the `next` method in a variable, as the `next` method should be called only *once* within the loop. Storing the result in a variable allows us to refer to this object as many times as we like. In this case we refer to the object only once, in the test of the **`if`** statement:

```
if (item.charAt(0)<= 'T') // check the first character in this String
```

If the registration number needs to be removed from the set, we may do so now safely—by calling the `remove` method of our `Iterator` object:

```
elements.remove(); // call Iterator method to remove current item
```

The `IteratorDemo` class gathers this code into a complete program with a few example registration numbers.

IteratorDemo

```
import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

public class IteratorDemo
{
    public static void main(String[] args)
    {
        Set<String> regNums = new HashSet<>();
        regNums.add("V53PLS");
        regNums.add("X85ADZ");
        regNums.add("L22SBG");
        regNums.add("W79TRV");
        regNums.add("E16UEL");
        System.out.println("items before removing: " + regNums);
        // create an Iterator object
        Iterator<String> elements = regNums.iterator();
        // repeatedly retrieve items as long as there are items to be retrieved
        while (elements.hasNext())
        {
            String item = elements.next(); // retrieve next element from set
            if (item.charAt(0) <= 'T') // check first letter of registration
            {
                elements.remove(); // call Iterator method to remove registration
            }
        }
        System.out.println("items after removing: " + regNums);
    }
}
```

The program produces the expected output demonstrating the removal of one of the registration numbers that start with ‘T’ or earlier:

```
items before removing: [L22SBG, V53PLS, W79TRV, E16UEL, X85ADZ]
items after removing: [V53PLS, W79TRV, X85ADZ]
```

An alternative approach we could have taken is to use the `forEachRemaining` method from Table 15.2. This allows us to specify an action to be carried out over the remaining items in an `Iterator` via a lambda expression. As noted in Table 15.2, we provide an implementation for the `Consumer` interface to code this action. The `ForEachRemainingDemo` program rewrites the `IteratorDemo` program to illustrate the use of this method:

ForEachRemainingDemo

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class ForEachRemainingDemo
{
    public static void main(String[] args)
    {
        Set<String> regNums = new HashSet<>();
        regNums.add("V53PLS");
        regNums.add("X85ADZ");
        regNums.add("L22SBG");
        regNums.add("W79TRV");
        regNums.add("E16UEL");
        System.out.println("items before removing: " + regNums);
        Iterator<String> elements = regNums.iterator();
        // the lambda expression is applied to each remaining item in the collection
        elements.forEachRemaining ( item ->
                                    {
                                        if(item.charAt(0) <= 'T')
                                        {
                                            elements.remove();
                                        }
                                    } );
        System.out.println(regNums);
    }
}
```

Effectively, behind the scenes, the `forEachRemaining` method repeatedly calls the `hasNext` method on remaining items (in this case that is all items) and applies the given lambda expression to an item retrieved by a `next` method. The output will be the same as the output of the `IteratorDemo` before.

15.6 The *Map* Interface and the *HashMap* Class

The `Map` interface is not one of the interfaces that inherit from the general `Collections` interface. The Map interface is separate from this group and defines the methods required to process a collection consisting of *pairs* of objects. Rather than looking up an item via an index value, the first object of the pair is used. The first object in the pair is considered a **key**, and the second its associated **value**. Ordering is unimportant in maps, and keys are unique.

It is often useful to think of a map as a *look-up* table, with the key object the item used to look up (access) an associated value in the table. For example, the password of users of a network can be looked up by entering their username. Table 15.3 gives an example of such a look-up table.

We can look up the password of a user by looking up their user name in Table 15.2. The password of *lauraHaliwell*, for example, is *unicorn*, whereas the password of *wendyHarris* is *bumble*. Notice that it is important we make usernames the key of the look-up table and *not* passwords. This is because usernames are unique—no two users can have the same username. However, passwords are not unique. Two or more users may have the same password. Indeed, in Table 15.2, two users (*lauraHaliwell* and *lucyLane*) do have the same password (*unicorn*).

Let us implement this kind of look-up table using a `Map`. There are three implementations provided for the `Map` interface: `HashMap`, `TreeMap` and `LinkedHashMap`. Here we will look at the `HashMap` class.

The constructor creates the empty map:

```
Map<String, String> users = new HashMap<>();
```

Table 15.3 A look-up table for users of a network

Username	Password
lauraHaliwell	unicorn
wendyHarris	bumble
bobbyMann	elephant
lucyLane	unicorn
kabirMohan	magic

As before the type of the collection is given as the interface: Map. Notice that to use the generics mechanism to fix the types used in a Map object, we must provide *two* types in the angle brackets. The first type will be the type of the key and the second the type of its associated value. In this case, *both* are String objects, but in general each may be of any object type.

15.6.1 *Map* Methods

To add a user's name and password to this map we use the put method as follows. The put method requires two parameters, the key object and the value object:

```
users.put("lauraHaliwell", "unicorn");
```

Note that the put method treats the first parameter as a key item and the second parameter as its associated value. Really, we should be a bit more careful before we add user IDs and passwords into this map—only user IDs that are not already taken should be added. If we did not check this, we would end up overwriting a previous user's password. The containsKey method allows us to check this. This method accepts an object and returns **true** if the object is a key in the map and **false** otherwise:

```
if (users.containsKey("lauraHaliwell")) // check if ID taken
{
      System.out.println("user ID already taken");
}
else // ok to use this ID
{
      users.put("lauraHaliwell", "unicorn");
}
```

Notice we do not need to check that the password is unique as multiple users can have the same password. If we did require unique passwords the containsValue method could be used in the same way we used the containsKey method above.

Later a user might wish to change his or her password. The put method overrides the value associated with a key if that key is already present in the map. The following changes the password associated with "lauraHaliwell" to "popcorn":

```
users.put("lauraHaliwell", "popcorn");
```

The put method returns the value that was overwritten, or **null** if there was no value before, and this can be checked if necessary.

Later, a user might be asked to enter his or her ID and password before being able to access company resources. The `get` method can be used to check whether or not the correct password has been entered. The `get` method accepts an object and searches for that object among the keys of the map. If it is found, the associated value object is returned. If it is not found the **`null`** value is returned:

```
System.out.print("enter user ID ");
String idIn = EasyScanner.nextString(); // requires EasyScanner class
System.out.print("enter password ");
String passwordIn = EasyScanner.nextString();// requires EasyScanner class
// retrieve the actual password for this user
String password = users.get(idIn);
// password will be 'null' if the user name was invalid
if (password != null)
{
       if(passwordIn.equals(password))// check password is correct
       {
          // allow access to company resources here
       }
       else // invalid password
       {
          System.out.println ("INVALID PASSWORD!");
       }
}
else // no such user
{
       System.out.println ("INVALID USERNAME!");
}
```

As you can see, once the user has entered what they believe to be their username and password, the actual password for the given user is retrieved using the `get` method.

We know the password retrieved will be of type `String` as we created our `Map` by specifying that both the keys and values of the `Map` object would be `Strings`. We can then check whether this password equals the password entered by calling the `equals` method of the `String` class:

```
if(passwordIn.equals(password))
```

We have to be careful when we use the `equals` method to compare two objects in the way that we have done here. In this case, we are comparing the password entered by the user with the password obtained by the `get` method. However, the `get` method might have returned a **`null`** value instead of a password (if the key entered was invalid). The `equals` method of the `String` class does not return **`false`** when comparing a `String` with a **`null`** value, instead it throws a `NullPointerException`. So, to avoid this exception, we must check the value returned by the `get` method is not **`null`** *before* we use the `equals` method:

```
// check password returned is not 'null' before calling 'equals' method
if (password!= null)
{
    if(passwordIn.equals(password))// now it is safe to call 'equals'
    {
       // allow access to company resources
    }
    else
    {
       System.out.println ("INVALID PASSWORD!")
    }
}
```

Note that the **null** value is always checked with primitive comparison operators (== for equality and != for inequality).

Like all the other Java collection classes, the `HashMap` class provides a `toString` method so that the items in the map can be displayed:

```
System.out.print(users); // implicitly calls 'toString' method
```

Key and value pairs are displayed in braces. Let us assume we have added two more users: "bobbyMann" and "wendyHarris", with passwords "elephant" and "bumble" respectively. Displaying the map would produce the following output:

{lauraHaliwell=popcorn, wendyHarris=bumble, bobbyMann=elephant}

As with a set, the order in which the items are displayed is not determined by the order in which they were added but upon how they have been stored internally.

As with the other collections, a map provides a `remove` method. In the case of map, a key value is given to this method. If the key is present in the map both the key and value pair are removed:

```
// this removes the given key and its associated value
users.remove("lauraHaliwell");
```

Displaying the map now shows the user's ID and password have been removed:

{wendyHarris=bumble, bobbyMann=elephant}

The `remove` method returns the value of the object that has been removed, or **null** if the key was not present. This value can be checked if necessary to confirm that the given key was in the map.

Finally, the map collection provides `size` and `isEmpty` methods that behave in exactly the same way as the `size` and `isEmpty` methods for sets and lists.

15.6.2 Iterating Through the Elements of a Map

In order to scan the items in the map, we can use any of the methods discussed so far (the enhanced **for** loop, `Iterators` or the **forEach** loop).

An enhanced **for** loop cannot directly be used with a map as it is designed for collections of single values such as arrays, lists and sets. A map, however, consists of pairs of values. To use an enhanced **for** loop with a map we can extract the set of keys, using the `keySet` method:

```
// the keySet method returns the keys of the map as a set object
Set<String> theKeys = users.keySet();
```

Table 15.4 Reminder of the `BiConsumer` interface

Functional interface	Abstract method name	Parameter types	Return type
`BiConsumer<T, U>`	`accept`	`T, U`	`void`

Again notice that we know this set of keys returned by the `keySet` method will be a set of `String` objects, so we mark the type of this set accordingly.

An enhanced **`for`** loop can now be used to iterate through the keys of the map; within the loop we can look up the associated password using the `get` method. For example, we might wish to display the contents of the map in our own way, rather than the format given to us by the `toString` method:

```
for(String username: theKeys)// iterate through the set of keys
{
     String password = users.get(item); // retrieve password value
     System.out.println(username + "\t" + password); // format output
}
```

This would display the map in the following table format:

```
lauraHaliwell       popcorn
wendyHarris         bumble
bobbyMann           elephant
```

While the enhanced **`for`** loop is restricted to iterating over single values, the**`forEach`** loop can be used with pairs of values as well as a single value, as the `forEach` method is overloaded to take a `BiConsumer` (see Table 15.4 for a reminder) as well as a `Consumer` implementation.

Here is the `forEach` implementation for displaying items in our map:

```
// the forEach loop can take two parameters, representing the key and value item of a map pair
users.forEach((username, password) -> System.out.println(username + "\t" + password));
```

You can see that the two parameters given to this `forEach` method represent the key and its associated value respectively. In this case the key is the username and its associated value is a password.

15.7 Using Your Own Classes with Java's Collection Classes

In the examples above, we stuck to the pre-defined `String` type for the type of objects used in the collection classes of Java. However, objects of any class can be used inside these collections—including objects of your own classes. Care needs to be taken, however, when using your own classes. As an example, let us consider an application to store a collection of books that a person may own.

15.7.1 The *Book* Class

Figure 15.1 gives the UML design for a Book class.

This class consists of an ISBN number, a title and an author. An ISBN number is a unique International Standard Book Number that is allocated to each new book title. Before we deal with a *collection* of books, here is the initial implementation of the Book class.

The initial Book class

```
public class Book
{
   private String isbn;
   private String title;
   private String author;

   public Book(String isbnIn, String titleIn, String authorIn)
   {
      isbn = isbnIn;
      title = titleIn;
      author = authorIn;
   }

   public String getISBN()
   {
      return isbn;
   }

   public String getTitle()
   {
      return title;
   }

   public String getAuthor()
   {
      return author;
   }
}
```

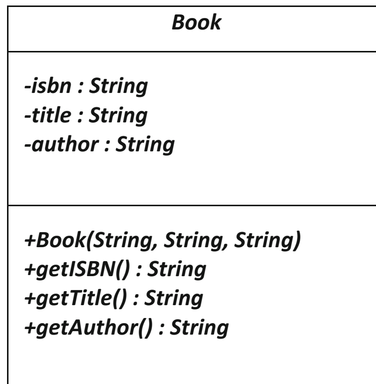


Fig. 15.1 Initial design of the *Book* class

There is nothing new here so let's move on to see how objects of this class can be stored in the Java collection classes. To begin with, let's create a list to contain `Book` objects:

```
// create empty list to contain Book objects
List<Book> books = new ArrayList<> ();
```

Note once again the use of the interface type, `List`, and generics to fix the collection type to `Book`. Now let's add some book objects and display the list:

```
// create two Book objects
Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
Book b2 = new Book("2222222222", "Travel With Me", "Sandy Beach");
// add Book objects to list
books.add(b1);
books.add(b2);
System.out.println(books); // implicitly call toString method of List
```

As things stand this will display something like the following list on the screen:

[Book@15db9742, Book@6d06d69c]

This isn't exactly what we require! When Java collections are displayed on the screen by calling their `toString` method, each object in the collection is displayed by calling its own `toString` method. As we said explained in Chap. 9, this will call the default `toString` method inherited from the `Object` class, and all this does is display information on the memory address of the object.

If you wish to use objects of your own classes as types in the Java collection classes, in Java's `toString` method a meaningful `toString` method should be defined in your class if you intend to make use of the toString method of the collection class.

Here is one possible `toString` method we could provide for our `Book` class:

```
@Override
public String toString()
{
   return "(" + isbn +", " + title + ", " + author +  ")";
}
```

We create a single `String` by joining the ISBN, title and author `Strings`. To improve the look of this `String` we have separated the attributes by commas and have enclosed it in round brackets. Notice the **`@Override`** annotation here, as we are overriding the `toString` method from the superclass (`Object`).

Now if we print out the list we get the following:

[(9999999999, Clowning Around, Joe King), (2222222222, Travel With Me, Sandy Beach)]

While you might wish to improve this `String` representation of a book further (maybe by adding more formatting), this is acceptable for testing purposes.

15.7.2 Defining an *equals* Method

Another issue arises if we wish to use methods such as contains and remove. Methods such as these call the equals method of the contained object to determine whether or not a given object is in the collection. The Book class does not define its own equals method so again the inherited equals method of the Object class is called. This method is inadequate as it simply compares the memory address of two objects rather than the attributes of two objects. For example, the following would display **false** on the screen when we would want it to display **true**:

```
// check a book that is in the list
boolean check = books.contains(new Book("9999999999", "Clowning Around", "Joe King"));
// display result
System.out.println(check);
```

Here, while the Book parameter has the same attribute data as one of the books in the list, it is a new Book object stored in a different memory location and so the contains method will return **false**.

To use objects of your own classes effectively, a meaningful equals method should be defined. One possible interpretation of two books being equal is simply that their ISBNs are equal, so the following equals method could be added to the Book class:

```
@Override
public boolean equals (Object objIn) // equals method must have this header
{
    Book bookIn = (Book) objIn; // type cast to a Book
    // check isbn
    return  isbn.equals(bookIn.isbn);
}
```

Again, we have added an **@Override** annotation. Notice that the equals method must accept an item of type Object. The body of the method then needs to type cast this item back to the required type (Book in this case).

If you are using lists, these are the only two additional methods you need to provide in your class. Also, if you are using objects of your own classes only as *values* of a map, then again these are the only two additional methods you need to provide. If, however, you are using objects of your classes as keys of a map or as the items of a set, then you need to include an additional method in your classes: hashCode.

15.7.3 Defining a *hashCode* Method

To understand how to define your own hashCode method you need to understand how the HashSet and HashMap implementations work. Both of these Java classes have been implemented using an array. Unlike the ArrayList class

however (which has also been implemented using an array), the position of the items in the `HashSet` and `HashMap` arrays is not determined by the order in which they were added. Instead, the position into which items are added into these arrays is determined by their `hashCode` method.

The `hashCode` method returns an integer value from an object. This integer value determines where in the array the given object is stored. Objects that are equal (as determined by the `equals` method) should produce identical `hashCode` numbers and, ideally, objects that are not equal should return different `hashCode` numbers.

The reason for using `hashCode` numbers is that they considerably reduce the time it takes to search a given array for a given item. If items were stored consecutively, then a search of the array would require *every* item in the array being checked using the `equals` method, until a match was found. This would become very inefficient when the collection becomes very large. So, instead, the `HashSet` and `HashMap` classes make use of the `hashCode` method, so that when a search is required for an item, say `x`, that `hashCode` number is computed and this value is used to look up other items in the array. Then, only objects with the same `hashCode` number are compared to `x` using their `equals` method.

If you are using objects of your own classes, and you have not provided a `hashCode` method, the inherited `hashCode` method from the `Object` class is called. This does not behave in the way we would wish. It generates the `hashCode` number from the memory address of the object, so two "equal" objects could have different `hashCode` values. The `TestHashCode` program demonstrates this.

TestHashCode

```
public class TestHashCode
{
    public static void main (String[] args)
    {
        // create two "equal" books
        Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
        Book b2 = new Book("9999999999", "Clowning Around", "Joe King");
        // check their hashCode numbers
        System.out.println(b1.hashCode());
        System.out.println(b2.hashCode());
    }
}
```

When we run this program it produces the following `hashCode` numbers for the two "equal" books:

1044036744
1826771953

As you can see, the `hashCode` numbers do not match—even though the objects are "equal". This means that if we were searching for the given book in a `HashSet` or in the keys of a `HashMap`, the book would not be found, as only objects with identical `hashCode` values are checked. Also, we cannot ensure that objects in the `HashSet` or the keys of the `HashMap` will be unique, as two (or more) identical books (with different `hashCode` values) would both be stored in the underlying array at different array positions.

We need to define our own hashCode method for the Book class so that objects of this class can be used effectively with the HashSet and HashMap classes.

Luckily, all of Java's predefined classes (such as String) have a meaningful hashCode method defined. These hashCode methods return equal hashCode numbers for "equal" objects.

So one way of defining the hashCode number for an object of your class would be to add together the hashCode numbers generated by all the attributes to determine object equality. For Book equality we checked the ISBN only. This ISBN is a String, so all we need to do is to return the hashCode number of this String:

```
@Override
// this is a suitable hashCode method for our Book class
public int hashCode()
{
    // derive hash code by returning hash code of ISBN string
    return isbn.hashCode();
}
```

If you have more than one attribute that plays a role in determining object equality, then add each such hashCode numbers (assuming the attribute is not of primitive type) to determine your hashCode number.

If primitive attributes also play a part in object equality, they too can simply be added into your hashCode formula by generating an integer value from each primitive attribute. Here is a simple set of guidelines for generating an integer value from the primitive types (although much more sophisticated algorithms exist than these!):

- **byte**, **short**, **int**, **long**: leave as they are;
- **float**, **double**: type cast to an integer;
- **char**: use its Unicode value;
- **boolean**: use an **if…else** statement to allocate 1 if the attribute is **true** and 0 if it is **false**.

15.7.4 The Updated *Book* Class

The complete code for the updated Book class is now presented below with toString, equals and hashCode methods included:

The updated Book class

```
public class Book
{
   private String isbn;
   private String title;
   private String author;

   public Book(String isbnIn, String titleIn, String authorIn)
   {
      isbn = isbnIn;
      title = titleIn;
      author = authorIn;
   }

   public String getISBN()
   {
      return isbn;
   }

   public String getTitle()
   {
      return title;
   }

   public String getAuthor()
   {
      return author;
   }

   @Override
   public String toString()
   {
      return "(" + isbn +", " + title + ", " + author +  ")";
   }

   @Override
   public boolean equals (Object objIn)
   {
      Book bookIn = (Book) objIn; // type cast to a Book
      // check isbn
      return  isbn.equals(bookIn.isbn);
   }

   @Override
   public int hashCode()
   {
      // derive hash code by returning hash code of ISBN string
      return isbn.hashCode();
   }
}
```

15.8 Developing a Collection Class for *Book* Objects

In the previous section we amended the `Book` class by supplying it with a `toString` method, an `equals` method and a `hashCode` method. If you look at the documentation of any class in the Java API you will see that it also possesses these three methods. When developing classes professionally you should always include these methods. That way, objects of these classes can be used with any collection type.

Now we have a suitable `Book` class we can store objects of this class in one of the Java collection classes. Which collection shall we use? There is no ordering required on the collection of books so we do not really need a list here. We could store these books in a set, but since each book has a unique ISBN it makes more

sense to use a map and have ISBNs as the keys to the map, with `Book` objects themselves as the values of the map.

We will use this collection to help us develop our own class, `Library`, which keeps track of the books that a person may own. Figure 15.2 gives its UML design.

Notice that the keys of the map are specified to be `String` objects (to represent ISBNs), and the values of the map are specified as `Book` objects.

Here is the implementation of the `Library` class. Take a look at it and then we will discuss it.

Library

```
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.HashSet;


public class Library
{
   Map <String, Book> books; // declare map collection
   // create empty map
   public Library()
   {
      books = new HashMap<> ();
   }

   // add the given book into the collection
   public boolean addBook(Book bookIn)
   {
      String keyIn = bookIn.getISBN(); // isbn will be key of map
      if (books.containsKey(keyIn)) // check if isbn already in use
      {
         return false; // indicate error
      }
      else // ok to add this book
      {
         books.put(keyIn, bookIn); // add key and book pair into map
         return true;
      }
   }

   // remove the book with the given isbn
   public boolean removeBook(String isbnIn)
   {
      if (books.remove(isbnIn)!= null) // check if item was removed
      {
         return true;
      }
      else // when item is not removed
      {
         return false;
      }
   }

   // return the number of books in the collection
   public int getTotalNumberOfBooks()
   {
      return books.size();
   }

   // return the book with the given isbn or null if no such book
   public Book getBook (String isbnIn)
   {
      return books.get(isbnIn);
   }

   // return the set of books in the collection
   public Set<Book> getAllBooks ()
   {
      Set<Book> bookSet = new HashSet<>(); // to store the set of books
      books.forEach((key,book) -> bookSet.add(book)); // use forEach loop to add books
      return bookSet; // return the set of books
   }
}
```

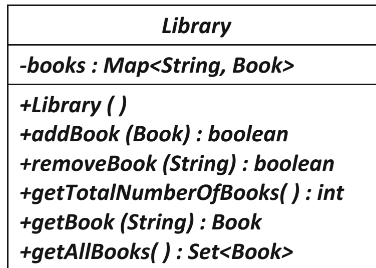


Fig. 15.2 UML design for the *Library* class

Most of this class should be self-explanatory. Just consider how complicated some of these methods would have been if we had used an array instead of a map!

We just draw your attention to the `getAllBooks` method. The UML diagram indicates that this method should return a *set* of books. An alternative approach could have been to return the map itself, but it would be more useful for this method to return a set of objects, as a set is easier to scan than a map. It is fine to use a set as there is no repetition of books.

```
public Set<Book> getAllBooks ()
{
  // code to create this set goes here
}
```

There is no map method that returns the set of values in the map directly,[3] so a suitable set needs to be created in this method. We begin by creating an empty set:

```
Set<Book> bookSet = new HashSet<>();
```

This set is empty initially and we must fill it with book objects. In order to access these values we use a **forEach** loop and simply add the values of the map into this set.

```
books.forEach((key,book) -> bookSet.add(book)); // use forEach loop to add books
```

Finally, we return this set of books:

```
return bookSet;
```

That completes our discussion of the `Library` class. We leave the task of creating a tester for this class to the end of chapter exercises.

[3]There are several approaches to indirectly create a set of values in a map. Here we look at just one approach.

15.9 Sorting Objects in a Collection

We have seen how we can use implementations of the `List` interface (such as `ArrayList`) to create an ordered collection in our programs. Previously we have seen how arrays can also be used to store an ordered collection. There may be times when you wish to sort these collections so they are presented in a different order. For example, you may have an ordered collection of students and wish to re-order these students based on their student ID, or their final mark.

15.9.1 The *Collections.sort* and *Arrays.sort* Methods

Sorting a collection can involve complex algorithms. Luckily Java provides two classes that contain class methods to sort your ordered collections. The `Collections` class and the `Arrays` class both have a number of utility methods for processing collections in the JCF and arrays respectively. In particular, both classes contain a `sort` method to sort a given list or array. Both classes are in the `java.util` package. The `StringSortDemo` program below demonstrates how to use these methods with an ordered collection of `String` objects. It introduces a few new additional concepts. Take a look at it and then we will discuss it:

StringSortDemo

```
import java.util.Collections;
import java.util.Arrays;
import java.util.List;

public class StringSortDemo
{
    public static void main(String[] args)
    {
        // create array of strings
        String[] citysArray = {"London", "Birmingham", "Manchester", "Liverpool"};
        // display array using Arrays.toString
        System.out.println("Original Array " + Arrays.toString(citysArray));
        // convert array to List using Arrays.asList
        List<String> citysList = Arrays.asList(citysArray);
        // display List
        System.out.println("Original List " + citysList);
        // sort array
        Arrays.sort(citysArray);
        // display array
        System.out.println("Sorted Array " + Arrays.toString(citysArray));
        // sort List
        Collections.sort(citysList);
        // display List
        System.out.println("Sorted List " + citysList);
    }
}
```

First you can see we have gone back to using an array type to store a collection `String` objects:

```
// create array of strings
String[] citysArray = {"London", "Birmingham", "Manchester", "Liverpool"};
```

The Arrays utility class has a toString class method that allows us to convert an array into a String which may then be displayed on the screen:

```
// display array using toString
System.out.println("Original Array " + Arrays.toString(citysArray));
```

Next we use the asList method of Arrays that returns an equivalent List object containing the same items in the collection, in the same order—we store this in a new List object, which we then display:

```
// convert array to List using asList
List<String> citysList = Arrays.asList(citysArray);
// display List
System.out.println("Original List " + citysList);
```

Now the really useful sort methods that re-order the given collection of strings so they are sorted in ascending alphabetical order:

```
// sort array
Arrays.sort(citysArray);
// display array using toString
System.out.println("Sorted Array " + Arrays.toString(citysArray));
// sort List
Collections.sort(citysList);
// display List using toString
System.out.println("Sorted List " + citysList);
```

You can see that in order to sort an array we use the Arrays.sort method and to sort a List we use the Collections.sort method. Here is the output from this program:

Original Array [London, Birmingham, Manchester, Liverpool]
Original List [London, Birmingham, Manchester, Liverpool]
Sorted Array [Birmingham, Liverpool, London, Manchester]
Sorted List [Birmingham, Liverpool, London, Manchester]

As expected the sort methods re-order the items in the collection so that they are now in ascending lexicographical order. Note that the Arrays.toString method returns a String representation of the array in the same format as a List (i.e. with items separated by commas and enclosed in square brackets).

For these sort methods to work the contained object needs to be derived from a class that implements the generic Comparable<T> interface.

15.9.2 The *Comparable<T>* Interface

The generic Comparable<T> interface consist of a single compareTo method that accepts an object of type T and returns an integer (see Table 15.5).

A compareTo method should return a positive integer when the first object is greater than the second, a negative integer when it is less than the second and zero when the two objects are equal. As we saw in Chap. 7, the String class does have such a method defined and so implements the Comparable<T> interface.

But what happens if we try to sort a collection of objects that do not have a compareTo method defined? Well, the Collections.sort method would result in a compiler error, whereas the Arrays.sort method would not give a compiler error but would throw an exception at runtime.

To sort ordered collections of objects derived from classes that you have defined you will need to ensure their associated class implements the Comparable<T> interface.

As an example, let's return to our Book class. We would be unable to use the sort methods we have met to sort lists or arrays of Book objects as the Book class does not implement the Comparable<T> interface. To implement this interface we need to define a suitable compareTo method. One way of comparing two Book objects might be just to compare their ISBN numbers. Since ISBN numbers have been stored as Strings, we just need to use the compareTo method of String and use that result as our Book compareTo result. Here is the code:

A sortable Book class

```
// this class can be used in conjunction with Collections.sort and Arrays.sort

public class Book implements Comparable <Book> // notice generic type fixed to Book
{
    //attributes and methods as before


   // add a compareTo method

   @Override
    public int compareTo(Book bIn)
    {
         return isbn.compareTo(bIn.isbn); // compare ISBN numbers
    }
}
```

Notice how we use the generics mechanism to indicate the type of object the Comparable interface will be fixed to. In this case it is set to Book as we wish to compare two books.

Table 15.5 The Comparable<T> interface

Functional interface	Abstract method name	Parameter types	Return type
Comparable<T>	compareTo	T	int

The BookSortDemo1 program below makes use our updated Book class to sort a list of Book objects.

BookSortDemo1

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;

public class BookSortDemo1
{
    public static void main(String[] args)
    {
       // create three Book objects
       Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
       Book b2 = new Book("2222222222", "Travel With Me", "Sandy Beach");
       Book b3 = new Book("4444444444", "Interior Design", "Anita Room");
       // create an empty list of books
       List<Book> bookList = new ArrayList<>();
       // add the three Book ojects
       bookList.add(b1);
       bookList.add(b2);
       bookList.add(b3);
       // display list before and after sorting
       System.out.println("***COMPARABLE DEMO***");
       System.out.println("\nBefore Sort\n"+bookList);
       Collections.sort(bookList); // uses the Book compareTo method
       System.out.println("\nAfter sort\n"+ bookList);
    }
}
```

Here is the output from this program:

```
***COMPARABLE DEMO***

Before Sort
[(9999999999, Clowning Around, Joe King), (2222222222, Travel With Me, Sandy Beach), (4444444444, Interior Design, Anita Room)]

After sort
[(2222222222, Travel With Me, Sandy Beach), (4444444444, Interior Design, Anita Room), (9999999999, Clowning Around, Joe King)]
```

As you can see the Book objects have been sorted on ISBN numbers.

15.9.3 The *Comparator<T>* Interface

The previous section demonstrated one technique for comparing objects for sorting purposes; implement the Comparable<T> interface. This technique is fine if we always wish to use the same criteria for sorting objects; in our case if we always wish to sort Book objects by their ISBN number.

But what if you wish to sort your collection by some other criteria, or multiple criteria? For example, we may wish to sort our list of Book objects via their title, at other times we might wish to sort via the author name. The Comparable<T>

Table 15.6 The Comparator interface

Functional interface	Abstract method name	Parameter types	Return type
Comparator<T>	compare	T, T	int

interface only allows us to define a single criteria for sorting purposes. But since Java 8 a more flexible way of doing this with the Comparator<T> interface has been introduced.

The Comparator<T> interface requires us to provide an implementation for a compare method (see Table 15.6).

The compare method accepts two objects of a given type and (as with the compareTo method) returns a positive integer indicating the first object is greater, or a negative integer if the first object is less than the second object, or zero if both objects are equal.

The List interface contains a **default** method[4] called sort that takes a single parameter of type Comparator. Since this is a **default** method of List it is available to ArrayList objects. BookSortDemo2 makes use of this sort method to sort the books based on author names. Take a look at it and then we will discuss it.

BookSortDemo2

```
import java.util.List;
import java.util.ArrayList;

public class BookSortDemo2
{
    public static void main(String[] args)
    {
       // create three Book objects
       Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
       Book b2 = new Book("2222222222", "Travel With Me", "Sandy Beach");
       Book b3 = new Book("4444444444", "Interior Design", "Anita Room");
       // create an empty list of books
       List<Book> bookList = new ArrayList<>();
       // add the three Book ojects
       bookList.add(b1);
       bookList.add(b2);
       bookList.add(b3);
       // display list before and after sorting
       System.out.println("***COMPARATOR DEMO***");
       System.out.println("\nBefore Sort\n"+bookList);
       // sort list using sort method of List and a Comparator implementation using lambda
       bookList.sort((book1,book2)  ->  {return  book1.getAuthor().compareTo(book2.getAuthor());});
       System.out.println("\nAfter Author sort\n"+ bookList);
    }
}
```

The key instruction here is call to bookList.sort(). You can see that this accepts an implementation of a compare method. The compare method takes two parameters representing two books that need to be compared. The given

[4]Remember from Chap. 13, **default** methods were added in Java 8 and are fully implemented methods within an interface.

lambda expression uses the author field of the two `Book` objects in order to compare them (using the `compareTo` method of `String`):

```
// note the lambda expression to provide a Comparator implementation
bookList.sort((book1,book2) -> {return book1.getAuthor().compareTo(book2.getAuthor());});
```

When we run this program we get the expected output:

```
***COMPARATOR DEMO***

Before Sort
[(9999999999, Clowning Around, Joe King), (2222222222,
Travel With Me, Sandy Beach), (4444444444, Interior Design,
Anita Room)]

After Author sort
[(4444444444, Interior Design, Anita Room), (9999999999,
Clowning Around, Joe King), (2222222222, Travel With Me,
Sandy Beach)]
```

If we wished to use another attribute for comparison, such as the title, we could replace the call to `getAuthor` with a call to `getTitle` (for example). In each case we could use the `compareTo` method to compare the appropriate attribute which acts as the key to our sort.

This pattern of picking a sort key (ISBN, author, title etc.) and then comparing two keys using `compareTo` is a very common approach to implementing a `Comparator` and a **`static`** `comparing` method of `Comparator` is provided in order to simplify this task. As mentioned in Chap. 13, since Java 8 interfaces can contain **`static`** methods.

The `comparing` method is provided with an appropriate sort key via a suitable method (`getAuthor` for the author key, `getTitle` for the title key and so on). Behind the scenes the `comparing` method then compares two keys for us using their `compareTo` method. The appropriate key can be sent to the `comparing` method via the method reference (double colon) notation we discussed in Chap. 13. `BookSortDemo3` illustrates how we can sort via all three attribute keys of a book (ISBN, author and title) using this technique:

BookSortDemo3

```
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

public class BookSortDemo3
{
    public static void main(String[] args)
    {
        // create three Book objects
        Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
        Book b2 = new Book("2222222222", "Travel With Me", "Sandy Beach");
        Book b3 = new Book("4444444444", "Interior Design", "Anita Room");
        // create an empty list of books
        List<Book> bookList = new ArrayList<>();
        // add the three Book ojects
        bookList.add(b1);
        bookList.add(b2);
        bookList.add(b3);
        // display list before and after sorting
        System.out.println("***COMPARATOR DEMO***");
        System.out.println("\nBefore Sort\n"+bookList);

        // sort list using getISBN method reference
        bookList.sort(Comparator.comparing(Book::getISBN));
        System.out.println("\nAfter ISBN sort\n"+ bookList);
        // sort list using getAuthor method reference
        bookList.sort(Comparator.comparing(Book::getAuthor));
        System.out.println("\nAfter Author sort\n"+ bookList);
        // sort list using the getTitle method reference
        bookList.sort(Comparator.comparing(Book::getTitle));
        System.out.println("\nAfter Title sort\n"+ bookList);
    }
}
```

Here is the expected output:

```
***COMPARATOR DEMO***

Before Sort
[(9999999999, Clowning Around, Joe King), (2222222222,
Travel With Me, Sandy Beach), (4444444444, Interior Design,
Anita Room)]

After ISBN sort
[(2222222222, Travel With Me, Sandy Beach), (4444444444,
Interior Design, Anita Room), (9999999999, Clowning Around,
Joe King)]

After Author sort
[(4444444444, Interior Design, Anita Room), (9999999999,
Clowning Around, Joe King), (2222222222, Travel With Me,
Sandy Beach)]

After Title sort
[(9999999999, Clowning Around, Joe King), (4444444444,
Interior Design, Anita Room), (2222222222, Travel With Me,
Sandy Beach)]
```

The `comparing` method will work on numeric attributes too (such as **`doubles`**, **`ints`** and **`longs`**) via the `compareTo` method provided in the equivalent wrapper classes. However, to reduce the overhead of boxing and unboxing primitive values you can use specific `comparing` methods for these types (`comparingDouble`, `comparingInt`, `comparingLong` etc.). You will see an example of this in Chap. 22.

It is worth pointing out that you can always revert to the natural ordering key defined in your own `compareTo` method by using the **`static`** `naturalOrder` method of `Comparator`.

We used ISBNs as our built-in comparison key when writing the `compareTo` method of `Book`, so instead of the following line from `BookDemo4`:

```
bookList.sort(Comparator.comparing(Book::getISBN));
```

we could have used the `naturalOrder` method instead as follows:

```
bookList.sort(Comparator.naturalOrder());
```

Note that if the underlying list contains strings or numeric values only, `naturalOrder` will sort the given list in alphabetical or numerical order respectively.

To find out more about the `Collections.sort`, `Arrays.sort` and the `sort` method of `List`, as well as more about the `Comparable<T>` and `Comparator<T>` interfaces, you can refer to the Oracle™ site.

15.10 Self-test Questions

1. Distinguish between the following types of collection in the Java Collections Framework:

   - `List`;
   - `Set`;
   - `Map`.

2. Identify, with reasons, the most appropriate Java collection type to implement the following collections:

   (a) An ordered collection of patient names waiting for a doctor;
   (b) An unordered collection of patient names registered to a doctor;
   (c) An unordered collection of `BankAccount` objects.

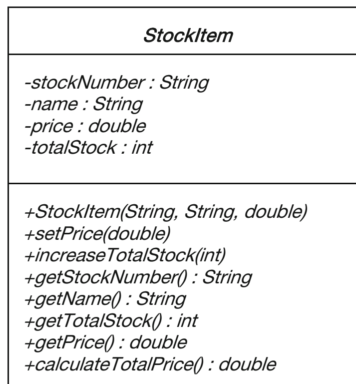
3. Consider the following instruction:

```
Map <String, Student> javaStudents = new HashMap<>();
```

   (a) Why is the type of this object given as `Map` and not `HashMap`?
   (b) Assuming the object `javaStudents` has been created as above, why would the following line cause a compiler error?

```
javaStudents.put("u0012345", "Jeet");
```

4. Consider again the `StockItem` class from the programming exercises of Chap. 8. Here is the UML diagram:



   (a) Identify the purpose of a `toString` method and define an appropriate `toString` method for this class.
   (b) Identify the purpose of an `equals` method and define an appropriate `equals` method for this class.
   (c) Identify the purpose of a `hashCode` method and define an appropriate `hashCode` method for this class.

5. In Sect. 15.5 a set called `regNums` was created to store a collection of car registration numbers.

   (a) Write a fragment of code that makes use of an enhanced **`for`** loop to display all registrations ending in ‘S’.
   (b) Write a fragment of code that makes use of a **`forEach`** loop to display all registrations ending in ‘S’.
   (c) Write a fragment of code, which makes use of the `iterator` method, to remove all registration numbers ending in ‘S’.

6. In Chap. 8 we introduced a BankAccount class and a collection class to hold bank accounts called Bank. The Bank class was implemented using a List in the form of an ArrayList class.

   (a) If we were to change from a List to a Map, what would be the key type of the Map and what would be the value type?
   (b) Write a fragment of code that declares a Map to hold BankAccount objects and add two BankAccount objects into this map.
   (c) Write a fragment of code that uses a **forEach** loop to display all BankAccount objects in the map that have a balance over 100.

7. Consider the BankAccount class from Chaps. 7 and 8 once again. Assume we have the following array to store a collection of BankAccount objects:

```
BankAccount[] accountList = new BankAccount[5];
```

   Now assume five BankAccount objects have been added into this list.

   (a) What method of the Arrays class would allow you to display this array?
   (b) What method of the Arrays class would you allow you to create an equivalent List from this array?
   (c) Assuming the array has been converted to a list, describe how the Comparable<T> and Comparator<T> interfaces can be used to help sort the list, via the account number, by making use of the Collections.sort method and the sort method of List.

15.11 Programming Exercises

1. Copy, from the accompanying website, the Book and Library classes and then implement a tester for the Library class.

2. Make the changes to the StockItem class you considered in self-test question 4 above, then:

   (a) Write a tester that adds five StockItem objects into a set.
   (b) Modify the tester so that the set is displayed using the toString method of Set.
   (c) Use the contains method of Set method to search for one of the stock items.

3. Copy, from the accompanying website, the Book and BookSortDemo2 classes and then:

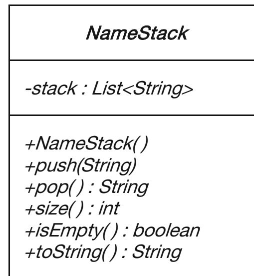
   (a) Rewrite the compareTo method (that implements the Comparable<Book> interface) in the Book class so the length of the title is used to compare two books. Run the BookSortDemo4 program to test this.
   (b) Modify the BookSortDemo4 program to sort the list of books based on length of title by using both the comparing method and naturalOrdering methods of Comparator.

4. Write a tester program to test your answers to self-test question 7 above.

5. In this chapter we looked at an example of a printer queue. A *queue* is a collection where the first item added into the queue is the first item removed from the queue. Consequently, a queue is often referred to as a *first in first out* (FIFO) collection.

   A *stack*, on the other hand, is a *last in first out* (LIFO) collection—much like a stack of plates, where the last plate added to the stack is the first plate removed from the stack. The method to add an item onto a stack is often called *push*. The method to remove an item from a stack is often called *pop*.

   Below, we give the UML design for a NameStack class, which stores a stack of names.



   A description of each NameStack method is given below:

   +NameStack()
   Initializes the stack of names to be empty.

   +push(String)
   Adds the given name onto the top of the stack.

   +pop() : String
   Removes and returns the name at the top of the stack. Throws a NameStackException (which will also need to be implemented) if an attempt is made to pop an item from an empty stack.

+size() : int
Returns the number of names in the stack.

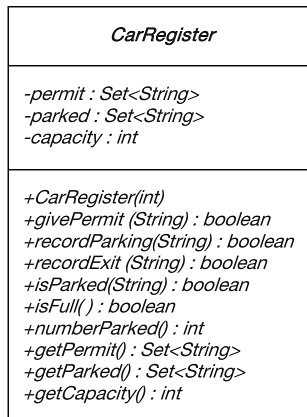
+isEmpty() : boolean
Returns **true** if the stack is empty and **false** otherwise.

+toString() : String
Returns a `String` representation of the stack of names.

(a) Implement the `NameStack` class. You will also need to implement a `NameStackException` class.

(b) Test the `NameStack` class with a suitable tester.

6. Consider an application that keeps track of the registration numbers of all cars that have a permit to use a company car park. It also keeps track of the registration numbers of the cars actually in the car park at any one time. While there is no limit to the number of cars that can have permits to park in the car park, the capacity of the car park is limited. Below we give the UML design for the `CarRegister` class:



A description of each `CarRegister` method is given below:

+CarRegister(int)
Initializes the `permit` and `parked` sets to be empty and sets the capacity of the car park with the given parameter. Throws a `CarRegisterException` (which will also need to be implemented) if the given parameter is negative.

+givePermit(String) : boolean
Records the registration of a car given a permit to park. Returns **false** if the car has already been given a permit and **true** otherwise.

+recordParking(String) : boolean
Records the registration of a car entering the car park. Returns **false** if the car park is full, or the car has no permit to enter the car park, and **true** otherwise.

+recordExit (String) : boolean
Records the registration of a car leaving the car park. Returns **false** if the car was not initially registered as being parked and **true** otherwise.

+isParked(String) : boolean
Returns **true** if the car with the given registration is recorded as being parked in the car park and **false** otherwise.

+isFull() : boolean

Returns **true** if the car park is full and **false** otherwise.

+numberParked() : int

Returns the number of cars currently in the car park.

+getPermit() : Set<String>

Returns the set of car registrations allocated permits.

getParked() : Set<String>

Returns the set of registration numbers of cars in the car park.

+getCapacity() : int

Returns the maximum capacity of the car park.

(a) Implement the `CarRegister` class. You will also need to implement a `CarRegisterException` class.
(b) Test the `CarRegister` class with a suitable tester.

7. Re-write the `Bank` class of Chap. 8 so that it makes use of a `Map` instead of a `List`, as discussed in self-test question 6 above.

Advanced JavaFX

16

Outcomes:

By the end of this chapter you should be able to:

- *program components to respond to* ***mouse events*** *and* ***key events***;
- *explain the term* ***property*** *as applied to JavaFX components*;
- ***bind*** *properties of two or more components*;
- *use a* ***slider*** *to change the value of a variable*;
- *embed images, videos and webpages in applications*;
- *format applications by using* ***cascading style sheets***.

16.1 Introduction

In this chapter we will explore the more advanced aspects of programming with JavaFX. We will begin by looking at the ways in which applications can be programmed to respond to input events such as keystrokes and mouse movements, rather than just simple mouse clicks. We will then look at the unique nature of the attributes of JavaFX classes and explain how we can add event handlers to the individual attributes of a component, rather than to the component itself.

We will then go on to explore the way in which we can turn our programs into multimedia applications, and finally we will introduce you to cascading style sheets so that you can keep your formatting information separate from the rest of the program, and apply different "skins" to an application.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_16

16.2 Input Events

In Chap. 10 we introduced you to the idea of events and event handling. The only event that we came across was an `ActionEvent`, the event that occurs when the mouse is clicked on a component. There are other events that are defined as subtypes of the `Event` class, the most important of which is the `InputEvent`. Two subtypes of `InputEvent` are discussed here—`MouseEvent` and `KeyEvent`. These occur, respectively, when a mouse button is pressed or the mouse is moved or dragged, and when a key is pressed on the keyboard.

16.2.1 Mouse Events

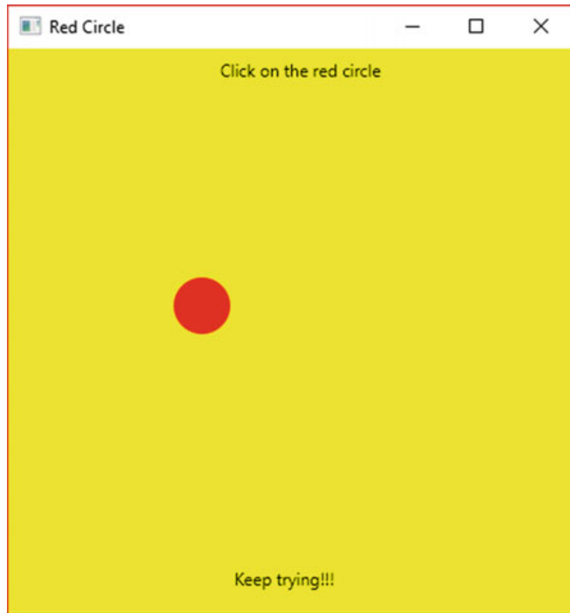
We saw in Chaps. 10 and 13 that a JavaFX `ActionEvent` has only one `EventType` defined (`ACTION`). In the case of `MouseEvent`, however, there are a number of `EventTypes`, the most common of which we have summarised in Table 16.1.

As you will see in a moment, there are convenience methods for all of these events—and in order to demonstrate some of these, we have developed a little game with which you can amuse your friends. Figure 16.1 shows how it looks when it runs. We have called it—rather unimaginatively—the `RedCircle` game; a red circle always moves away from the cursor so you can never click on it, despite being told to do so! And if in desperation you start to click the mouse, the words “Keep Trying” flash onto the screen!

Table 16.1 Types of mouse event

MOUSE_CLICKED	This event occurs when the mouse has been clicked (pressed and released)
MOUSE_PRESSED	This event occurs when the mouse button is pressed
MOUSE_RELEASED	This event occurs when the mouse button is released
MOUSE_MOVED	This event occurs when the mouse moves within a node and no buttons are pressed
MOUSE_DRAGGED	This event occurs when the mouse moves within a node with a pressed button
MOUSE_ENTERED	This event occurs when the mouse enters a node
MOUSE_EXITED	This event occurs when the mouse exits a node

Fig. 16.1 The *RedCircle* application



Here is the code for the class:

RedCircle

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.text.Text;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;

public class RedCircle extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 400;

        // circle starts in centre of screen
        Circle circle = new Circle(WIDTH/2, HEIGHT/2, 20, Color.RED);

        Text heading = new Text(WIDTH/2 - 50, 20, "Click on the red circle");
        Text message = new Text(WIDTH/2 - 40, HEIGHT - 20 , "");

        Group root = new Group(heading, circle, message);
        Scene scene = new Scene(root, WIDTH, HEIGHT, Color.YELLOW);

        /* when the mouse is move or dragged the centre of the circle is
           repositioned so that it is always 50 pixels to the left and
           50 pixels above the cursor */

        scene.setOnMouseMoved(e -> {
                                        circle.setCenterX(e.getX()-50);
                                        circle.setCenterY(e.getY()-50);
                                   }
                             );

        scene.setOnMouseDragged(e -> {
                                        circle.setCenterX(e.getX()-50);
                                        circle.setCenterY(e.getY()-50);
                                     }
                               );
```

```
            // a message is displayed when the mouse button is depressed
            scene.setOnMousePressed(e -> message.setText("Keep trying!!!"));

            // the message is blanked when the mouse button is released
            scene.setOnMouseReleased(e -> message.setText(""));

            stage.setScene(scene);
            stage.setTitle("Red Circle");
            stage.show();
        }

        public static void main(String[] args)
        {
            launch(args);
        }
    }
```

We have begun by declaring two constants `WIDTH` and `HEIGHT` to represent the dimensions of the scene, and then gone on to define a circle of radius 20 pixels that will start off at the centre of the graphic:

```
Circle circle = new Circle(WIDTH/2, HEIGHT/2, 20, Color.RED);
```

Next we create two instances of the `Text` class. The first is the heading, the second will hold the message that appears when the mouse button is depressed—this one starts off empty.

```
Text heading = new Text(WIDTH/2 - 50, 20, "Click on the red circle");
Text message = new Text(WIDTH/2 - 40, HEIGHT - 20 , "");
```

Next we group these three items together and add them to the scene.

```
Group root = new Group(heading, circle, message);
Scene scene = new Scene(root, WIDTH, HEIGHT, Color.YELLOW);
```

Now we come to the code that allows the graphic to respond to the mouse being moved.

```
scene.setOnMouseMoved(e -> {
                                circle.setCenterX(e.getX()-50);
                                circle.setCenterY(e.getY()-50);
                            }
                    );
```

Notice that we attach this to the scene itself. We use the convenience method `setOnMouseMoved` for this purpose. The idea is that whenever the mouse moves the circle also moves so that it is always 50 pixels above and 50 pixels to the left of the cursor.

So how do we find out where the cursor is? For this, we have made use of the parameter that is received by the `handle` method of `EventHandler`, to which

the lambda expression refers. This parameter, `e`, to the left of the arrow is of type `MouseEvent` and has methods `getX` and `getY` that return the current position of the cursor. We use these to set the centre of the circle to the desired position.

We want exactly the same thing to happen when the mouse is dragged (that is, it is moved with the button depressed), so the `setOnMouseDragged` method is coded in the same way. However, when the button is pressed, we also want the message at the bottom of the screen to change from blank to "Keep trying!!!", and for this purpose we use the `setOnMousePressed` method of `Scene`.

```
scene.setOnMousePressed(e -> message.setText("Keep trying!!!"));
```

When the button is released, the message returns to a blank message:

```
scene.setOnMouseReleased(e -> message.setText(""));
```

The final lines of code add the scene to the stage, set the title and make the stage visible, as you have seen in other examples.

16.2.2 Key Events

The other common input event is a key event. A key event occurs whenever a key is pressed or released. One of the common applications of this event is to check if the key pressed was the <Enter> key, which indicates that the entry is completed.

There are three common types of key event which are summarised in Table 16.2.

We have developed a couple of little applications to demonstrate these. The first one, which we have called `TextConverter`, allows the user to type something into a text field, and each time a character is entered, the content of the text field is displayed below in upper case. This is shown in Fig. 16.2—you can see we have used a similar interface to the `PushMe` class of Chap. 10, but this time there is no button, as the application responds each time a character is entered (as you will see, it actually responds when the key is released).

Table 16.2 Types of key event

KEY_TYPED	This event occurs when a key has been typed (pressed and released)
KEY_PRESSED	This event occurs when a key has been pressed
KEY_RELEASED	This event occurs when a key has been released

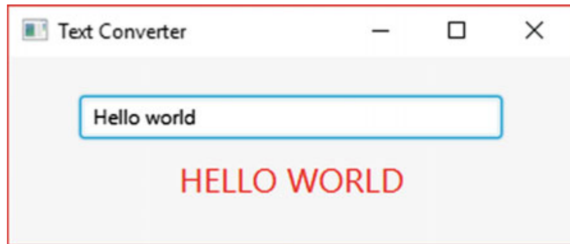


Fig. 16.2 The *TextConverter* application

Here is the code for the class:

TextConverter

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class TextConverter extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure a text field for user entry
        TextField textField = new TextField();
        textField.setMaxWidth(250);

        // create and configure a label to display the output
        Label label= new Label();
        label.setTextFill(Color.RED);
        label.setFont(Font.font("Ariel", 20));

        // display the contents of textField in upper case
        textField.setOnKeyReleased(e -> label.setText(textField.getText().toUpperCase()));

        // create and configure a VBox to hold the components
        VBox root = new VBox();
        root.setSpacing(10);
        root.setAlignment(Pos.CENTER);

        //add the components to the VBox
        root.getChildren().addAll(textField, label);


        // create a new scene
        Scene scene = new Scene(root);

        //add the scene to the stage, then configure the stage and make it visible
        stage.setScene(scene);
        stage.setTitle("Text Converter");
        stage.setHeight(150);
        stage.setWidth(350);
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }

}
```

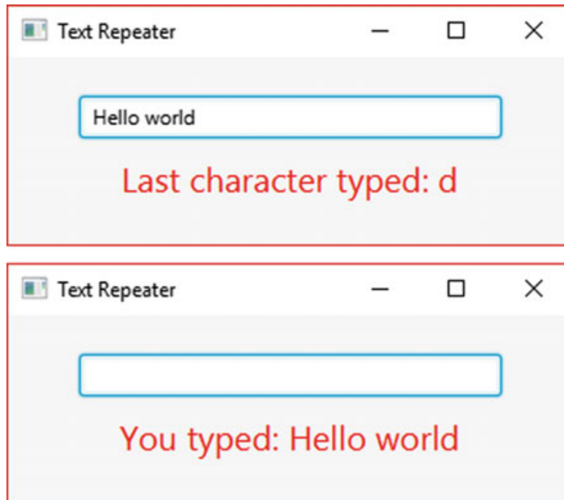


Fig. 16.3 The *TextRepeater* application

The only thing we need to draw your attention to is this line:

```
textField.setOnKeyReleased(e -> label.setText(textField.getText().toUpperCase()));
```

We are using the convenience method `setOnKeyReleased` to program the response to a key stroke. The event is triggered when the key is released (as opposed to when it is pressed), and as you can see from the lambda expression, after each key press the entire content of the text field is copied to the label and converted to upper case.

Our next program—the `TextRepeater`—is slightly different. In this application, as soon as character is typed it is echoed on the label at the bottom—however, if it is the <Enter> key that is pressed then the text field is cleared, and the entire phrase that was typed is displayed. You can see how this works in Fig. 16.3.

The code for this application appears below:

TextRepeater

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class TextRepeater extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create and configure a text field for user entry
        TextField textfield = new TextField();
        textfield.setMaxWidth(250);

        // create and configure a label to display the output
        Label repeatLabel= new Label();
        repeatLabel.setTextFill(Color.RED);
        repeatLabel.setFont(Font.font("Ariel", 20));


        textfield.setOnKeyTyped(e ->
                        {
                              if(e.getCharacter().equals("\r")) // check for the Enter key
                              {
                                   repeatLabel.setText("You typed: " + textfield.getText());
                                   textfield.setText("");
                              }
                              else
                              {
                                   repeatLabel.setText("Last character typed: " + e.getCharacter());
                              }
                        });

        // create and configure a VBox to hold our components
        VBox root = new VBox();
        root.setSpacing(10);
        root.setAlignment(Pos.CENTER);

        //add the components to the VBox
        root.getChildren().addAll(textfield, repeatLabel);


        // create a new scene
        Scene scene = new Scene(root);

        //add the scene to the stage, then configure the stage and make it visible
        stage.setScene(scene);
        stage.setTitle("Text Repeater");
        stage.setHeight(150);
        stage.setWidth(350);
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }

}
```

Once again, the only thing we need to discuss is how we code the convenience method, in this case `setOnKeyTyped`:

```
textfield.setOnKeyTyped(e ->
                       {
                             if(e.getCharacter().equals("\r")) // check for the Enter key
                             {
                                  repeatLabel.setText("You typed: " + textfield.getText());
                                  textfield.setText("");
                             }
                             else
                             {
                                  repeatLabel.setText("Last character typed: " + e.getCharacter());
                             }
                       });
```

You can see that the lambda expression deals with the possibility that the <Enter> key has been pressed. In order to do this we have made use of the `getCharacter` method of `e`, the `KeyEvent` parameter that is sent into the `handle` method of `EventHandler` when the event concerned is a key event. After a `KEY_TYPED` event, the `getCharacter` method returns a string which holds the value of the character returned. We have checked to see if this is the special character "`\r`" which represents the <Enter> key (Unicode 13). If it was the <Enter> key, the entire string is copied from the text field to the label (appended to the message "You typed:"), and the text field is cleared.

If the key pressed was any other key, the character is echoed on the label, again appended to a message.

16.3 Binding Properties

When we refer to the classes in the JavaFX package, we tend not to talk about the attributes of a class, but rather its **properties**.

In order to understand why we do this, let's consider the JavaFX `TextField` class for a moment. You have seen how we use the `setText` and `getText` methods of this class—these methods accept and return `String`s, and you would therefore be forgiven for assuming that the `TextField` class has a `text` attribute which is of type `String`. But you would be wrong. The attribute in question (which is inherited from a higher level class) is actually of type `StringProperty`. This class, along with many other similar classes such as `DoubleProperty`, `IntegerProperty`, `BooleanProperty` and so on, is a wrapper class —similar to classes such as `Double` and `Integer` that you came across in the first semester.

Methods of control classes such as `TextField` hide the details of its properties because they are set up to deal with the more familiar types such as `String` and **`double`**. The interesting thing about these property classes is that they have some useful methods. Some of these methods enable you to add event handlers to a property rather than an object itself—you will see this in action in the next section. But two particularly interesting methods are `bind` and `bindBidirectional`.

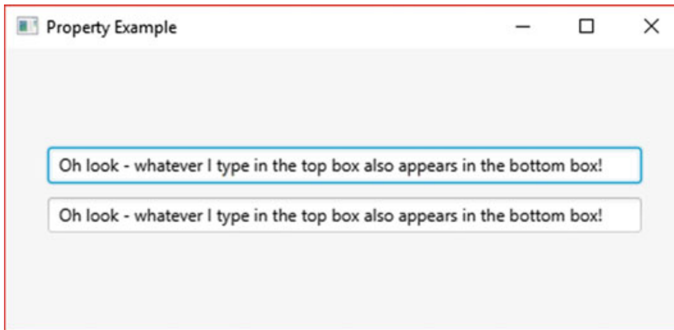


Fig. 16.4 Binding properties

These allow us to bind the properties of two different objects so that they act in unison.

To demonstrate this we have created a very simple application indeed, consisting essentially of two `textFields`. We have bound the text property of each of these with the result that the bottom one is frozen, while whatever is typed in the top one is applied immediately to the bottom one.

You can see this in Fig. 16.4.

The code for this class is shown below:

PropertyExample

```
import javafx.application.Application;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.geometry.Pos;
import javafx.scene.Scene;

public class PropertyExample extends Application
{
    @Override
    public void start(Stage stage)
    {
        TextField top = new TextField();
        TextField bottom = new TextField();
        top.setMaxWidth(420);
        bottom.setMaxWidth(420);

        // bind the text proerty of the bottom text field to that of the top
        bottom.textProperty().bind(top.textProperty());

        VBox root = new VBox(10);
        root.getChildren().addAll(top, bottom);
        root.setAlignment(Pos.CENTER);
        Scene scene = new Scene(root, 480, 200);
        stage.setScene(scene);
        stage.setTitle("Property Example");
        stage.show();
    }

     public static void main(String[] args)
     {
         launch(args);
     }
}
```

All that we are interested in here is the line of code that binds the properties:

```
bottom.textProperty().bind(top.textProperty());
```

You have seen previously that the `getText` method of `TextField` returns a `String`. However, the `text` property itself is retrieved by using the method `textProperty` (inherited from the parent class). This is the general pattern for properties of JavaFX classes: for example, you will see below that a `Slider` class has a property called `value` of type `DoubleProperty`; this is retrieved with the `valueProperty` method.

As we have said, properties have a method called `bind`. We have used this in our example, calling the `bind` method of the `text` property of the bottom field. The property that is sent into this `bind` method—in our case the `text` property of the top field—is the one that can change and whose changes are mirrored in the bound property.

We could have replaced the above line of code with this:

```
bottom.textProperty().bindBidirectional(top.textProperty());
```

Using the `bindBidirectional` method means that we can change either property and it is mirrored in the other one.

We could, of course, "chain" properties so that we could have three or more properties working in unison. This is left for you to try in the end of chapter exercises.

16.4 The *Slider* Class

A slider allows us to control the value of a variable by moving a sliding bar which is used to vary the value within a particular range. The application we have developed in Fig. 16.5 shows two sliders, one horizontal and one vertical. The current value of the movable "thumb" on each slider is displayed below the slider.

The complete code is shown below. Once you have had a look at it we will go through it with you.

SliderDemo

```
import java.text.DecimalFormat;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
public class SliderDemo extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double horizSliderWidth = 300;
        final double vertSliderHeight = 300;

        // numbers will be formatted to one decimal place
        DecimalFormat df = new DecimalFormat("0.0");

        // create and configure the vertical slider
        Slider vertSlider = new Slider(0, 20, 0);
        vertSlider.setMinHeight(vertSliderHeight);
        vertSlider.setShowTickMarks(true);
        vertSlider.setShowTickLabels(true);
        vertSlider.setSnapToTicks(true);
        vertSlider.setMajorTickUnit(5.0);
        vertSlider.setMinorTickCount(10);
        vertSlider.setOrientation(Orientation.VERTICAL); // default is horizontal

        // create and configure the horizontal slider
        Slider horizSlider = new Slider(0, 10, 0);
        horizSlider.setMinWidth(horizSliderWidth);
        horizSlider.setShowTickMarks(true);
        horizSlider.setShowTickLabels(true);
        horizSlider.setSnapToTicks(true);
        horizSlider.setMajorTickUnit(1.0);
        horizSlider.setMinorTickCount(4);

        // create two lables to keep track of each slider position
        Label horizLabel = new Label("Current value is 0.0");
        Label vertLabel = new Label("Current value is 0.0");

        // add a listener to the vertical slider
        vertSlider.valueProperty().addListener((observable, oldValue, newValue)  ->
                                   vertLabel.setText("Current value is " + df.format(newValue)));

        // add a listener to the horizonal slider
        horizSlider.valueProperty().addListener((obsValue, oldValue, newValue)  ->
                                   horizLabel.setText("Current value is " + df.format(newValue)));

        // create and configure a VBox to hold the vertical slider and label
        VBox vertBox = new VBox(10);
        vertBox.setAlignment(Pos.BOTTOM_LEFT);
        vertBox.setMinWidth(horizSliderWidth/3);
        vertBox.getChildren().addAll(vertSlider, vertLabel);

        // create and configure a VBox to hold the horizontal slider and label
        VBox horizBox = new VBox(10);
        horizBox.setAlignment(Pos.BOTTOM_LEFT);
        horizBox.getChildren().addAll(horizSlider, horizLabel);

        // create and configure an HBox as root
        HBox root = new HBox(30);
        root.setPadding(new Insets(10, 10, 10, 10));
        root.getChildren().addAll(horizBox, vertBox);

        // create and configure the scene and stage
        Scene scene = new Scene(root, 460, 350);
        stage.setScene(scene);
        stage.setTitle("Slider Example");
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

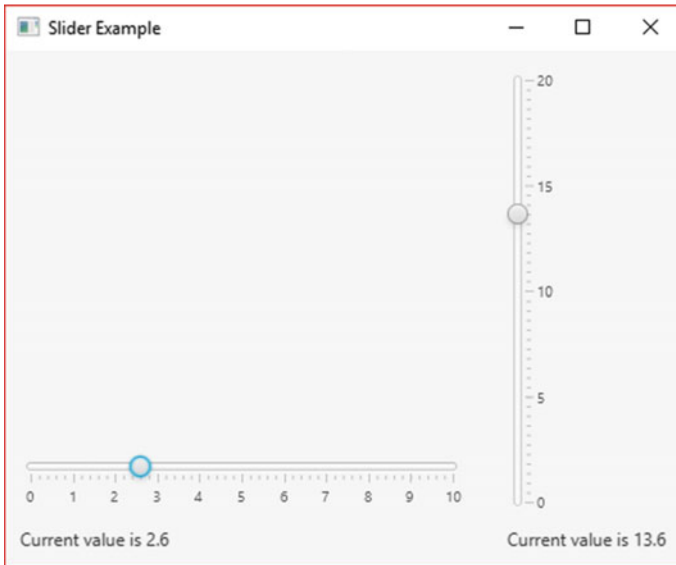


Fig. 16.5 A horizontal and a vertical slider

After declaring some constants for the preferred width and height of the sliders, and defining a decimal format for the values, we create our sliders. First the vertical slider:

```
Slider vertSlider = new Slider(0, 20, 0);
vertSlider.setMinHeight(vertSliderHeight);
vertSlider.setShowTickMarks(true);
vertSlider.setShowTickLabels(true);
vertSlider.setSnapToTicks(true);
vertSlider.setMajorTickUnit(5.0);
vertSlider.setMinorTickCount(10);
vertSlider.setOrientation(Orientation.VERTICAL);
```

The constructor takes three **double** values, which represent, respectively, the minimum, maximum and initial values of the movable thumb. In our case the slider will have a range of zero to 20, and start at zero.

The other methods that you see should be self-explanatory—there are still more methods that you can explore, and as usual the best approach is to experiment with these for yourself.

The horizontal slider is created and configured in the same way, but in this case it is not necessary to specify the orientation, as horizontal is the default.

We go on to create the labels where the values will be displayed;

```
Label horizLabel = new Label("Current value is 0.0");
Label vertLabel = new Label("Current value is 0.0");
```

Next we define what happens when the value of the slider is moved, starting with the vertical slider. In the case of a slider we add the listener not to the slider itself but to one of its properties, `valueProperty`, which is of type `DoubleProperty`, and which holds the current position of the movable thumb:

```
vertSlider.valueProperty().addListener((observable, oldValue, newValue)  ->
                                vertLabel.setText("Current value is " + df.format(newValue)));
```

We are not able to use a convenience method for this, but instead use the `addListener` method. The kind of listener we are adding is a `ChangeListener`, which has an abstract method named `changed`. You can see that we are using a lambda expression for this purpose. The JavaFX documentation tells us that the `changed` method is specified as follows:

```
changed(ObservableValue<? extends T> observable, T oldValue, T newValue)
```

When the value of the slider position changes, the `changed` method is called. It receives three parameters. The first of these represents the current value, and is of type `ObservableValue`—this is yet another wrapper class. As you can see it is a generic class. You will notice that the method is specified using a wildcard so that it will take objects of a particular type or subtypes of that type. In the case of a slider it will handle values of type `Double`. The next two parameters are the old value before it was changed, and the new value after the change.

Fortunately, in our lambda expression, we do not have to worry about specifying the types of the parameters, because the compiler does this for us—another example of the very useful ability of Java compilers to utilize type inference.

You can see that in our lambda expression we have used the new value parameter to display the position of the slider on the correct label.

The behaviour of the horizontal slider is specified in exactly the same way.

The rest of the code is all to do with placing our sliders on the scene. There is not too much to say about this because there is nothing new here—notice however that the box holding the vertical slider has been specified to be one-third the width of the box containing the horizontal slider.

16.5 Multimedia Nodes

It is very common that we want to embed such things as images or videos into our applications. Here we will briefly explore three classes that help us to do this—`ImageView`, `MediaView` and `WebView`.

16.5.1 Embedding Images

We will begin by looking at `ImageView`, a class that, as its name suggests, allows us to embed images. In Fig. 16.6 we see a very simple application in which an image is embedded into a scene graphic.

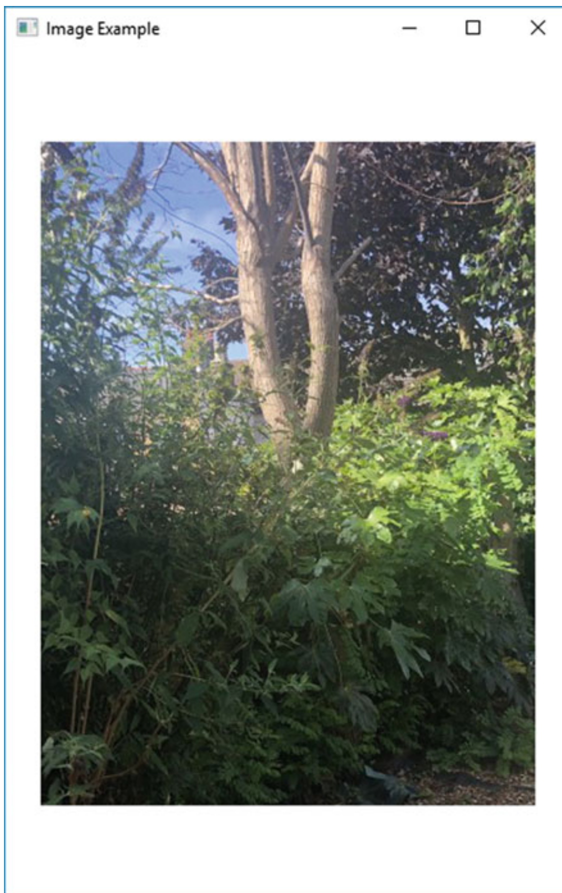


Fig. 16.6 Embedding an image

The application code below shows how we did this:

ImageHolder

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.FlowPane;


public class ImageHolder extends Application
{
    @Override
    public void start(Stage stage)
    {
       Image image = new Image("Trees.jpg");  // create an image from a file
       ImageView imageView = new ImageView(image); // wrap the image in an ImageView node

       FlowPane root = new FlowPane();
       root.setAlignment(Pos.CENTER);
       root.getChildren().add(imageView); // add the ImageView object to the container

       Scene scene = new Scene(root, 400, 600);
       stage.setScene(scene);
       stage.setTitle("Image Example");

       stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

You can see that we created an object of the `Image` class from a file, `Trees.jpg`.[1]

```
Image image = new Image("Trees.jpg");
```

In order to be able to add this image to a container, we need create an object of the `ImageView` class:

```
ImageView imageView = new ImageView(image);
```

`ImageView` extends `Node`, and therefore, once we have wrapped it around an image, it can be added to a container (in this case a `FlowPane`) in the usual way:

```
root.getChildren().add(imageView);
```

Figure 16.7 shows a variation of this application.

[1]The program needs to have access to this file when it is run—normally it would be in the same directory as the `main` class.

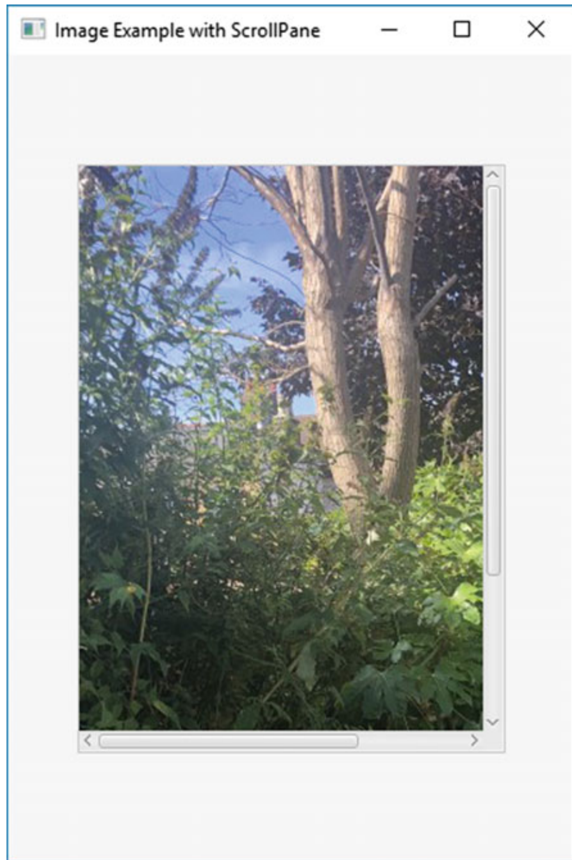


Fig. 16.7 Embedding an image with a scroll pane

In this case, we have added our `imageView` object to a scroll pane, which we have then configured with a particular width and height:

```
ScrollPane scrollPane = new ScrollPane(imageView);
scrollPane.setPrefViewportWidth(250);
scrollPane.setPrefViewportHeight(350);
```

Then, instead of adding the `imageView` object directly to the `FlowPane`, we have added the `ScrollPane` object instead:

```
root.getChildren().add(scrollPane);
```

We can also add images to controls such as buttons. Figure 16.8 shows an image, a simple orange square, embedded in a button. In the first version the button was created without a caption, in the second with the caption “Quit”.

Once the button was created, the image was added as follows:

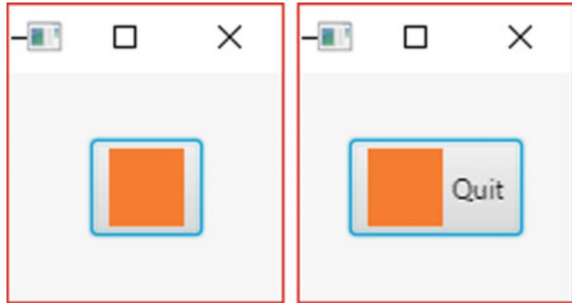


Fig. 16.8 Adding an image to a button

```
Image image = new Image("OrangeSquare.png");
ImageView imageView = new ImageView(image);
button.setGraphic(imageView);
```

16.5.2 Embedding Videos

In the case of embedding a video there are some additional concepts that we need to show you, but the process is similar. We create a `Media` object from a file from which we then create a `MediaPlayer` object. This is then wrapped in a `MediaView` object. `MediaView` is an extension of `Node`, so can be added to a container. The video formats that are supported are MPEG-4 and FLV. MP3 audio is also supported.

Our application is shown in Fig. 16.9.

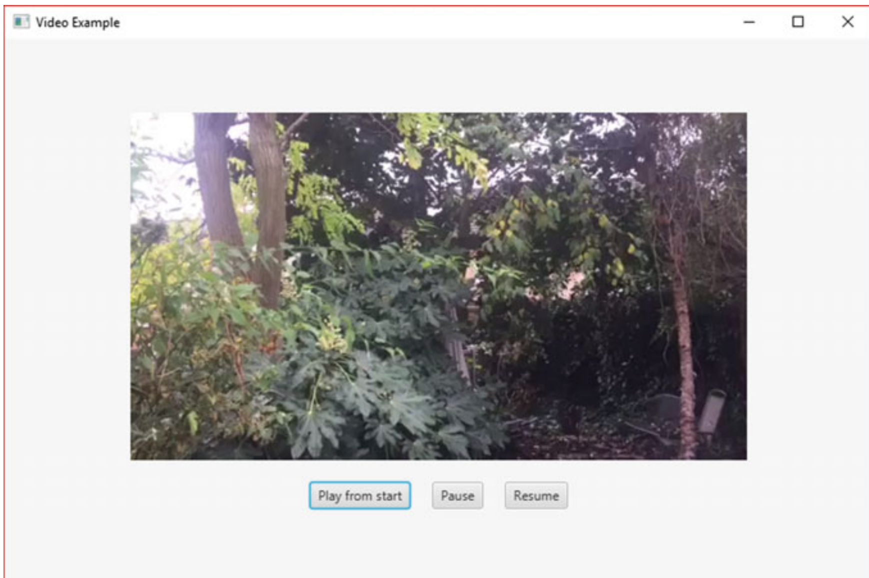


Fig. 16.9 Embedding a video

As you can see, we have provided options for playing the video from the start as well as pausing and resuming. The code below shows how we have achieved this:

VideoPlayer

```
import java.io.File;
import javafx.util.Duration;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;


public class VideoPlayer extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create a File object from the video file
        File file = new File("Trees.mp4");

        // create a Media object from the File Object
        Media media = new Media(file.toURI().toString());

        // create a MediaPlayer object from the Media Object
        MediaPlayer mp = new MediaPlayer(media);

        // create a MediaView object from the MediaPlayer Object
        MediaView mv = new MediaView(mp);

        // create the buttons
        Button playFromStartButton = new Button("Play from start");
        Button pauseButton = new Button("Pause");
        Button resumeButton = new Button("Resume");

        // create and configure an HBox to hold the buttons
        HBox buttonBox = new HBox(20);
        buttonBox.setAlignment(Pos.CENTER);
        buttonBox.getChildren().addAll(playFromStartButton, pauseButton, resumeButton);

        // add event handlers to the buttons
        playFromStartButton.setOnAction(e -> {
                                        mp.seek(Duration.millis(0));
                                        mp.play();
                                       }
                                );

        pauseButton.setOnAction(e -> mp.pause());

        resumeButton.setOnAction(e-> mp.play());

        // create the root container, and add it to the scene and stage
        VBox root = new VBox(20);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(mv, buttonBox);


        Scene scene = new Scene(root,800,500);
        stage.setScene(scene);
        stage.setTitle("Video Example");

        stage.show();

    }

        public static void main(String[] args)
    {
        launch(args);
    }

}
```

The Media object that we need to create requires an absolute reference to the source file. This is referred to as a **Uniform Resource Identifier (URI)**. In order to provide this we need first to create an instance of the File class which is provided in the java.io package.

```
File file = new File("Trees.mp4");
```

File has a method called toURI which retrieves the file's URI and allows us to send a String representation to the new Media object in the following way:

```
Media media = new Media(file.toURI().toString());
```

Now we are able to create our MediaPlayer object and hence our Media-View object

```
MediaPlayer mp = new MediaPlayer(media);
MediaView mv = new MediaView(mp);
```

Next we create our buttons and add them to an HBox, which will later be added to a VBox along with the MediaView object. Having done that, we need to add event handlers to hold the code that must be executed when the buttons are pressed.

The MediaPlayer class has a progress counter to keep track of the progress of the video. This value is held in an object of the Duration class, which has methods toMillis, toSeconds, toMinutes and toHours, which express the duration in milliseconds, seconds, minutes and hours respectively. It also has **static** methods called millis and minutes, each of which accepts a **double** and returns a Duration object representing the specified number of milliseconds or minutes respectively.

MediaPlayer has a method called seek which accepts a Duration object and moves the progress counter to the specified point. It also has a pause method and a play method, the second of which plays the video from the point indicated by the progress counter (so it resumes after a pause).

With this information in mind we can look at the code for the "Play from Start" button.

```
playFromStartButton.setOnAction(e -> {
                                          mp.seek(Duration.millis(0));
                                          mp.play();
                                     }
                                );
```

You can see how we use the `seek` method to set the counter back to the beginning, and then call the `play` method.

The "pause" and "resume" buttons simply invoke `pause` and `play` respectively.

16.5.3 Embedding Web Pages

To embed a webpage we need to create a WebView object that we can add to a container. This class incorporates a `WebEngine`, which is a class that is capable of managing web pages.

In Fig. 16.10 you can see our simple application. It provides a field for the user to type in the website address (the URL), and once the <Enter> key has been pressed the web page is retrieved and displayed.

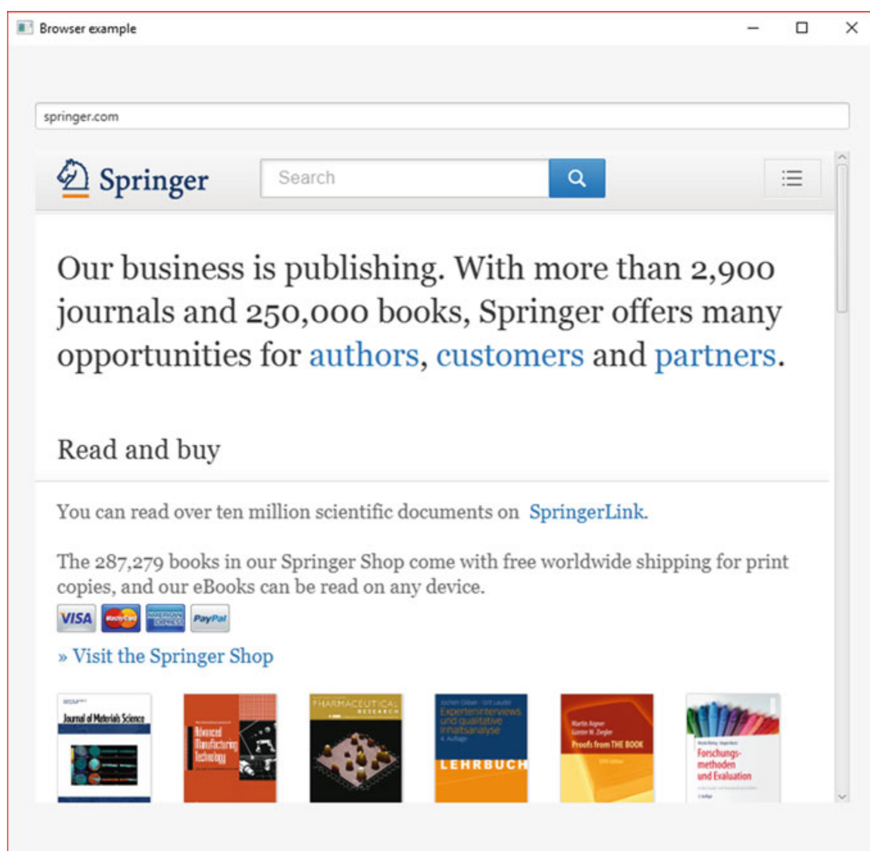


Fig. 16.10 Embedding a web page

Here is the code:

WebBrowser

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class WebBrowser extends Application
{
  @Override
  public void start(Stage stage)
  {
        // create and configure a WebView node
        WebView wv = new WebView();
        wv.setMaxSize(750, 700);

        // create the text field where the URL will be entered
        TextField entry = new TextField();
        entry.setMaxWidth(750);

        /* define the behaviour that occurs when the URL has been entered and
           the Enter key has been pressed */
        entry.setOnKeyTyped(e -> {
                              String url;
                              if(e.getCharacter().equals("\r"))
                              {
                                        url = entry.getText();
                                        if(!url.startsWith("http"))
                                        {
                                            url = "http://" + url;
                                        }
                                        wv.getEngine().load(url);
                         }

                   });


        VBox root = new VBox(20);
        root.setAlignment(Pos.CENTER);

        root.setMaxSize(750,700);
        root.getChildren().addAll(entry, wv);

        Scene scene = new Scene(root, 800, 750);

        stage.setScene(scene);
        stage.setTitle("Browser example");
        stage.show();
  }

  public static void main(String[] args)
  {
        launch(args);
  }
}
```

You can see that having created both the `WebView` and the `TextField`, we go on to add an event handler to the `TextField` in order to provide the code that will be executed when the URL is entered:

```
entry.setOnKeyTyped(e -> {
                        String url;
                        if(e.getCharacter().equals("\r"))
                        {
                                url = entry.getText();
                                if(!url.startsWith("http"))
                                {
                                   url = "http://" + url;
                                }
                                wv.getEngine().load(url);
                    }

               });
```

After declaring a variable to hold the URL, we check to see if the <Enter> key has been pressed (special character "`\r`"). We then check to see if the string entered starts with "http". This is because the method we are going to use to load the page requires the full URL, which includes the protocol—normally this would be "http" or "https". Usually people leave this out went entering a URL, and let the browser deal with it. That is what we have done here—if the protocol has not been included we prefix the address with "http://".

Next we retrieve the engine from the `WebView` object with the `getEngine` method, and use the `load` method of this object to load the URL.

16.6 Cascading Style Sheets

Those of you who have had experience in website development will certainly be familiar with cascading style sheets (CSS). These enable a webpage, written in HTML, to be free from any formatting detail. The HTML code is on the whole restricted to providing the basic components and functionality, whereas the appearance of the page is placed in a separate file with the extension `.css`.

JavaFX provides a similar capability, although as we shall see the syntax is slightly different, with the properties requiring the prefix `-fx-`.

This topic of style sheets is in fact a vast one, and all that it is possible to do here is to whet your appetite by explaining the basic principles, which will enable you to go on and study more detail if you are interested.

In Fig. 16.11 we have created a simple registration form (the buttons have not been activated), but we have left out any formatting information, so the application is simply formatted according to the JavaFX defaults.

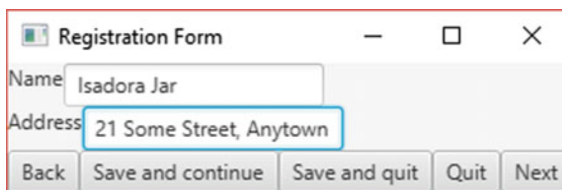


Fig. 16.11 An unformatted application

Take a look at the code:

CSSDemo

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CSSDemo extends Application
{
  @Override
  public void start(final Stage stage)
  {
      HBox row1 = new HBox();
      HBox row2 = new HBox();
      HBox row3 = new HBox();

      Label nameLabel = new Label("Name");
      TextField nameField = new TextField();

      Label addressLabel = new Label("Address");
      TextField addressField = new TextField();

      Button backButton = new Button("Back");
      Button saveAndContinueButton = new Button("Save and continue");
      Button saveAndQuitButton = new Button("Save and quit");
      Button quitButton = new Button("Quit");
      Button nextButton = new Button("Next");

      row1.getChildren().addAll(nameLabel, nameField);
      row2.getChildren().addAll(addressLabel, addressField);
      row3.getChildren().addAll(backButton, saveAndContinueButton,
                                               saveAndQuitButton, quitButton, nextButton);

      VBox root = new VBox();
      root.getChildren().addAll(row1, row2, row3);

      Scene scene = new Scene(root);
      stage.setScene(scene);
      stage.setTitle("Registration Form");
      stage.show();
  }

  public static void main(String[] args)
  {
        launch(args);
  }
 }
```

As you can see, without the formatting information the code looks very uncluttered and easy to read. This is not only something that is very useful when developing and maintaining an application, but it also has other important benefits. By placing the formatting information in a separate file it means that we can provide a similar look and feel—a corporate look—across different applications, by using the same style sheet. It also means that if we want to change the appearance of the application we can simply replace one style sheet with another. We often use the word **skin** to refer to a particular look that an application has—so to replace one skin with another, we simply change the style sheet associated with the application.

So, we added a style sheet to the above code. The result is shown in Fig. 16.12.

Fig. 16.12 The registration form with a style sheet

We called this style sheet `Example.css`. To load it into our application it is necessary to add the following line of code:

```
scene.getStylesheets().add("Example.css");
```

The file needs to be located in the same directory as the `main` class.
The CSS file looks like this:

Example.css

```
.root
{
    -fx-font-size: 16pt;
    -fx-font-family: "Calibri";
    -fx-base: #add8e6; /* lightblue */
    -fx-spacing: 20px;
    -fx-padding: 20px;
}

.button
{
    -fx-text-fill: #008000;  /* green */
    -fx-font-size: 12pt;
}

.label
{
    -fx-background-color: #ccccff ;
    -fx-min-width : 80px;
    -fx-min-height: 42px;
    -fx-font-weight: bold;
    -fx-padding: 4px;
}

.text-field
{
    -fx-min-width: 380px;
    -fx-text-fill: #1100cc;
    -fx-background-color: #ccffcc ;
    -fx-font-style: italic;

}

.button1
{
    -fx-text-fill: #ff00ff; /* magenta */
    -fx-background-color: #ffcc99;
}

#quit
{
    -fx-text-fill: red;
    -fx-background-color: #ffff00; /* yellow */
}

#row3
{

    -fx-spacing: 10px;
    -fx-border-color: #000000; /* black */
    -fx-border-width: 2px;
    -fx-padding: 5px;

}
```

The style sheet consists of a number of **styles** or **selectors**, introduced by a full-stop or a hash. These styles refer to particular components. Many style names are provided by the system and are the same or similar to the node to which they refer: for example button, label and text-field (note the hyphen—style names tend to have hyphens when they consist of two words joined together). Styles corresponding to classes are referred to as class styles. Styles have **properties**, which refer to the properties of the particular node. The properties are set by rules which are placed between braces.

Let's take a look at the first style in our style sheet:

```
.root
{
    -fx-font-size: 16pt;
    -fx-font-family: "Calibri";
    -fx-base: #add8e6; /* lightblue */
    -fx-spacing: 20px;
    -fx-padding: 20px;
}
```

As you can see, class styles are introduced by a full-stop. Names of properties all begin with `-fx-`.

`root` is a style that refers to the root node in the scene, and all descendant nodes will have this style unless these definitions are overridden.

The first two lines set the size of the font (in points), and the font family.

The next line sets the base colour of the node. The colour here is expressed as a hexadecimal number, which corresponds to the RGB values we explained in Sect. 16.3. The first two digits (starting at the left) represent red, the next two green and the final two blue. So here we have red with an intensity of 173 (AD in hexadecimal), green with 216 (D8 in hex), and blue with 230 (E6 in hex). This actually corresponds to the pre-defined colour `lightblue`, and we could have used this constant instead. We could also have written `rgb(173, 216, 230).`

The last two lines set the spacing and padding, measured in pixels (`px`). We could have used `cm` or `in` (centimetres or inches), or we could have used `em`. An `em` is the size of the font that is currently in use—so here, 1 `em` would represent 16`pt`.

We go on to set the style properties for the `.button` class, the `.label` class and the `.text-field` class. These properties will take precedence over any properties set in the `.root` class.

After this we see the following:

```
.button1
{
    -fx-text-fill: #ff00ff; /* magenta */
    -fx-background-color: #ffcc99;
}
```

What we have done here is to set the properties for our own named style `button1`. This can then be applied to components of our choice. We set this style specifically for our "back" and "next" buttons. We apply this style to those components with the following code:

```
backButton.getStyleClass().add("button1");
nextButton.getStyleClass().add("button1");
```

Finally we see two styles introduced not with a full stop, but with a hash (#):

```
#quit
{
    -fx-text-fill: red;
    -fx-background-color: #ffff00; /* yellow */
}

#row3
{

    -fx-spacing: 10px;
    -fx-border-color: #000000; /* black */
    -fx-border-width: 2px;
    -fx-padding: 5px;
}
```

A hash introduces a style for a particular named component. In our code we gave two components an id, with the `setId` method:

```
row3.setId("row3");
quitButton.setId("quit");
```

Once we have assigned an id to a node we can then set its style with the hash as shown.

Just to summarise, the additional lines of code that we needed to include in our program were as follows:

```
scene.getStylesheets().add("Example.css");
backButton.getStyleClass().add("button1");
nextButton.getStyleClass().add("button1");
row3.setId("row3");
quitButton.setId("quit");
```

As you can imagine, we have only scratched the surface of what is available when using style sheets, but we hope it will be enough for you to do some additional reading and experiments, and of course to check out the Oracle™ website for further help and information. Those of you who are interested in JavaFX style sheets will get an opportunity to apply them as part of an end of chapter programming exercise in the advanced case study of Chap. 21.

16.7 Self-test Questions

1. Distinguish between a `MouseEvent` and a `KeyEvent`.
2. The following application draws a small rectangle on a scene graphic.

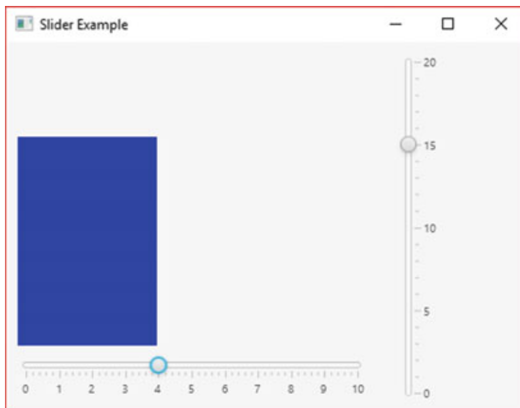
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.geometry.Pos;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

public class DrawRectangle extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 400;

        Rectangle rect = new Rectangle(10, 10);
        rect.setFill(Color.RED);

        VBox root = new VBox();
        root.getChildren().add(rect);
        root.setAlignment(Pos.TOP_LEFT);

        Scene scene = new Scene(root, WIDTH, HEIGHT);

        // method goes here

        stage.setScene(scene);
        stage.setTitle("Draw Rectangle");
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

   Replace the comment in the program with some code that would allow the user to change the width and height of the rectangle by dragging the mouse. You should look at the `RedCircle` class to get some ideas for how to go about this.

3. Explain the term *property* in the context of JavaFX components.
4. Explain what is meant by *binding* properties, and describe how this could be achieved in the case of a slider.
5. Describe how different types of *multi-media nodes* that can be incorporated into your JavaFX applications.
6. Explain the reasons for using *cascading style sheets* to separate formatting information from the rest of the application.

16.8 Programming Exercises

1. Implement a few of the programs that we have developed in this chapter, and experiment with different settings in order to change some the features.
2. Adapt the PropertyExample application of Sect. 16.3 so that three or more TextFields operate in unison.
3. Implement the program you adapted in question 2 of the self-test questions. Try this with other shapes such as circle and ellipse.
4. In the RedCircle class of Sect. 16.2.1 we used convenience methods to program responses to a MOUSE_PRESSED event, a MOUSE_RELEASED event, a MOUSE_MOVED event and a MOUSE_DRAGGED event.
   Experiment with a the MOUSE_ENTERED and MOUSE_EXITED events. One idea might be to change the colour of a button when the cursor is moved over it.
5. Adapt the SliderDemo from Sect. 16.4 so that instead of printing an integer value, it draws an expanding rectangle as the slider is moved. This is demonstrated in the following diagram:



6. Adapt the SliderDemo program so that the properties of the two sliders are bound, and when one slider moves, the other moves accordingly.
7. Design your own skin for the registration form in Sect. 16.6 by creating a new style sheet.

Check for updates

JavaFX: Interacting with the User 17

Outcomes:

By the end of this chapter you should be able to:

- *create applications that utilize pull-down* ***menus*** *and* ***context menus***;
- *create a* ***modal*** *dialogue by defining a secondary stage*;
- *write applications that offer choices* via ***combo boxes***, ***check boxes*** *and* ***radio buttons***;
- *use a stack pane to create a* ***card menu***;
- *enable user interaction by using the subclasses of the* `Dialog` *class*.

17.1 Introduction

One of the most common ways for an application to interact with the user is to provide a number of choices—just as we did with the text-based menus that we introduced you to in the first semester. With graphical applications, there are a number of ways of doing this, and in this chapter we present you with a variety of techniques you can use—we will show you how to create drop-down menus, pop-up menus (also called context menus), check boxes, radio buttons and combo boxes. We will also show you how to interact with the user via popup dialogue windows.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_17

17.2 Drop-Down Menus

A drop-down menu, or pull-down menu, is a very common way to offer choices to the user of a program. In Fig. 17.1. we see a very simple example. The program displays a flag consisting of three horizontal stripes—the colour of each stripe can be changed by means of the pull-down menus on the top bar.

As an example, Fig. 17.2 shows the choices offered by the first menu:

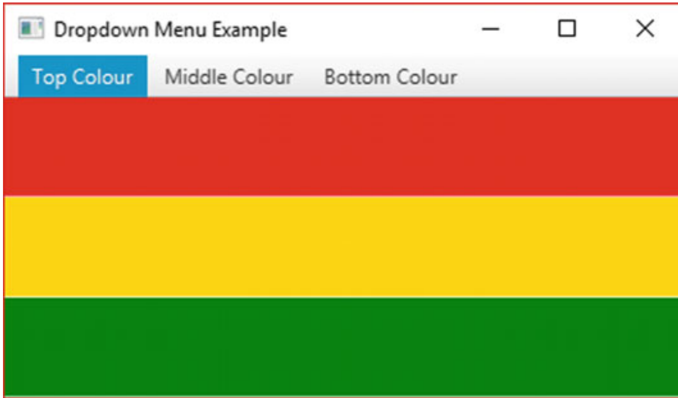


Fig. 17.1 An application with three pull-down menus

Fig. 17.2 The options on the "Top Colour" menu

The code for the `Flag` class is presented below:

Flag

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.layout.Background;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.geometry.Pos;

public class Flag extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 200;

        // create and configure a menu bar
        MenuBar bar = new MenuBar();
        bar.setMinHeight(25);

        // create drop-down menus
        Menu topStripeMenu = new Menu("Top Colour");
        Menu middleStripeMenu = new Menu("Middle Colour");
        Menu bottomStripeMenu = new Menu("Bottom Colour");

        // add the drop-down menus to the menu bar
        bar.getMenus().addAll(topStripeMenu, middleStripeMenu, bottomStripeMenu);

        // create menu items - two for each drop-down menu
        Menu red = new Menu("Red");
        Menu blue = new Menu("Blue");

        Menu gold = new Menu("Gold");
        Menu orange = new Menu("Orange");

        Menu green = new Menu("Green");
        Menu purple = new Menu("Purple");

        // add menu items to drop-down menus
        topStripeMenu.getItems().addAll(red, blue);
        middleStripeMenu.getItems().addAll(gold, orange);
        bottomStripeMenu.getItems().addAll(green, purple);


        // create the stripes
        Rectangle topStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);
        Rectangle middleStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);
        Rectangle bottomStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);

        // set initial colours
        topStripe.setFill(Color.RED);
        middleStripe.setFill(Color.GOLD);
        bottomStripe.setFill(Color.GREEN);

        // define the behaviour for each menu item
        red.setOnAction(e ->  topStripe.setFill(Color.RED));
        blue.setOnAction(e ->  topStripe.setFill(Color.BLUE));
        gold.setOnAction(e ->  middleStripe.setFill(Color.GOLD));
        orange.setOnAction(e ->  middleStripe.setFill(Color.ORANGE));
        green.setOnAction(e ->  bottomStripe.setFill(Color.GREEN));
        purple.setOnAction(e ->  bottomStripe.setFill(Color.PURPLE));

        // create VBox to hold the menu bar and the stripes
        VBox root = new VBox();
        root.setAlignment(Pos.TOP_LEFT);
        root.setBackground(Background.EMPTY);
        root.getChildren().addAll(bar, topStripe, middleStripe, bottomStripe);

        // create the scene and add the VBox
        Scene scene = new Scene(root, WIDTH, HEIGHT);

        // confgure the stage
        stage.setScene(scene);
        stage.setTitle("Dropdown Menu Example");
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

The first thing we do (after declaring constants for the width and height of the scene) is to create a menu bar:

```
MenuBar bar = new MenuBar();
bar.setMinHeight(25);
```

Next we create the three menus that will appear on the bar:

```
Menu topStripeMenu = new Menu("Top Colour");
Menu middleStripeMenu = new Menu("Middle Colour");
Menu bottomStripeMenu = new Menu("Bottom Colour");
```

Once you have created an instance of the `Menu` class, this object can hold other sub-menu items or can have an event handler attached to it which will enable it to respond to a mouse click (an `ActionEvent`).

In this case we will add two menu items for each stripe. First we create these items:

```
Menu red = new Menu("Red");
Menu blue = new Menu("Blue");

Menu gold = new Menu("Gold");
Menu orange = new Menu("Orange");

Menu green = new Menu("Green");
Menu purple = new Menu("Purple");
```

Then we add each one to the correct menu. The `getItems` method of the `Menu` class returns a list of items—and we add our items to this list with the `addAll` method.

```
topStripeMenu.getItems().addAll(red, blue);
middleStripeMenu.getItems().addAll(gold, orange);
bottomStripeMenu.getItems().addAll(green, purple);
```

The next thing is to declare and configure three rectangles for our stripes:

```
Rectangle topStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);
Rectangle middleStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);
Rectangle bottomStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);

topStripe.setFill(Color.RED);
middleStripe.setFill(Color.GOLD);
bottomStripe.setFill(Color.GREEN);
```

We have arranged it so that the height of each stripe takes up one-third of the total height, less the height of the menu bar.

Now we add the event handlers, using the convenience method `setOnAction` that we have seen before:

```
red.setOnAction(e ->  topStripe.setFill(Color.RED));
blue.setOnAction(e ->  topStripe.setFill(Color.BLUE));
gold.setOnAction(e ->  middleStripe.setFill(Color.GOLD));
orange.setOnAction(e ->  middleStripe.setFill(Color.ORANGE));
green.setOnAction(e ->  bottomStripe.setFill(Color.GREEN));
purple.setOnAction(e ->  bottomStripe.setFill(Color.PURPLE));
```

All that remains is to create and configure a `VBox` to which we add the menu bar and the three stripes; then we add this to the scene, and finally we add the scene to the stage as usual:

```
VBox root = new VBox();
root.setAlignment(Pos.TOP_LEFT);
root.setBackground(Background.EMPTY);
root.getChildren().addAll(bar, topStripe, middleStripe, bottomStripe);

Scene scene = new Scene(root, WIDTH, HEIGHT);

stage.setScene(scene);
stage.setTitle("Dropdown Menu Example");
stage.show();
```

17.3 Context (Pop-Up) Menus

An alternative to a pull-down menu is a context menu, also referred to as a pop-up menu. A context menu is normally not available all the time, but pops up only when it is necessary, and then disappears. To demonstrate this we have created an application in which the menu is used simply to change the background colour of the graphic, and is invoked by pressing a button. This is demonstrated in Fig. 17.3.

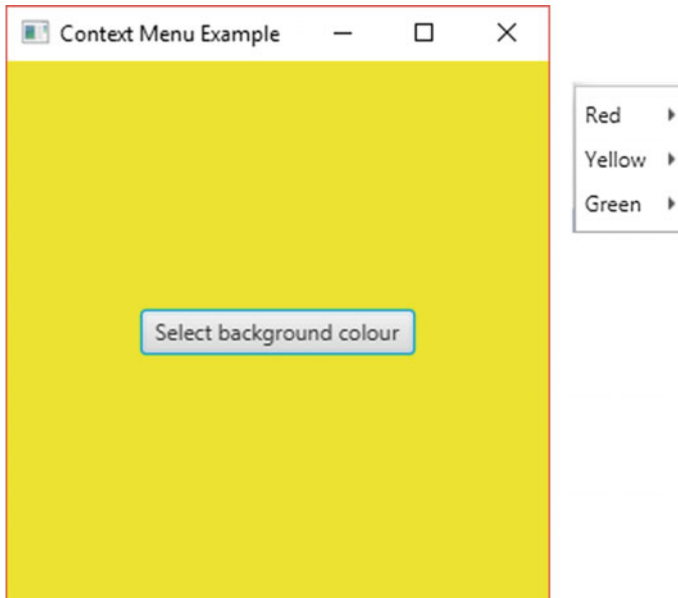


Fig. 17.3 A context menu

Here is the code for the `ContextMenuExample` class:

ContextMenuExample

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.geometry.Pos;
import javafx.geometry.Side;
import javafx.geometry.Insets;
import javafx.scene.control.Button;
import javafx.scene.control.ContextMenu;
import javafx.scene.control.Menu;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.FlowPane;

public class ContextMenuExample extends Application
{
   @Override
   public void start(Stage stage)
   {
     final double WIDTH = 300;
     final double HEIGHT = 300;

     // create a flow pane to be used as the root node
     FlowPane root = new FlowPane();
     root.setAlignment(Pos.CENTER);


     // create a button and add it to the flow pane
     Button button = new Button("Select background colour");
     root.getChildren().add(button);

     // create a context menu
     ContextMenu popup = new ContextMenu();

     // define the menu items
     Menu red = new Menu("Red");
     Menu yellow = new Menu("Yellow");
     Menu green = new Menu("Green");

     // add the menu items to the context menu
     popup.getItems().addAll(red, yellow, green);

     // add the event listeners: the background of the pane is changed and then the menu is closed
     red.setOnAction(e -> {
                              root.setBackground(new Background(new BackgroundFill(Color.RED,
                                                              CornerRadii.EMPTY, Insets.EMPTY)));
                              popup.hide();
                         }
                    );

     yellow.setOnAction(e -> {
                              root.setBackground(new Background(new BackgroundFill(Color.YELLOW,
                                                              CornerRadii.EMPTY, Insets.EMPTY)));
                              popup.hide();
                            }
                        );

     green.setOnAction(e ->  {
                              root.setBackground(new Background(new BackgroundFill(Color.GREEN,
                                                              CornerRadii.EMPTY, Insets.EMPTY)));
                              popup.hide();
                            }
                        );

     // add the event listener to the button: the menu is shown when the button is pressed
     button.setOnAction(e -> popup.show(root, Side.RIGHT, 10, 10));

     // configure the scene and the stage
     Scene scene = new Scene(root, WIDTH, HEIGHT);
     stage.setScene(scene);
     stage.setTitle("Context Menu Example");
     stage.show();
  }

  public static void main(String[] args)
  {
      launch(args);
  }
}
```

There is nothing very new here, and the code is mostly self-explanatory. But notice the code for the `setOnAction` method of the button:

```
button.setOnAction(e -> popup.show(root, Side.RIGHT, 10, 10));
```

The popup menu appears on the right side of the "anchor" node, offset by 10 pixels to the right and 10 from the top.

There is something else to notice about the menu that appears—it is **non-modal**. This means that the parent window is still accessible. This might be good for some applications, but in other applications you may want the application to be frozen until the context menu is hidden. In this case the menu is referred to as **modal**.

If we want a modal dialogue you can achieve this by creating a secondary stage. We have done this so that the popup menu has three buttons as shown in Fig. 17.4.

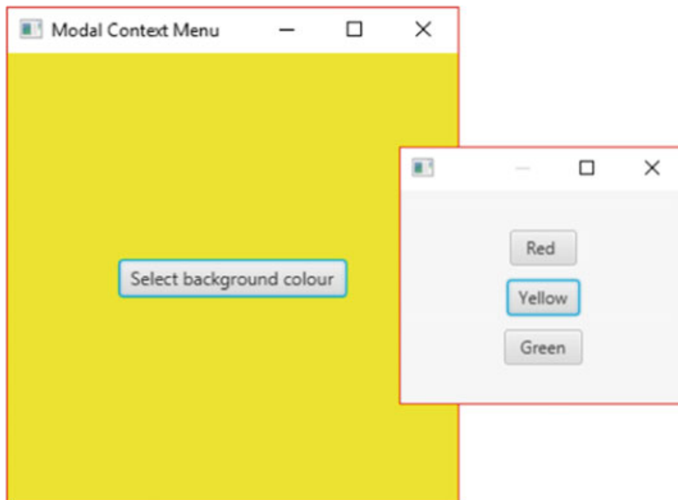


Fig. 17.4 A modal context menu

Here is the code:

ModalContextMenuExample

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Modality;
import javafx.stage.Stage;

public class ModalContextMenuExample extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        final double WIDTH = 300;
        final double HEIGHT = 300;

        // create a flow pane to be used as the root node
        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

        // create a button and add it to the flow pane
        Button button = new Button("Select background colour");
        root.getChildren().add(button);

        // create buttons for the menu choices
        Button red = new Button(" Red  ");
        Button yellow = new Button("Yellow");
        Button green = new Button(" Green ");

        // create a VBox to hold the buttons
        VBox box = new VBox(10);
        box.setAlignment(Pos.CENTER);
        box.getChildren().addAll(red, yellow, green);

        // create a secondary scene
        Scene secondaryScene = new Scene(box,200,150);

        // create a secondary stage
        Stage secondaryStage = new Stage();

        // add the secondary scene to the secondaryary stage
        secondaryStage.setScene(secondaryScene);

        // set the modality of the secondary stage
        secondaryStage.initModality(Modality.APPLICATION_MODAL);

        // code the button so that the secondary stage is made visible
        button.setOnAction(e ->
        {               secondaryStage.setX(primaryStage.getX() + 250);
                                 secondaryStage.setY(primaryStage.getY() + 100);
                                 secondaryStage.show();
                                         });

        // code the menu items
        red.setOnAction(e -> {
                                  root.setBackground(new Background(new BackgroundFill(Color.RED,
                                                               CornerRadii.EMPTY, Insets.EMPTY)));
                                  secondaryStage.hide();
                              }
                         );

        yellow.setOnAction(e -> {
                                  root.setBackground(new Background(new BackgroundFill(Color.YELLOW,
                                                                CornerRadii.EMPTY, Insets.EMPTY)));
                                  secondaryStage.hide();
                                 }
                            );

        green.setOnAction(e -> {
                                  root.setBackground(new Background(new BackgroundFill(Color.GREEN,
                                                                CornerRadii.EMPTY, Insets.EMPTY)));
                                  secondaryStage.hide();
                                }
                          );

        // create the primary scene and stage
        Scene primaryScene = new Scene(root, WIDTH, HEIGHT);
        primaryStage.setScene(primaryScene);
        primaryStage.setTitle("Modal Context Menu");
        primaryStage.show();

        }

        public static void main(String[] args)
        {
            launch(args);
        }
}
```

Much of this you will have seen before. The important thing here is how we create a secondary scene (which will hold a VBox containing the buttons) and a secondary stage:

```
Scene secondaryScene = new Scene(box,200,150);
Stage secondaryStage = new Stage();
secondaryStage.setScene(secondaryScene);
```

The Stage class has a method called initModality, which allows us to make the stage modal with respect to the rest of the application, so that once the stage (our menu in this case) appears the application freezes until it disappears. We do this with the following line of code:

```
secondaryStage.initModality(Modality.APPLICATION_MODAL);
```

17.4 Combo Boxes

You will be familiar with a combo box as it is a very common way of offering choices. Our example is shown in Fig. 17.5.

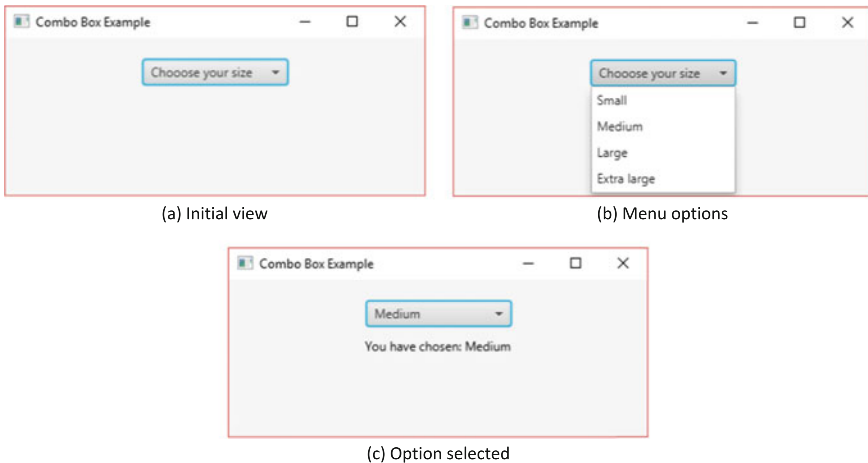


Fig. 17.5 Combo box example

Here is the code:

ComboBoxExample

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ComboBoxExample extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 150;

        // declare a String type combo box
        ComboBox<String> box = new ComboBox<>();

        // add the choices
        box.getItems().addAll("Small", "Medium", "Large", "Extra large");

        // set the initial text
        box.setValue("Chooose your size");

        Label message = new Label();

        // display the user's choice
        box.setOnAction(e -> message.setText("You have chosen: " + box.getValue()));

        VBox root = new VBox(10);
        root.setPadding(new Insets(20, 20, 20, 20));
        root.setAlignment(Pos.TOP_CENTER);

        root.getChildren().addAll(box, message);

        Scene scene = new Scene(root, WIDTH, HEIGHT);
        stage.setScene(scene);
        stage.setTitle("Combo Box Example");
        stage.show();
    }

    public static void main(String[] args)
    {
            launch(args);
    }
}
```

The only new thing here is the combo box itself. The `ComboBox` class is a generic class, so that the type of items held can vary—most commonly the box will hold strings, but we could just as easily have images for example. In our case we are using strings, and the declaration is therefore as follows:

```
ComboBox<String> box = new ComboBox<>();
```

The menu items are added by using the `getItems` method:

```
box.getItems().addAll("Small", "Medium", "Large", "Extra large");
```

We want the box to start off displaying the instruction, so we use the `setValue` method for this purpose:

```
box.setValue("Chooose your size");
```

The other point to note is the use of the `getValue` method to retrieve the current item displayed—we use this to display the choice made by the user:

```
box.setOnAction(e -> message.setText("You have chosen: " + box.getValue()));
```

17.5 Check Boxes and Radio Buttons

Check boxes are a very familiar way of offering choices. Our simple example is shown in Fig. 17.6.

You can see the code for this below. By now this should be self-explanatory—but notice that in this case we used the method `isSelected` to determine whether the box is selected or not.

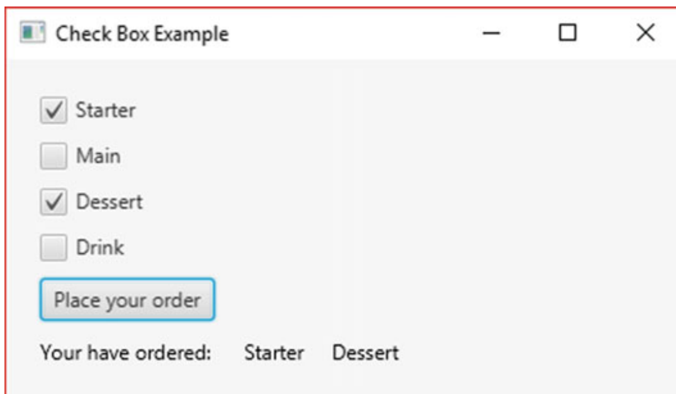


Fig. 17.6 Check box example

CheckBoxExample

```
import javafx.scene.control.Button;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CheckBoxExample extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 200;

        // create four check boxes
        CheckBox starter = new CheckBox("Starter");
        CheckBox mainCourse = new CheckBox("Main");
        CheckBox dessert = new CheckBox("Dessert");
        CheckBox drink = new CheckBox("Drink");

        Button submitButton = new Button("Place your order");
        Label message = new Label();

        // clicking the button
        submitButton.setOnAction(e -> {
                                        String yourOrder = "Your have ordered: ";

                                        if(!starter.isSelected()&& !mainCourse.isSelected()
                                                   && !dessert.isSelected() && !drink.isSelected())
                                        {
                                                yourOrder = "You did not select anything";
                                        }
                                        else
                                        {
                                                if(starter.isSelected())
                                                {
                                                       yourOrder = yourOrder + "     Starter";
                                                }
                                                if(mainCourse.isSelected())
                                                {
                                                       yourOrder = yourOrder + "     Main";
                                                }
                                                if(dessert.isSelected())
                                                {
                                                       yourOrder = yourOrder + "     Dessert";
                                                }
                                                if(drink.isSelected())
                                                {
                                                       yourOrder = yourOrder + "     Drink";
                                                }
                                        }
                                             message.setText(yourOrder);
                                      }
                                 );


        VBox root = new VBox(10);
        root.setPadding(new Insets(20, 20, 20, 20));
        root.setAlignment(Pos.CENTER_LEFT);

        root.getChildren().addAll(starter, mainCourse, dessert, drink, submitButton, message);

        Scene scene = new Scene(root, WIDTH, HEIGHT);

        stage.setScene(scene);
        stage.setTitle("Check Box Example");
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

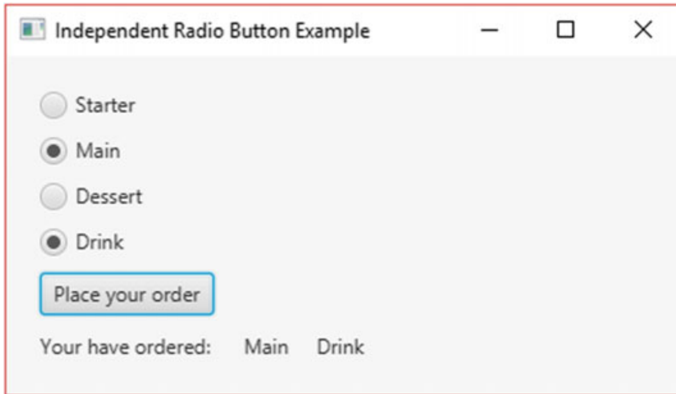


Fig. 17.7 Independent radio buttons

Radio buttons are very similar to check boxes, although they are round instead of square. They can operate in exactly the same way as check boxes, but can also be made to operate as a group, so that only one item can be selected at a time; if a box is selected and then the user chooses another box, the first one is cleared.

Figure 17.7 shows an example of radio buttons working independently. You can see that it is the same as our check box example, with the check boxes replaced by radio buttons.

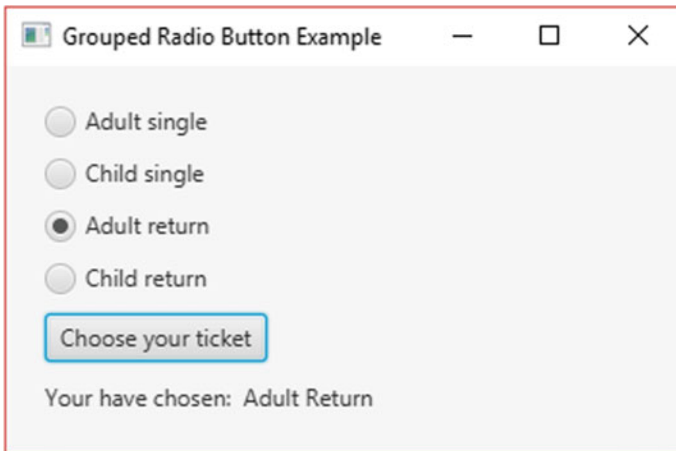


Fig. 17.8 Grouped radio buttons

All we needed to do was to replace the declarations of the four check boxes with the following code:

```
RadioButton starter = new RadioButton("Starter");
RadioButton mainCourse = new RadioButton("Main");
RadioButton dessert = new RadioButton("Dessert");
RadioButton drink = new RadioButton("Drink");
```

Figure 17.8 shows an example of radio buttons acting together in a group—only one item can be selected.

Here is the code:

GroupedRadioButtonExample

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.geometry.Pos;
import javafx.scene.control.ToggleGroup;

public class GroupedRadioButtonExample extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 350;
        final double HEIGHT = 200;

        // declare the radio buttons
        RadioButton adultSingle = new RadioButton("Adult single");
        RadioButton childSingle = new RadioButton("Child single");
        RadioButton adultReturn = new RadioButton("Adult return");
        RadioButton childReturn = new RadioButton("Child return");

        // add the radio buttons to a toggle group
        ToggleGroup group = new ToggleGroup();
        group.getToggles().addAll(adultSingle, childSingle, adultReturn, childReturn);

        Button submitButton = new Button("Choose your ticket");
        Label message = new Label();

        // clicking the button
        submitButton.setOnAction(e-> {
                                        String yourOrder = "Your have chosen: ";
                                        if(!adultSingle.isSelected()&& !childSingle.isSelected()
                                          && !adultReturn.isSelected() && !childReturn.isSelected())
                                        {
                                                 yourOrder = "You did not chose a ticket";
                                        }
                                        else
                                        {
                                                 if(adultSingle.isSelected())
                                                 {
                                                        yourOrder = yourOrder + " Adult Single";
                                                 }
                                                 else if(childSingle.isSelected())
                                                 {
                                                        yourOrder = yourOrder + " Child Single";
                                                 }
                                                 else if(adultReturn.isSelected())
                                                 {
                                                        yourOrder = yourOrder + " Adult Return";
                                                 }
```

```
                                        else if(childReturn.isSelected())
                                        {
                                               yourOrder = yourOrder + " Child Return";
                                        }
                               }
                               message.setText(yourOrder);
                            }
                    );

           VBox root = new VBox(10);
           root.setPadding(new Insets(20, 20, 20, 20));
           root.setAlignment(Pos.CENTER_LEFT);
           root.getChildren().addAll(adultSingle, childSingle, adultReturn, childReturn,
                                                   submitButton, message);

           Scene scene = new Scene(root, WIDTH, HEIGHT);
           stage.setScene(scene);
           stage.setTitle("Grouped Radio Button Example");
           stage.show();
       }

       public static void main(String[] args)
       {
           launch(args);
       }
   }
```

As you can see we have declared a ToggleGroup and added our radio buttons to it:

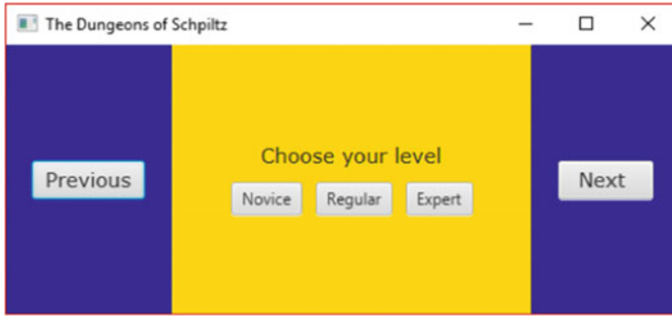
```
ToggleGroup group = new ToggleGroup();
group.getToggles().addAll(adultSingle, childSingle, adultReturn, childReturn);
```

The buttons now act as one unit—if a button is already selected when another button is chosen, then the first button is cleared.

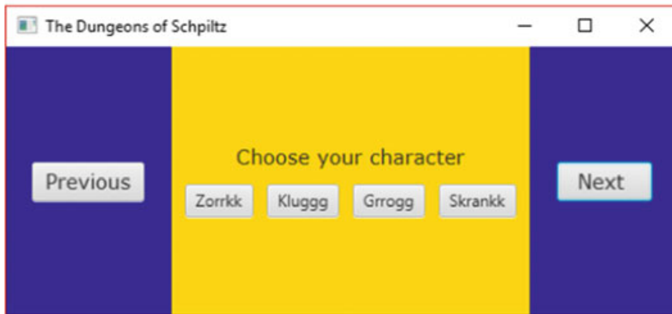
17.6 A Card Menu

A very familiar way of entering information into forms is via a serious of screens. As we mentioned in Chap. 10, one way to achieve this is via a StackPane, which enables us to present a series of containers as if they represented a pack of cards. The top "cards" can be removed in order, revealing the card underneath—or we could move in the other direction, replacing the cards in the order they were removed.

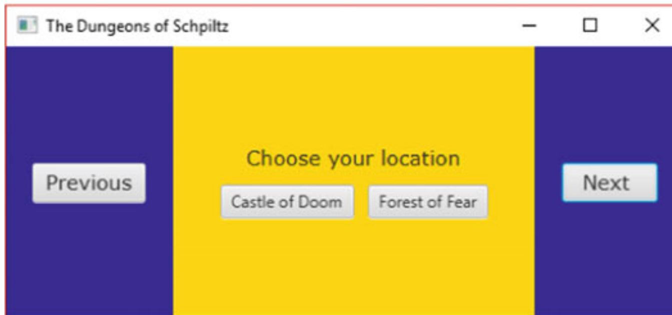
The example shown in Fig. 17.9 represents the initial screens in a made-up game ("The Dungeons of Schpiltz"). A StackPane holds the cards, each of which is a VBox containing buttons. We have not coded the buttons, as it is just for illustration. On either side of the StackPane are a "Previous" button and a "Next" button.



(a) The first screen allows the user to choose a level



(b) The second screen allows the user to choose a character



(c) The third screen allows the user to choose a location

Fig. 17.9 A card menu

Here is the complete code:

CardMenuExample

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.paint.Color;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.text.Font;

public class CardMenuExample extends Application
{
    private int currentCard = 0; // to keep track of the cards
    private StackPane stack = new StackPane();  // to hold the cards

    @Override
    public void start(Stage stage)
    {
        // create a label for the first card (choosing the level)
        Label levelLabel = new Label("Choose your level");
        levelLabel.setFont(Font.font ("Verdana", 15));

        // create an HBox containing three dummy buttons for the first card (choosing the level)
        HBox levelButtons = new HBox(10);
        levelButtons.getChildren().addAll(new Button("Novice"),
                                            new Button("Regular"), new Button ("Expert"));
        levelButtons.setAlignment(Pos.CENTER);

        // create a VBox to act as the first card
        VBox levelPanel = new VBox(10);

        // add the label and buttons to the first VBox
        levelPanel.getChildren().addAll(levelLabel, levelButtons);
        levelPanel.setAlignment(Pos.CENTER);

        // create a label for the second card (choosing the character)
        Label characterLabel = new Label("    Choose your character    ");
        characterLabel.setFont(Font.font ("Verdana", 15));

        // create an HBox containing four dummy buttons for the second card (choosing the character)
        HBox characterButtons = new HBox(10);
        characterButtons.getChildren().addAll(new Button("Zorrkk"), new Button("Kluggg"),
                                            new Button ("Grrogg"), new Button("Skrankk"));
        characterButtons.setAlignment(Pos.CENTER);

        // create a VBox to act as the second card
        VBox characterPanel = new VBox(10);
        characterPanel.getChildren().addAll(characterLabel, characterButtons);
        characterPanel.setAlignment(Pos.CENTER);

        // create a label for the second card (choosing the location)
        Label locationLabel = new Label("Choose your location");
        locationLabel.setFont(Font.font ("Verdana", 15));

        // create an HBox containing two dummy buttons for the third card (choosing the location)
        HBox locationButtons = new HBox(10);
        locationButtons.getChildren().addAll(new Button("Castle of Doom"),
                                                       new Button("Forest of Fear"));
        locationButtons.setAlignment(Pos.CENTER);

        // create a VBox to act as the third card
        VBox locationPanel = new VBox(10);
        locationPanel.getChildren().addAll(locationLabel, locationButtons);
        locationPanel.setAlignment(Pos.CENTER);


        // create and configure buttons for moving back and forth through the cards
        Button nextButton = new Button(" Next  ");
        Button previousButton = new Button("Previous");

        nextButton.setFont(Font.font ("Verdana", 15));
        previousButton.setFont(Font.font ("Verdana", 15));
```

```
        // configure the stack pane
        stack.setPadding(new Insets(10, 10, 10, 10));
        stack.setBackground(new Background(new BackgroundFill(Color.GOLD,
                                                     CornerRadii.EMPTY, Insets.EMPTY)));
        stack.setAlignment(Pos.CENTER);

        // add the cards to the stack pane
        stack.getChildren().addAll(levelPanel, characterPanel, locationPanel);

        // show the first card and hide the other two
        stack.getChildren().get(0).setVisible(true);
        stack.getChildren().get(1).setVisible(false);
        stack.getChildren().get(2).setVisible(false);

        // create and configure an HBox
        HBox root = new HBox(20);
        root.setBackground(Background.EMPTY);
        root.setAlignment(Pos.CENTER);

        // add the "previous" button, the stack of cards and the "next" button
        root.getChildren().addAll(previousButton, stack, nextButton);

        // add event handlers to call the relevant helper methods
        nextButton.setOnAction (e -> next());
        previousButton.setOnAction (e -> previous());


        // create and configure the scene
        Scene scene = new Scene(root, 500, 200, Color.DARKBLUE);

        // configure the stage
        stage.setScene(scene);
        stage.setTitle("The Dungeons of Schpiltz");
        stage.show();
    }

    // define the helper method for the "next" button
    private void next()
    {
            if(currentCard !=  stack.getChildren().size()- 1) // if we are not at the last card
            {
                currentCard++; // make the next card the current card

                // move through the cards: show the current card, hide the others
                for(int i = 0; i <=  stack.getChildren().size()- 1; i++)
                {
                    if(i == currentCard)
                    {
                        stack.getChildren().get(i).setVisible(true);
                    }
                    else
                    {
                        stack.getChildren().get(i).setVisible(false);
                    }
                }
            }
    }

    // define the helper method for the "previous" button
     private void previous()
     {
             if(currentCard != 0) // if we are not at the first card
             {
                 currentCard--; // make the previous card the current card

                 // move through the cards: show the current card, hide the others
                 for(int i = 0; i <=  stack.getChildren().size()- 1; i++)
                 {
                     if(i == currentCard)
                     {
                         stack.getChildren().get(i).setVisible(true);
                     }
                     else
                     {
                         stack.getChildren().get(i).setVisible(false);
                     }
                 }
             }
     }

     public static void main(String[] args)
     {
         launch(args);
     }

}
```

We are using helper methods for the “Previous” button and the “Next” buttons, so we have defined a couple of attributes that can be accessed by these methods. The first is a counter that keeps track of the current card, the second is the stack pane that will hold the cards:

```
private int currentCard = 0;
private StackPane stack = new StackPane();
```

The `start` method begins by creating the first card. First we create a label which gives the instruction to the user:

```
Label levelLabel = new Label("Choose your level");
levelLabel.setFont(Font.font ("Verdana", 15));
```

Next an `HBox` to hold three dummy buttons:

```
HBox levelButtons = new HBox(10);
levelButtons.getChildren().addAll(new Button("Novice"),
                                                 new Button("Regular"), new Button ("Expert"));
levelButtons.setAlignment(Pos.CENTER);
```

Finally we create a `VBox` to act as the first card, then add the label and the button container to it:

```
VBox levelPanel = new VBox(10);
levelPanel.getChildren().addAll(levelLabel, levelButtons);
levelPanel.setAlignment(Pos.CENTER);
```

We do the same thing for the other two cards, then go on to configure the stack pane and add the three cards to it.

The items held by the stack pane are indexed from zero (in the order they were added), and they are retrieved with the `get` method of the list retrieved by the `getChildren` method. Using the indices we set the initial state, with the first card visible and the other two invisible:

```
stack.getChildren().get(0).setVisible(true);
stack.getChildren().get(1).setVisible(false);
stack.getChildren().get(2).setVisible(false);
```

We go on to create and configure an `HBox`, to which we add the “Previous” button, the stack and the “Next” button.

Before we finally create the scene and the stage, we add our event handlers to the two buttons; these call the helper methods, `next` and `previous`.

```
nextButton.setOnAction (e -> next());
previousButton.setOnAction (e -> previous());
```

The code for the `next` method is as follows:

```
private void next()
{
        if(currentCard != stack.getChildren().size()- 1)
        {
            currentCard++;
            for(int i = 0; i <= stack.getChildren().size()- 1; i++)
            {
                if(i == currentCard)
                {
                    stack.getChildren().get(i).setVisible(true);
                }
                else
                {
                    stack.getChildren().get(i).setVisible(false);
                }
            }
        }
}
```

First we have checked that we are not at the last card—for this purpose we have used the `getSize` method of the list of items (although we know that the index of the last item is 2, doing it this way would allow us to add more cards without changing the code).

If we are not at the last card we increment the counter and then cycle through the cards; we set the current card to be visible, the others to be invisible.

The `previous` method behaves in a similar way.

17.7 The *Dialog* Class

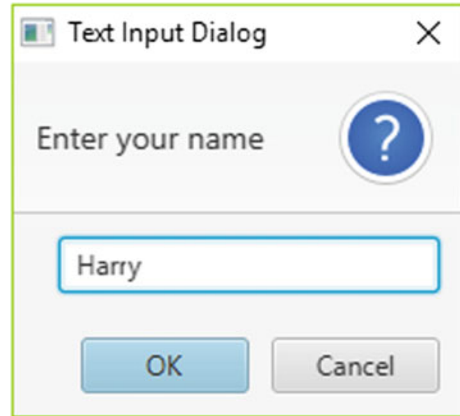
JavaFX provides a very useful control class called `Dialog`, which has subclasses called `Alert`, `ChoiceDialog`, `TextInputDialog`. These provide popup windows which allow the user to view or enter information. One of the very useful aspects of this is that we can begin an application by showing a popup window that gets some information from the user before showing the main scene graphic.

The program below demonstrates how these classes work.

DialogDemo

```
import java.util.Optional;
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Button;
import javafx.scene.control.ChoiceDialog;
import javafx.scene.control.Label;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class DialogDemo extends Application
{
    private String name;
    private String colour;

    @Override
    public void start(Stage stage)
    {
        name = getUserName(); // get the user name by calling a text input dialog

        Label label1 = new Label();
        Label label2 = new Label();
        Button button1 = new Button ("Alert");
        Button button2 = new Button ("Choice");

        button1.setOnAction(e -> showAlert()); // show an alert

        // call a choice dialog
        button2.setOnAction(e ->
                                 {
                                    colour = showChoice();
                                    label2.setText("You chose " + colour);
                                 }
                            );

        VBox root = new VBox(10);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(label1, button1, button2, label2);
        label1.setFont(Font.font("Ariel", 20));
        label2.setFont(Font.font("Ariel", 20));
        label1.setText("Hello " + name);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Demo");
        stage.setWidth(250);
        stage.setHeight(250);
        stage.show();
    }
    private String getUserName()
    {
        TextInputDialog dialog = new TextInputDialog();
        dialog.setHeaderText("Enter your name");
        dialog.setTitle("Text Input Dialog");

        Optional<String> response = dialog.showAndWait();
        return response.get();
    }

    private void showAlert()
    {
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setHeaderText("Information Alert");
        alert.setContentText(name + " is a cool name");
        alert.showAndWait();
    }

    private String showChoice()
    {
        ChoiceDialog<String> choice = new ChoiceDialog<>("Red", "Yellow", "Blue");
        choice.setContentText("Choose colour");
        choice.setHeaderText("Choice dialog");

        Optional<String> response = choice.showAndWait();
        return response.get();
    }

    public static void main(String[] args)
    {
        launch(args);
    }

}
```

Fig. 17.10 A text input dialog



As you can see, the first thing that happens, even before the scene is configured and shown, is that a helper method `getUserName` is called. This causes the following popup to appear as shown in Fig. 17.10.

The code for `getUserName` is as follows:

```
private String getUserName()
{
    TextInputDialog dialog = new TextInputDialog();
    dialog.setHeaderText("Enter your name");
    dialog.setTitle("Text Input Dialog");

    Optional<String> response = dialog.showAndWait();
    return response.get();
}
```

As you can see from the code, we create and configure a `TextInputDialog`, then call its `showAndWait` method, which does exactly what it says—shows the dialogue and waits for a value to be entered. The value entered is returned as an `Optional` object (as explained in Chap. 14), in this case `Optional <String>`. The `String` value is retrieved with the `get` method of `Optional`.

Once the dialogue is closed, the main graphic appears (Fig. 17.11).

The two buttons are provided to demonstrate the `Alert` and the `ChoiceDialog` classes. Pressing the "Alert" button calls another helper method, `showAlert`, which brings up the dialogue window shown below in Fig. 17.12:

The code for `showAlert` is as follows:

```
    private void showAlert()
    {
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setHeaderText("Information Alert");
        alert.setContentText("name + " is a cool name");
        alert.showAndWait();
    }
```

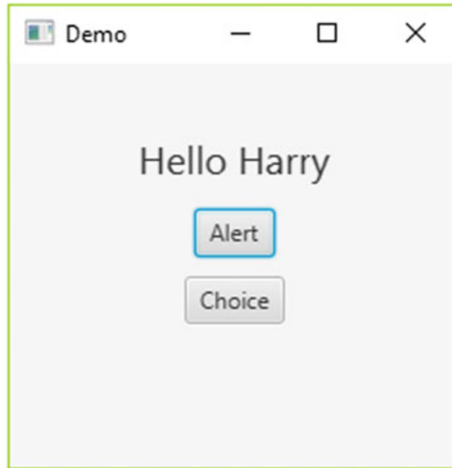


Fig. 17.11 Dialog demo application

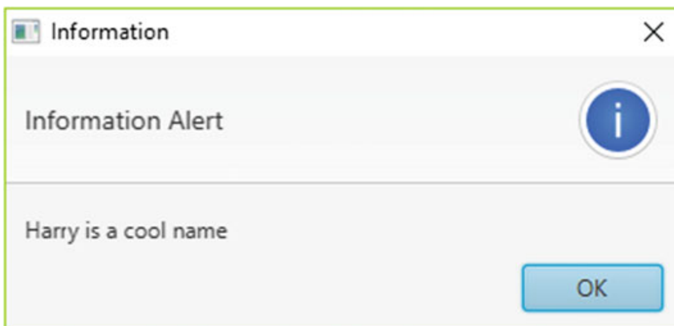


Fig. 17.12 An information alert

There are a number of different types of `Alert`, and the particular type is provided as a parameter to the constructor. The type that you see in Fig. 17.12 is `AlertType.INFORMATION`. Other types are shown in Fig. 17.13:

There is an additional constructor of `Alert` that enables you to choose which buttons you would like. It takes the following form:

```
Alert(Alert.AlertType alertType, String contentText,
ButtonType... buttons)
```

Fig. 17.13 Other alert types

The last of these parameters, `buttons`, allows you to decide upon the type or button—or buttons—you require. So for example, the following statement:

```
Alert alert
   = new Alert(AlertType.INFORMATION, "Alert with three buttons", ButtonType.APPLY, ButtonType.OK, ButtonType.CLOSE);
```

would give rise to the alert shown in Fig. 17.14.

The button types available are shown in Table 17.1.

In order to find out which of the three buttons had been pressed you would need to examine the return type of the `showAndWait` method:

```
Optional<ButtonType> response = alert.showAndWait();
```

The `ButtonType` could then be extracted with the `get` method of `Optional`, and a string representing the button type could then be extracted with the `getText` method of `ButtonType`:

```
String buttonPressed = response.get().getText();
```

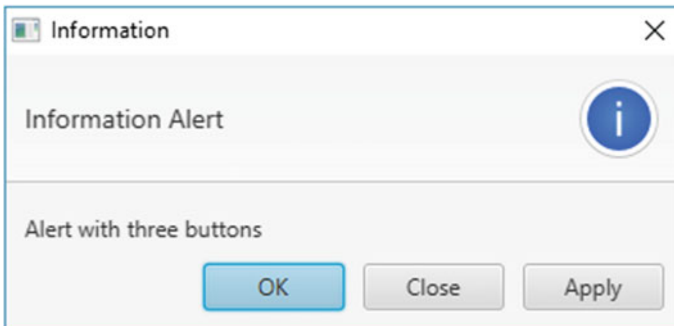


Fig. 17.14 A choice dialog with three buttons

Table 17.1 Available button types

ButtonType.APPLY
ButtonType.CLOSE
ButtonType.CANCEL
ButtonType.FINISH
ButtonType.NEXT
ButtonType.PREVIOUS
ButtonType.YES
ButtonType.NO
ButtonType.OK

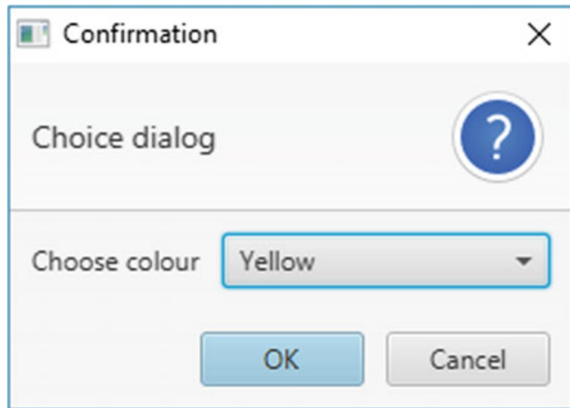


Fig. 17.15 A choice dialog

Finally, the "choice" button in Fig. 17.11 will invoke the `showChoice` method which brings up a choice dialog (Fig. 17.15).

Here is the code for `showChoice`, which by now should be self-explanatory (notice, however, that `ChoiceDialog` is a generic class and requires the type of the items to be held):

```
ChoiceDialog<String> choice = new ChoiceDialog<>("Red", "Yellow", "Blue");
choice.setContentText("Choose colour");
choice.setHeaderText("Choice dialog");

Optional<String> response = choice.showAndWait();
return response.get();
```

Once this dialogue window is closed, the choice made is shown (Fig. 17.16).

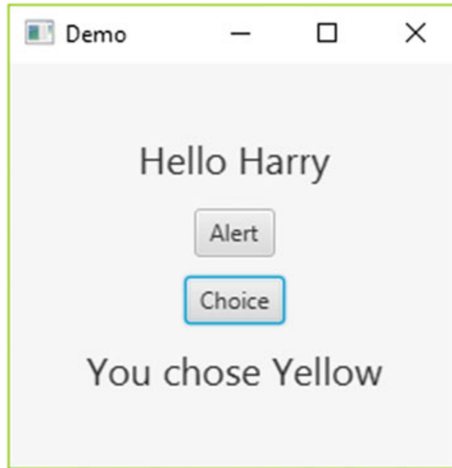
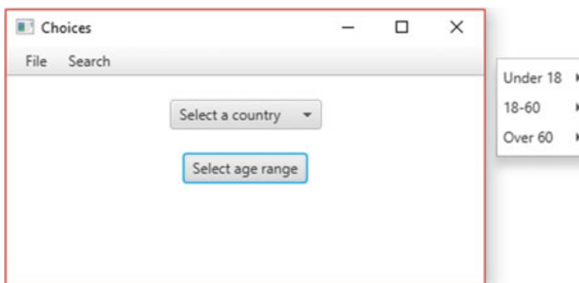


Fig. 17.16 Dialog demo after the choice of colour has been made

17.8 Self-test Questions

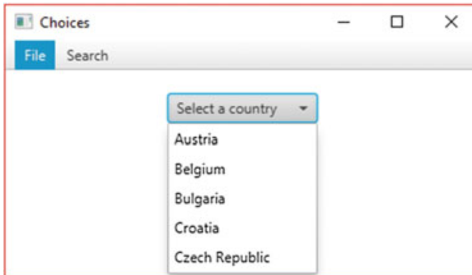
1. (a) In the application shown below, identify the various ways that users are given for making choices.



   (b) What alternatives could have been used for selecting the age range?

2. What is the difference between a *modal* and a *non-modal* dialogue?
3. In what circumstances might a *context* menu be preferable to a *drop-down* menu?

4. Explain how a number of *radio buttons* can be made to work together.
5. The diagram below shows the choices available under the "Select a country" option of the application shown in question 1.



Referring to the above diagram, explain how you would begin a conditional statement that would execute some code if Bulgaria had been chosen.

17.9 Programming Exercises

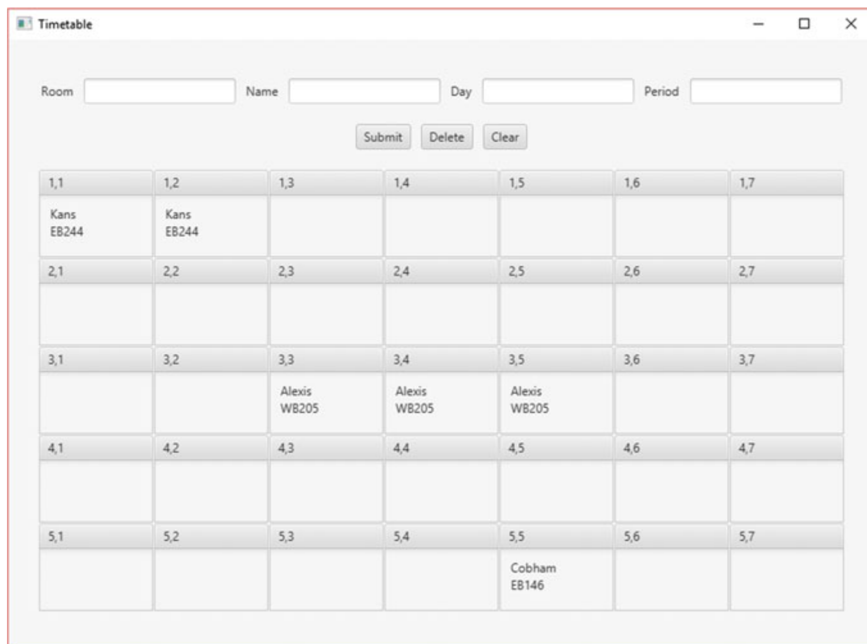
1. Implement a few of the programs that we have developed in this chapter, and experiment with different settings in order to change some the features.
2. Adapt the `Hostel` case study of Chaps. 11 and 12 as follows:

   (a) Make use of a `ComboBox` to enter the month.

   (b) Enable the number of rooms to be entered via a `Dialog`, as described in Sect. 17.7.

   (c) Make use of `Alerts`, as described in Sect. 17.7.

3. Create a graphical user interface for the `Library` class that we developed in Chap. 15. Make use of the many JavaFX features discussed in this chapter.
4. At the end of Chap. 8 you were asked to develop a time table application. You later enhanced this application by making use of exceptions at the end of Chap. 14. Now develop a JavaFX GUI for this application, with the timetable displayed as a grid. The following is an example—you can choose your own design:


Timetable
Room
Name
Day
Period
Submit
Delete
Clear
1,1
1,2
1,3
1,4
1,5
1,6
1,7
Kans
EB244
Kans
EB244
2,1
2,2
2,3
2,4
2,5
2,6
2,7
3,1
3,2
3,3
3,4
3,5
3,6
3,7
Alexis
WB205
Alexis
WB205
Alexis
WB205
4,1
4,2
4,3
4,4
4,5
4,6
4,7
5,1
5,2
5,3
5,4
5,5
5,6
5,7
Cobham
EB146

18 Working with Files

Outcomes:

By the end of this chapter you should be able to:

- *explain the principles of* ***input*** *and* ***output*** *and identify a number of different input and output devices*;
- *explain the concept of an* ***I/O stream***;
- *describe the basic file-handling techniques used in the Java language*;
- *distinguish between* ***text***, ***binary*** *and* ***object*** *encoding of data*;
- *distinguish between* ***serial*** *access files and* ***random*** *access files*;
- *create and access files in Java using all the above encoding and access methods.*

18.1 Introduction

When we developed our case study in Chaps. 11 and 12 it became apparent that in reality an application such as that one wouldn't be much use unless we had some way of storing our data permanently—even when the program has been terminated and the computer has been switched off. You will remember in those chapters, because you had not yet learnt how to do this, we provided a special class called `TenantFileHandler` that enabled you to keep permanent records on disk.

Now it is time to learn how to do this yourself. As you are no doubt already aware, a named block of externally stored data is called a **file**.

When we are taking an object-oriented approach, as we have been doing, we tend not to separate the data from the behaviour; however, when it comes to storing information in files then of course it is only the data that we are interested in storing. When referring to data alone it is customary to use the terms **record** and **field**. A record refers to a single data instance—for example a person, a stock-item, a

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_18

student and so on; a **field** refers to what in the object-oriented world we would normally call an attribute—a name, a stock number, an exam mark etc.

In this chapter we will learn how to create files, and write information to them, and to read the information back when we need it. We start by looking at this process in the overall context of input and output, or I/O as it is often called; you will then go on to learn a number of different techniques for keeping permanent copies of your data.

18.2 Input and Output

Any computer system must provide a means of allowing information to come in from the outside world (**input**) and, once it has been processed, to be sent out again (**output**). The whole question of input and output, particularly where files are concerned, can sometimes seem rather complex, especially from the point of view of the programmer.

As with all aspects of a computer system, the processes of input and output are handled by the computer hardware working in conjunction with the system software—that is, the operating system (Windows™, macOS™ or Linux™ for example). The particular application program that is running at the time normally deals with input and output by communicating with the operating system, and getting it to perform these tasks in conjunction with the hardware.

All this involves some very real complexity and involves a lot of low-level details that a programmer is not usually concerned with; for example, the way in which the system writes to external media such as disks, or the way it reconciles the differences between the speed of the processor with the speed of the disk-drive.

18.3 Input and Output Devices

The most common way of getting data input from the outside world is via the keyboard; and the most common way of displaying output data is on the screen. Therefore, most systems are normally set up so that the *standard* input and output devices are the keyboard and the screen respectively. However, there are many other devices that are concerned with input and output: magnetic, solid state and optical disks for permanent storage of data (both local and remote); flash drives; network interface cards and modems for communicating with other computers; and printers for producing hard copies.

We should bear in mind that the process, in one sense, is always the same, no matter what the input or output device. All the data that is processed by the computer's central processing unit in response to program instructions is stored in the computer's main memory or RAM (Random Access Memory). Input is the transfer of data from some external device to main memory whereas output is the

transfer of data from main memory to an external device. In order for input or output to take place, it is necessary for a channel of communication to be established between the device and the computer's memory. Such a channel is referred to as a **stream**. The operating system will have established a **standard input stream** and a **standard output stream**, which will normally be the keyboard and screen respectively. In addition to this, there is usually a **standard error stream** where error messages can be displayed; this is normally also set to the screen. All of these default settings for the standard streams can be changed either via the operating system or from within the program itself.

In previous chapters you have seen that the `System` class has two attributes called `in` and `out`. In addition to this, it has an additional attribute called `err`; these objects are already set up to provide access to the standard input, output and error streams. The attribute `in` is an object of a class called `InputStream`. This class provides some low-level methods to deal with basic input—they are low-level because they deal with sequences of bytes, rather than characters. A higher-level class, `InputStreamReader` can be wrapped around this class to deal with character input; `InputStreamReader` objects can subsequently be wrapped by another class, `BufferedReader`, which handles input in the form of strings.

This rather complex way of reading from the keyboard is how things were done before Java 5.0 provided the `Scanner` class. The following program illustrates how you would get keyboard input in this manner.

KeyboardInput

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class KeyboardInput
{
    public static void main(String[] args)
    {
        InputStreamReader input = new InputStreamReader(System.in); // to handle low-level details
        BufferedReader reader = new BufferedReader(input); // to handle high-level details

        try
        {
            System.out.print("Enter a string: ");
            String test = reader.readLine();  // gets a string of characters from the keyboard
            System.out.println("You entered: " + test);
        }

        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

You can see that the `readLine` method of `BufferedReader` is used to get a string of characters from the keyboard; as you would expect, the method reads characters from the keyboard until the user presses the enter key. It throws an `IOException` if an error (such as a keyboard lock) occurs during the process, and this has to be handled.

In this chapter, instead of dealing with input and output to the standard streams, we are going to be dealing with the input and output of data to external disk drives in the form of files—but, as you will see, the principles are the same.

18.4 File-Handling

The output process, which consists of transferring data from memory to a file, is usually referred to as **writing**; the input process, which consists of transferring data from a file to memory, is referred to as **reading**. Both of these involve some low-level detail to do with the way in which data is stored physically on the disk. As programmers we do not want to have to worry more than is necessary about this process—which, of course, will differ from one machine to the next and from one operating system to the next. Fortunately, Java makes it quite easy for us to deal with these processes. As we shall see, Java provides low-level classes which create **file streams**—input or output streams that handle communication between main memory and a named file on a disk. It also provides higher-level classes which we can "wrap around" the low-level objects, enabling us to use methods that relate more closely to our logical way of thinking about data. In this way we are shielded from having to know too much detail about the way our particular system stores and retrieves data to or from a file.

As we shall see, this whole process enables us to read and write data in terms of units that we understand—for example, in the form of strings, lines of text, or basic types such integers or characters; Java even allows us to store and retrieve whole objects.

18.4.1 Encoding

Java supports three different ways of **encoding** data—that is, representing data on a disk. These are **text**, **binary** and **object**.

Text encoding means that the data on the disk is stored as characters in the form used by the external system (often ASCII). Java, as we know, uses the Unicode character set, so, depending on the form used by the external system, some conversion might take place in the process, but fortunately the programmer does not have to worry about that. As an example, consider saving the number 107 to a text file—it will be saved as the character '1' in ASCII code (or whatever is used by the system) followed by the character '0', followed by the character '7'. A text file is therefore readable by a text editor (such as Windows™ Notepad).

Binary encoding, on the other hand, means that the data is stored in the same format as the internal representation of the data used by the program to store data in memory. So the number 107 would be saved as the binary number 1101011. A binary file could not be read properly by a text editor as we shall see in Sect. 18.6.

Finally, there is object-encoding which is a powerful mechanism provided by Java whereby a whole object can be input or output with a single command.

You are probably asking yourself which is the best method to use when you start to write applications that read and write to files. Well, if your files are going to be read and written by the same application, then it really makes very little difference how they are encoded. Just use the method that seems the easiest for the type of data you are storing. However, do bear in mind that if you wanted your files to be read by a text editor then you must, of course, use the text encoding method.

18.4.2 Access

The final thing that you need to consider before we show you how to write files in Java is the way in which files are accessed. There are two ways in which this can take place—**serial** access and **random** access. In the first (and more common) method, each item of data is read (or written) in turn. The operating system provides what is known as a **file pointer**, which is really just a location in memory that keeps track of where we have got to in the process of reading or writing to a file.

Another way to access data in a file is to go directly to the record you want—this is known as random access, and is a bit like going straight to the clip you want on a DVD; whereas serial access is like using an old fashioned video tape, where you have to work your way through the entire tape to get to the bit you want. Java provides a class (`RandomAccessFile`) that we can use for random access. We will start, however, with serial access.

18.5 Reading and Writing to Text Files

In this and the following section we are going to use as an example a very simple class called `Car`; the code for this class is given below:

The *Car* class

```
public class Car
{
   private String registration;
   private String make;
   private double price;

   public Car(String registrationIn, String makeIn, double priceIn)
   {
      registration = registrationIn;
      make = makeIn;
      price = priceIn;
   }

   public String getRegistration()
   {
      return registration;
   }

   public String getMake()
   {
      return make;
   }

   public double getPrice()
   {
      return price;
   }
}
```

The program below, `TextFileTester`, is a very simple menu-driven program that manipulates a list of cars, held in memory as a `List`; it provides the facility to add new cars to the list, to remove cars from the list and to display the details of all the cars in the list. As it is a demonstration program only, we have not bothered with such things as input validation, or checking if the list is empty before we try to remove an item.

The difference between this and other similar programs that we have discussed before, is that the list is kept as a permanent record—as we mentioned before, we did a similar thing in our case study in Chap. 12, but there the process was hidden from you.

The program is designed so that reading and writing to the file takes place as follows: when the quit option is selected, the list is written as a permanent text file called `Cars.txt`; each time the program is run, this file is read into memory.

The program is presented below; notice that we have provided two helper methods, `writeList` and `readList` for the purpose of accessing the file; as we shall explain, the `writeList` method also deals with creating the file for the first time. Notice also that, for convenience, we are making use of the `EasyScanner` class that we developed in Chap. 8.

TextFileTester

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;

public class TextFileTester
{
   public static void main(String[] args)
   {
      char choice;

      // create an empty list to hold Cars
      List<Car> carList = new ArrayList<>();

      // read the list from file when the program starts
      readList(carList);

      // menu options
      do
      {
         System.out.println("\nText File Tester");
         System.out.println("1. Add a car");
         System.out.println("2. Remove a car");
         System.out.println("3. List all cars");
         System.out.println("4. Quit\n");
         choice = EasyScanner.nextChar();
         System.out.println(); switch(choice)
         {
            case '1' : addCar(carList);
                       break;
            case '2' : removeCar(carList);
                       break;
            case '3' : listAll(carList);
                       break;
            case '4' : writeList(carList); // write to the file
                       break;
            default : System.out.print("\nPlease choose a number from 1 - 4 only\n ");
         }
      }while(choice != '4');
   }

   // method for adding a new car to the list
   static void addCar(List<Car> carListIn)
   {
      String tempReg;
      String tempMake;
      double tempPrice;

      System.out.print("Please enter the registration number: ");
      tempReg = EasyScanner.nextString();
      System.out.print("Please enter the make: ");
      tempMake = EasyScanner.nextString();
      System.out.print("Please enter the price: ");
      tempPrice = EasyScanner.nextDouble();
      carListIn.add(new Car(tempReg, tempMake, tempPrice));
   }

   /* method for removing a car from the list - in a real application this would need to include
      some validation */
   static void removeCar(List<Car> carListIn)
   {
      int pos;
      System.out.print("Enter the position of the car to be removed: ");
      pos = EasyScanner.nextInt();
      carListIn.remove(pos - 1);
   }

   // method for listing details of all cars in the list
   static void listAll(List<Car> carListIn)
```

```
    {
        for(Car item : carListIn)
        {
            System.out.println(item.getRegistration()
            + " "
            + item.getMake()
            + " "
            + item.getPrice());
        }
    }

    // method for writing the file
    static void writeList(List<Car> carListIn)
    {
        // use try-with-resources to ensure file is closed safely
        try(
                /* create a FileWriter object, carFile, that handles the low-level details of writing
                the list to a file which we have called "Cars.txt" */
                FileWriter carFile = new FileWriter("Cars.txt");
                /* now create a PrintWriter object to wrap around carFile; this allows us to user
                high-level functions such as println */
                PrintWriter carWriter = new PrintWriter(carFile);
            )
        {
            // write each element of the list to the file
            for(Car item : carListIn)
            {
                carWriter.println(item.getRegistration());
                carWriter.println(item.getMake());
                carWriter.println(item.getPrice()); // println can accept a double, and write it as a string
            }
        }
        // handle the exception thrown by the FileWriter methods
        catch(IOException e)
        {
            System.out.println("There was a problem writing the file");
        }
    }

    // method for reading the file
    static void readList(List<Car> carListIn)
    {
        String tempReg;
        String tempMake;
        String tempStringPrice;
        double tempDoublePrice;

        // use try-with-resources to ensure file is closed safely
        try(
                    /* create a FileReader object, carFile, that handles the lowlevel details of reading
                    the list from the "Cars.txt" file */
                    FileReader carFile = new FileReader("Cars.txt");
                    /* now create a BufferedReader object to wrap around carFile; this allows us to user
                    high-level functions such as readLine */
                    BufferedReader carStream = new BufferedReader(carFile);
            )
        {
            // read the first line of the file
            tempReg = carStream.readLine();
            /* read the rest of the first record, then all the rest of the records until the end of
               the file is reached */
            while(tempReg != null) // a null string indicates end of file
            {
                tempMake = carStream.readLine();
                tempStringPrice = carStream.readLine();
                // as this is a text file we have to convert the price to double
                tempDoublePrice = Double.parseDouble(tempStringPrice);
                carListIn.add(new Car(tempReg, tempMake, tempDoublePrice));
                tempReg = carStream.readLine();
            }
        }

        // handle the exception that is thrown by the FileReader constructor if the file is not found
        catch(FileNotFoundException e)
        {
            System.out.println("\nNo file was read");
        }

        // handle the exception thrown by the FileReader methods
        catch(IOException e)
        {
            System.out.println("\nThere was a problem reading the file");
        }
    }
}
```

It is only the `writeList` and `readList` methods that we need to analyse here—none of the other methods involves anything new. Let's start with `writeList`:

```
// method for writing the file
static void writeList(List<Car> carListIn)
{
   // use try-with-resources to ensure file is closed safely
   try(
         /* create a FileWriter object, carFile, that handles the low-level details of writing
         the list to a file which we have called "Cars.txt" */
         FileWriter carFile = new FileWriter("Cars.txt");
         /* now create a PrintWriter object to wrap around carFile; this allows us to user
         high-level functions such as println */
         PrintWriter carWriter = new PrintWriter(carFile);
      )
   {
      // write each element of the list to the file
      for(Car item : carListIn)
      {
         carWriter.println(item.getRegistration());
         carWriter.println(item.getMake());
         carWriter.println(item.getPrice()); // println can accept a double, and write it as a string
      }
   }
   // handle the exception thrown by the FileWriter methods
   catch(IOException e)
   {
      System.out.println("There was a problem writing the file");
   }
}
```

The first thing to notice is that we have enclosed everything in a **try** block. Here we are creating our files using the *try-with-resources* mechanism that was introduced to you in Chap. 14. We use this because the file has to be closed after we finish using it. Closing the file achieves two things. First, it ensures that a special character, the end-of-file marker,[1] is written at the end of the file. This enables us to detect when the end of the file has been reached when we are reading it—more about this when we explore the `readList` method. Second, closing the file means that it is no longer accessible by the program, and is therefore not susceptible to being written to in error.

Using *try-with-resources* ensures that the file is always closed, no matter what other errors or exceptions may have occurred—before the advent of *try-with-resources* we would have had to specifically close the file by calling the `close` method of `PrintWriter` (this would have been done in a **finally** clause). As you can see, the instructions for opening the file have been placed in the brackets after the **try** keyword. We will use *try-with-resources* to create and open files throughout this chapter.

The file is opened by using a class called `FileWriter`; this is one of the classes we talked about earlier that provide the low-level communication between the program and the file. By opening a file we establish a *stream* through which we can output data to the file. We create a `FileWriter` object, `carFile`, giving it the name of the file to which we want to write the data:

```
FileWriter carFile = new FileWriter("Cars.txt");
```

In this case we have called the file `Cars.txt`.[2] Creating the new `FileWriter` object causes the file to be opened in output mode—meaning that it is ready to receive data; if no file of this name exists then one will be created. Opening the file in this way (in output mode) means that any data that we write to the file will

[1]Most systems use Unicode character 26 as the end-of-file marker.
[2]As we have not supplied an absolute pathname, the file will be saved in the current directory.

wipe out what was previously there. That is what we need for this particular application, because we are simply going to write the entire list when the program terminates. Sometimes, however, it is necessary to open a file in **append** mode; in this mode any data written to the file would be written after the existing data. To do this we would simply have used another constructor, which takes an additional (**boolean**) parameter indicating whether or not we require append mode:

```
FileWriter carFile = new FileWriter("Cars.txt", true);
```

The next thing we do is create an object, carWriter, of the PrintWriter class, sending it the carFile object as a parameter.

```
PrintWriter carWriter = new PrintWriter(carFile);
```

This object can now communicate with our file via the carFile object; PrintWriter objects have higher level methods than FileWriter objects (for example print and println) that enable us to write whole strings like we do when we output to the screen.

Now we are ready to write each Car in the list to our file—we can use a **for** loop for this:

```
for(Car item : carListIn)
{
   carWriter.println(item.getRegistration());
   carWriter.println(item.getMake());
   carWriter.println(item.getPrice());
}
```

On each iteration we use the println method of our PrintWriter object, carWriter, to write the registration number, the make and the price of the car to the file; println converts the price to a String before writing it. Notice also that the println method inserts a newline character at the end of the string that it prints; if we did not want the newline character to be inserted, we would use the print method instead.

Finally we have to handle any IOExceptions that may be thrown by the FileWriter methods:

```
catch(IOException e)
{
   System.out.println("There was a problem writing the file");
}
```

In a moment we will explore the code for reading the file. But bear in mind that if we were to run our program and add a few records, and then quit the program we would have saved the data to a text-file called Cars.txt, so we should be able to read this file with a text editor. When we did this, we created three cars, and then looked inside the file using Windows™ Notepad. Figure 18.1 shows the result.

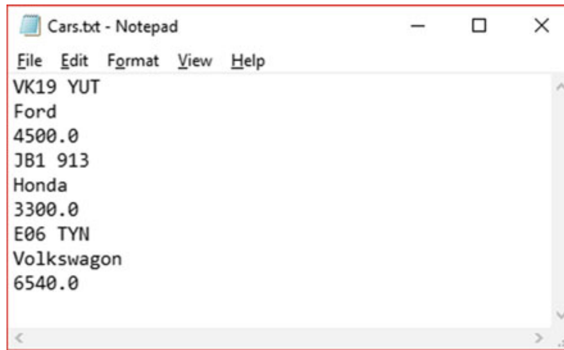


Fig. 18.1 Viewing a text file with windows notepad

As we have written each field using the `println` statement, each one, as you can see, starts on a new line. If our aim were to view the file with a text editor as we have just done, then this might not be the most suitable format—we might, for example, have wanted to have one record per line; we could also have printed some headings if we had wished. However, it is actually our intention to make our program read the entire file into our list when the program starts—and as we shall now see, one field per line makes reading the text file nice and easy. So let's take a look at our `readList` method:

```
// method for reading the file
static void readList(List<Car> carListIn)
{
    String tempReg;
    String tempMake;
    String tempStringPrice;
    double tempDoublePrice;

    // use try-with-resources to ensure file is closed safely
    try(
              /* create a FileReader object, carFile, that handles the lowlevel details of reading
              the list from the "Cars.txt" file */
              FileReader carFile = new FileReader("Cars.txt");
              /* now create a BufferedReader object to wrap around carFile; this allows us to user
              high-level functions such as readLine */
              BufferedReader carStream = new BufferedReader(carFile);

       )
    {
        // read the first line of the file
        tempReg = carStream.readLine();
        /* read the rest of the first record, then all the rest of the records until the end of
          the file is reached */
        while(tempReg != null) // a null string indicates end of file
        {
            tempMake = carStream.readLine();
            tempStringPrice = carStream.readLine();
            // as this is a text file we have to convert the price to double
            tempDoublePrice = Double.parseDouble(tempStringPrice);
            carListIn.add(new Car(tempReg, tempMake, tempDoublePrice));
            tempReg = carStream.readLine();
        }
    }

    // handle the exception that is thrown by the FileReader constructor if the file is not found
    catch(FileNotFoundException e)
    {
        System.out.println("\nNo file was read");
    }

    // handle the exception thrown by the FileReader methods
    catch(IOException e)
    {
        System.out.println("\nThere was a problem reading the file");
    }
}
```

First, we have declared some variables to hold the value of each field as we progressively read through the file. Remembering that this is a text file we have declared three `Strings`:

```
String tempReg;
String tempMake;
String tempStringPrice;
```

The last of these will have to be converted to a **`double`** before we store it in the list so we also need a variable to hold this value once it is converted:

```
double tempDoublePrice;
```

Now, as before, we put everything into a **`try`** block, as we are going to have to deal with the exceptions that may be thrown by the various methods we will be using. Again we are using *try-with-resources*, so the process of opening the file is place in the brackets after the **`try`**. As you can see from the code, we start by creating an object—`carFile`—of the class `FileReader` which deals with the low-level details involved in the process of reading a file. The name of the file, `Cars.txt`, that we wish to read is sent in as a parameter to the constructor; this file is then opened in read mode.

```
FileReader carFile = new FileReader("Cars.txt");
```

Now, in order that we can use some higher-level read methods, we wrap up our `carFile` object in an object of a class called `BufferedReader`. We have called this new object `carStream`.

```
BufferedReader carStream = new BufferedReader(carFile);
```

Now we are going to read each field of each record in turn, so we will need some sort of loop. The only problem is to know when to stop—this is because the number of records in the file can be different each time we run the program. There are different ways in which to approach this problem. One very good way (although not the one we have used here), if the same program is responsible for both reading and writing the file, is simply to write the total number of records as the first item when the file is written. Then, when reading the file, this item is the first thing to be read —and once this number is known a **`for`** loop can be used to read the records.

However, it may well be the case that the file was written by another program (such as a text editor). In this case it is necessary to check for the end-of-file marker that we spoke about earlier. In order to help you understand this process we are using this method here, even though we could have used the first (and perhaps simpler) method.

This is what we have to do: we have to read the first field of each record, then check whether that field began with the end-of-file marker. If it did, we must stop reading the file, but if it didn't we have to carry on and read the remaining fields of that record. Then we start the process again for the next record.

Some pseudocode should make the process clear; we have made this pseudocode specific to our particular example:

```
BEGIN
   READ the registration number field of the first record
   LOOP while the field just read does not contain the end-of-file marker
   BEGIN
      READ the make field of the next record
      READ the price field of the next record
      CONVERT the price to a double
      CREATE a new car with details just read and add it to the list
      READ the registration number field of the next record
   END
END
```

The code for this is shown below:

```
tempReg = carStream.readLine();
while(tempReg != null) // a null string indicates end of file
{
   tempMake = carStream.readLine();
   tempStringPrice = carStream.readLine();
   tempDoublePrice = Double.parseDouble(tempStringPrice);
   carListIn.add (new Car(tempReg, tempMake, tempDoublePrice));
   tempReg = carStream.readLine();
}
```

Notice that we are using the `readLine` method of `BufferedReader` to read each record. This method reads a line of text from the file; a line is considered anything that is terminated by the newline character. The method returns that line as a `String` (which does not include the newline character). However, if the line read consists of the end-of-file marker, then `readLine` returns a **`null`**, making it very easy for us to check if the end of the file has been reached. In Sect. 18.7 you will be able to contrast this method of `BufferedReader` with the `read` method, which reads a single character only.

Finally, we must handle any exceptions that may be thrown by the methods of `FileReader`; first, the constructor throws a `FileNotFoundException` if the file is not found:

```
catch(FileNotFoundException e)
{
   System.out.println("\nNo file was read");
}
```

All the other methods may throw `IOExceptions`:

```
catch(IOException e)
{
   System.out.println("\nThere was a problem reading the file");
}
```

18.6 Reading and Writing to Binary Files

In many ways, it makes little difference whether we store our data in text format or binary format; but it is, of course, important to know the sort of file that we are dealing with when we are reading it. For example, in the previous section you saw that we needed to convert a `String` to a **`double`** when it came to handling the price of a car. However, it is important for you to be familiar with the ways of handling both types of file, so now we will show you how to read and write data to a binary file using exactly the same example as before.

The only difference in our program will be the `writeList` and `readList` methods. First let's look at the code for the new `writeList` method:

```
static void writeList(List<Car> carListIn)
{
       // use try-with-resources to ensure file is closed safely
       try(
             FileOutputStream carFile = new FileOutputStream("Cars.bin");
             DataOutputStream carWriter = new DataOutputStream(carFile);
          )
       {

          for(Car item : carListIn)
          {
             carWriter.writeUTF(item.getRegistration());
             carWriter.writeUTF(item.getMake());
             carWriter.writeDouble(item.getPrice());
          }

       }

       catch(IOException e)
       {
          System.out.println("There was a problem writing the file");
       }
```

You can see that the process is similar to the one we used to write a text file, but here the two classes that we are using are `FileOutputStream` and `DataOutputStream` which deal with the low-level and high-level processes respectively. The `DataOutputStream` class provides methods such as `writeDouble`, `writeInt` and `writeChar` for writing all the basic scalar types, as well a method called `writeUTF` for writing strings. UTF stands for *Unicode Transformation Format*, and the method is so-called because, when it writes the string to a file, it converts the Unicode characters (which are used in Java) to the machine-specific format.

Before moving on to the `readList` method it is worth reminding ourselves that a file written in this way—that is, a binary file—cannot be read by a text editor. And to prove the point, Fig. 18.2 shows the result of trying to read such a file in Windows™ Notepad.

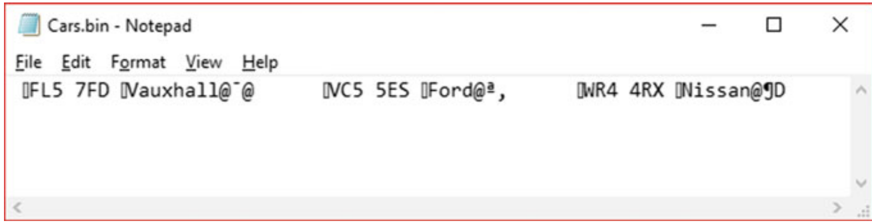


Fig. 18.2 Trying to read a binary file with a text editor

So now we can look at the `readList` method:

```
static void readList(List<Car> carListIn)
{
    String tempReg;
    String tempMake;
    double tempPrice;
    boolean endOfFile = false;

    // use try-with-resources to ensure file is closed safely
    try(
          FileInputStream carFile = new FileInputStream("Cars.bin");
          DataInputStream carStream = new DataInputStream(carFile);
       )
    {
       while(endOfFile == false)
       {
          try
          {
             tempReg = carStream.readUTF();
             tempMake = carStream.readUTF();
             tempPrice = carStream.readDouble();
             carListIn.add(new Car(tempReg, tempMake, tempPrice));
          }
          catch(EOFException e)
          {
             endOfFile = true;
          }
       }

    }
    catch(FileNotFoundException e)
    {
       System.out.println("\nThere are currently no records");
    }

    catch(IOException e)
    {
       System.out.println("There was a problem reading the file");
    }
}
```

You can see that the two classes we use for reading binary files are `FileInputStream` for low-level access and `DataInputStream` for the higher-level functions; they have equivalent methods to those we saw previously when writing to files.

The most important thing to observe in this method is the way we test whether we have reached the end of the file. In the case of a binary file we can do this by making use of the fact that the `DataInputStream` methods throw `EOFExceptions` when an end of file marker has been detected during a read operation. So all we have to do is declare a **`boolean`** variable, `endOfFile`, which we initially set to **`false`**, and we use this as the termination condition in the **`while`** loop. Then we enclose our read operations in a **`try`** block, and, when an exception is thrown, `endOfFile` is set to **`true`** within the **`catch`** block, causing the **`while`** loop to terminate.

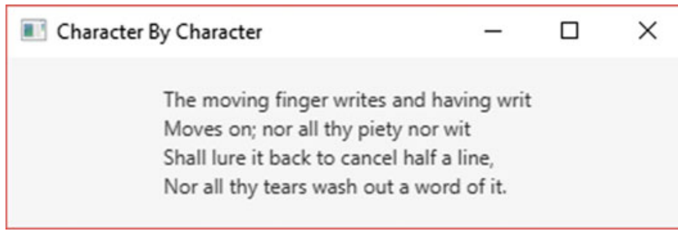


Fig. 18.3 Reading a file character by character

18.7 Reading a Text File Character by Character

As you will have realized by now, there are many ways in which we can deal with handling files, and the methods we choose will depend largely on what it is we want to achieve.

In this section we will show you how to read a text file character by character—this is a useful technique if we do not know anything about the structure of the file.

We have written a JavaFX application which reads a text file, `Poem.txt`, character by character, and builds a string by appending each character as it is read. The process continues until the end of the file is reached, or until a stipulated number of characters have been read. We put this last condition in as a safeguard in case the user should try to display a very large file by mistake.

Once the reading process has finished, the string is displayed by adding it to a label, as shown in Fig. 18.3.

Here is the code for the application:

CharacterByCharacter

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CharacterByCharacter extends Application
{
    private Label viewArea = new Label();

    @Override
    public void start(Stage stage)
    {
        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(viewArea);
        Scene scene = new Scene(root, 400, 100);
        stage.setScene(scene);
        stage.setTitle("Character By Character");
        stage.show();

        readAndWrite();
    }
    private void readAndWrite()
```

```
    {
        // use try-with-resources to ensure file is closed safely
        try(
              FileReader testFile = new FileReader("Poem.txt");
              BufferedReader textStream = new BufferedReader(testFile);
           )
        {
           String str = new String();
           final int MAX = 1000;

           int ch; // to hold the integer (Unicode) value of the character
           char c; // to hold the character when type cast from integer
           int counter = 0; // to count the number of characters read so far
           ch = textStream.read(); // read the first character from the file
           c = (char) ch; // type cast from integer to character
           /* continue through the file until either the end of the file is
              reached (in which case -1 is returned) or the maximum number of
              characters stipulated have been read */
           while( ch != -1 && counter <= MAX)
           {
              counter++; // increment the counter
              str = str + c;
              ch = textStream.read(); // read the next character
              c = (char) ch;
           }

           str = str + "\n";
           viewArea.setText(str);
        }

        catch(IOException ioe)
        {
           viewArea.setText("There was a problem reading the file\n");
        }
    }

    public static void main(String[] args)
    {
           launch(args);
    }

}
```

As you can see, we have placed the functionality in a helper method, `read-AndWrite`. The main thing to notice here is that we are using the `read` method of `BufferedReader`; this method reads a single character from the file and returns an integer, the Unicode value of the character read. If the character read was the end-of-file marker then it returns −1, making it an easy matter for us to check whether the end of the file has been reached. In the above example, as explained earlier, we stop reading the file if we have reached the end or if more than the maximum number of characters allowed has been read; here we have set that maximum to 1000. You can see that in the above method, after each read operation, we type cast the integer to a character, which we then append to the string.

18.8 Object Serialization

If you are going to be dealing with files that will be accessed only within a Java program, then one of the easiest ways to do this is to make use of two classes called `ObjectInputStream` and `ObjectOutputStream`. These classes have methods called, respectively, `readObject` and `writeObject` that enable us to read and write whole objects from and to files. The process of converting an object into a stream of data suitable for storage on a disk is called **serialization**.

Any class whose objects are to be read and written using the above methods must implement the interface `Serializable`. This is a type of interface that we have not actually come across before—it is known as a **marker** and in fact contains

no methods. Its purpose is simply to make an "announcement" to anyone using the class; namely that objects of this class can be read and written as whole objects. In designing a class we can, then, choose not to make our class `Serializable`—we might want to do this for security reasons (for example, to stop whole objects being transportable over the Internet) or to avoid errors in a distributed environment where the code for the class was not present on every machine.

In the case of our `Car` class, we therefore need to declare it in the following way before we could use it in a program that handles whole objects:

```
public class Car implements Serializable
```

The `Serializable` interface resides within the `java.io` package.

Now we can re-write the `writeList` and `readList` methods of `TextFileTester` so that we manipulate whole objects. First the `writeList` method:

```
static void writeList(List<Car> carListIn)
{
     // use try-with-resources to ensure file is closed safely
     try(
             FileOutputStream carFile = new FileOutputStream("Cars.dat");
             ObjectOutputStream carStream = new ObjectOutputStream(carFile);
        )
     {

        for(Car item : carListIn)
        {
            carStream.writeObject(item);
        }

     }
     catch(IOException e)
     {
        System.out.println("There was a problem writing the file");
     }
}
```

You can see how easy this is—you just need one line to save a whole object to a file by using the `writeObject` method of `ObjectOutputStream`.

Now the `readList` method:

```
static void readList(List<Car> carListIn)
{
     boolean endOfFile = false;
     Car tempCar;

     // use try-with-resources to ensure file is closed safely
     try(
          // create a FileInputStream object, carFile
          FileInputStream carFile = new FileInputStream("Cars.dat");
          // create an ObjectInputStream object to wrap around carFile
          ObjectInputStream carStream = new ObjectInputStream(carFile);
        )
     {

        // read the first (whole) object with the readObject method
        tempCar =  (Car) carStream.readObject();
        while(endOfFile != true)
        {
            try
             {
                 carListIn.add(tempCar);
                 // read the next (whole) object
                 tempCar = (Car) carStream.readObject();
             }

            /* use the fact that readObject throws an EOFException to
               check whether the end of the file has been reached */
```

```
        catch(EOFException  e)
        {
           endOfFile = true;
        }
    }

}
        catch(FileNotFoundException e)
        {
           System.out.println("\nNo file was read");
        }

        catch(ClassNotFoundException e) // thrown by readObject
        {
           System.out.println("\nTrying to read an object of an unknown class");
        }

        catch(StreamCorruptedException e) // thrown by the constructor
        {
           System.out.println("\nUnreadable file format");
        }

        catch(IOException e)
        {
           System.out.println("There was a problem reading the file");
        }
```

Again you can see how easy this is—a whole object is read with the `readObject` method.

We should draw your attention to a few of the exception handling routines we have used here—first notice that we have once again made use of the fact that `readObject` throws an `EOFException` to check for the end of the file. Second, notice that `readObject` also throws a `ClassNotFoundException`, which indicates that the object just read does not correspond to any class known to the program. Finally, the constructor throws a `StreamCorruptedException`, which indicates that the input stream given to it was not produced by an `ObjectOutputStream` object—underlining the fact that reading and writing whole objects are complementary techniques that are specific to Java programs.

One final thing to note—if an attribute of a `Serializable` class is itself an object of another class, then that class too must be `Serializable` in order for us to be able to read and write whole objects as we have just done. You will probably have noticed that in the case of the `Car` class, one of its attributes is a `String`—fortunately the `String` class does indeed implement the `Serializable` interface, which is why we had no problem using it in this way in our example.

Before moving on, it is worth noting that all the Java collection classes such as `HashMap` and `ArrayList` are themselves `Serializable`.

18.9 Random Access Files

All the programs that we have looked at so far in this chapter have made use of serial access. For small applications this will probably be all you need—however, if you were writing applications that handled very large data files it would be desirable to use random access methods. Fortunately Java provides us with this facility.

Table 18.1 Size of the primitive types

byte	1 byte
short	2 bytes
char	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
boolean	1 bit[3]

The class that we need is called `RandomAccessFile`. This enables us to open a file for random access. Random access files can be opened in either read–write mode or read-only mode; the constructor therefore takes, in addition to the name of the file, an additional `String` parameter which can be either "`rw`" or "`r`", indicating the mode in which the file is to be opened.

In addition to methods similar to those of the `DataOutputStream` class (such as `writeUTF`, `readDouble` and so on), `RandomAccessFile` has a method called `seek`. This takes one attribute, of type `long`, which indicates how many bytes to move the file-pointer before starting a read or write operation.

So now we have the question of how far to move the pointer—we need to be able to calculate the size of each record. If we are dealing only with primitive types, this is an easy matter. These types all take up a fixed amount of storage space, as shown in Table 18.1 overleaf.

The difficulty comes when a record contains `Strings`, as is commonly the case. The size of a `String` object varies according to how many characters it contains. What we have to do is to restrict the length of each string to a given amount; let's take the `Car` class as an example. The data elements of any `Car` object consist of two `Strings` and a **`double`**. We will make the decision that the two `String` attributes—registration number and make—will be restricted to 10 characters only. Now, any `String` variable will always take up one byte for each character, plus two extra bytes (at the beginning) to hold an integer representing the length of the `String`. So now we can calculate the maximum amount of storage space we need for a car as follows:

registration (String)	12 bytes
make (String)	12 bytes
price (double)	8 bytes
Total	**32 bytes**

This still leaves us with one problem—what if the length of one of the `String` attributes entered is actually *less* than 10? The best way to deal with this is to pad the string out with spaces so that it always contains *exactly* 10 characters. This

[3] Allow for 1 byte when calculating storage space.

means that the size of every `Car` object will always be exactly 32 bytes—you will see how we have done this when you study program below, `RandomFileTester`. This program uses a rather different approach to the one we have used so far in this chapter. Two options (as well as a *Quit* option) are provided. The first, the option to add a car, simply adds the car to the end of the file. The second, to display the details of a car, asks the user for the position of the car in the file then reads this record directly from the file. You can see that there is now no need for a `List` in which to store the cars.

Study the program carefully—then we will discuss it. Note that we have made use of our `EasyScanner` class here.

RandomFileTester

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomFileTester
{
   static final int CAR_SIZE = 32; // each record will be 32 bytes

   public static void main(String[] args)
   {
      char choice;
      do
      {
         System.out.println("\nRandom File Tester");
         System.out.println("1. Add a car");
         System.out.println("2. Display a car");
         System.out.println("3. Quit\n");
         choice = EasyScanner.nextChar();
         System.out.println();
         switch(choice)
         {
            case '1' :addCar();
                      break;
            case '2' :displayCar();
                      break;
            case '3' :break;
            default : System.out.print("\nChoose 1 - 3 only please\n ");
         }
      }while(choice != '3');
   }

   static void addCar()
   {
      String tempReg;
      String tempMake;
      double tempPrice;
      System.out.print("Please enter the registration number: ");
      tempReg = EasyScanner.nextString();

      if(tempReg.length() > 10) //limit the registration number to 10 characters
      {
         System.out.print("Ten characters only - please re-enter: ");
         tempReg = EasyScanner.nextString();
      }
      // pad the string with spaces to make it exactly 10 characters long
      for(int i = tempReg.length() + 1 ; i <= 10 ; i++)
      {
         tempReg = tempReg.concat(" ");
      }

      // get the make of the car from the user
      System.out.print("Please enter the make: ");
      tempMake = EasyScanner.nextString();

      // limit the make number to 10 characters
      if(tempMake.length() > 10)
      {
         System.out.print("Ten characters only - please re-enter: ");
         tempMake = EasyScanner.nextString();
      }
      // pad the string with spaces to make it exactly 10 characters long
      for(int i = tempMake.length() + 1; i <= 10; i++)
      {
         tempMake = tempMake.concat(" ");
      }
```

```
        // get the price of the car from the user
        System.out.print("Please enter the price: ");
        tempPrice = EasyScanner.nextDouble();

        // write the record to the file
        writeRecord(new Car(tempReg, tempMake, tempPrice));
    }

    static void displayCar()
    {
        int pos;
        // get the position of the item to be read from the user
        System.out.print("Enter the car's position in the list: ");
        pos = EasyScanner.nextInt(); // read the record requested from file
        Car tempCar = readRecord(pos);
        if(tempCar != null)
        {
            System.out.println(tempCar.getRegistration().trim()
            + " "
            + tempCar.getMake().trim()
            + " "
            + tempCar.getPrice());
        }
        else
        {
            System.out.println("Invalid postion") ;
        }
    }

    static void writeRecord(Car tempCar)
    {
        // use try-with-resources to ensure file is closed safely
        try(
                // open a RandomAccessFile in read-write mode
                RandomAccessFile carFile = new RandomAccessFile("Cars.rand", "rw");
          )
       {

          // move the pointer to the end of the file
          carFile.seek(carFile.length());
          // write the three fields of the record to the file
          carFile.writeUTF(tempCar.getRegistration());
          carFile.writeUTF(tempCar.getMake());
          carFile.writeDouble(tempCar.getPrice());
       }
       catch(IOException e)
       {
          System.out.println("There was a problem writing the file");
       }
   }

   static Car readRecord(int pos)
   {
        String tempReg;
        String tempMake;
        double tempPrice;
        Car tempCar = null; // a null value is returned if there is a problem reading the record

        // use try-with-resources to ensure file is closed safely
        try(
                // open a RandomAccessFile in read-only mode
                RandomAccessFile carFile = new RandomAccessFile("Cars.rand","r");
           )
        {

           // move the pointer to the start of the required record
           carFile.seek((pos-1) * CAR_SIZE);
           // read the three fields of the record from the file
           tempReg = carFile.readUTF();
           tempMake = carFile.readUTF();
           tempPrice = carFile.readDouble();
           // use the data just read to create a new Car object
           tempCar = new Car(tempReg, tempMake, tempPrice);
        }
        catch(FileNotFoundException e)
        {
           System.out.println("\nNo file was read");
        }

        catch(IOException e)
        {
           System.out.println("There was a problem reading the file");
        }
        // return the record that was read
        return tempCar;
    }
}
```

You can see that in the addCar method we have called writeRecord with a Car object as a parameter. Let's take a closer look at the writeRecord method. First the line to open the file in read-write mode:

```
RandomAccessFile carFile = new RandomAccessFile("Cars.rand", "rw");
```

Now the instruction to move the file pointer:

```
carFile.seek(carFile.length());
```

You can see how we use the seek method to move the pointer a specific number of bytes; here the correct number of bytes is the size of the file (as we want to write the new record at the end of the file), so we use the length method of RandomAccessFile to determine this number.

Now we can move on to look at the readRecord method. You can see that this is called from within the displayCar method, with an integer parameter, representing the position of the required record in the file.

The file is opened in read-only mode:

```
RandomAccessFile carFile = new RandomAccessFile("Cars.rand","r");
```

Then the seek method of RandomAccessFile is invoked as follows:

```
carFile.seek((pos-1) * CAR_SIZE);
```

You can see that the number of bytes through which to move the pointer has been calculated by multiplying the size of the record by one less than the position. This is because in order to read the first record we don't move the pointer at all; in order to read the second record we must move it 1×32 bytes; for the third record 2×32 bytes; and so on.

The final thing to note about the program is that in the displayCar method we have used the trim method of String to get rid of the extra spaces that we used to pad out the first two fields of the record.

Here is a test run from the program (starting off with an empty file):

```
Random File Tester
1. Add a car
2. View a car
3. Quit

1

Please enter the registration number: R54 HJK
Please enter the make: Vauxhall
Please enter the price: 7000

Random File Tester
1. Add a car
2. View a car
3. Quit

1

Please enter the registration number: T87 EFU
Please enter the make: Nissan
Please enter the price: 9000

Random File Tester
1. Add a car
2. View a car
3. Quit

2

Enter the car's position in the list: 2
T87 EFU Nissan 9000.0
```

18.10 Self-test Questions

1. Explain the principles of *input* and *output* and identify different input and output devices.
2. What is meant by the term *input/output stream?*
3. Distinguish between *text*, *binary* and *object encoding* of data.
4. Explain why we have used the *try-with-resources* construct throughout this chapter to create and open files.
5. The `TextFileTester` of Sect. 18.5 is to be adapted so that the user is simply asked to enter a number of cars, which, when that process is finished, saves those cars to a text file. The program then terminates. The file does not have to be read from within the program, but should be able to be read by a text editor such as Windows™ Notepad. The format should be as follows:

```
Cars2.txt - Notepad
File Edit Format View Help
Registration Number:  A297 ABF
Make: Ford
Price: 4560.0

Registration Number: U423 GAX
Make: Vauxhall
Price: 5999.0

Registration Number: T945 KMN
Make: Citroen
Price: 3795.0
```

Adapt the `writeList` method accordingly.

Hint: remember that a blank line is obtained by calling `println` *with no parameters.*

6. What is the difference between *serial access* files and *random access* files?

7. Explain the purpose of the `Serializable` interface.

8. Calculate the number of bytes required to store an object of a class, the attributes of which are declared as follows:

```
private int x;
private char c;
private String s;
```

You can assume that the `String` attribute will always consists of exactly 20 characters.

18.11 Programming Exercises

You will need to have access to the `Car` *class, the source code for which is available on the website. Or you can simply copy it from this chapter. The source code for the programs in the chapter is also on the website.*

1. Run the `TextFileTester` from Sect. 18.5 then adapt it so that it handles binary files, as described in Sect. 18.6.

2. Adapt the `TextFileTester` so that it behaves in the way described in question 5 of the self-test questions.

3. Implement the `CharacterByCharacter` program from Sect. 18.7, using a text file that you have created.

4. Adapt the `TextFileTester` so that it uses object encoding, as explained in Sect. 18.8 (don't forget that the `Car` class must implement the `Serializable` interface).

5. Adapt the `Bank` application of Chap. 8 so that it keeps permanent records using text encoding.

6. In Chap. 12 the case study made use of a file called `TenantFileHandler` that we wrote for you. This is available on the website. Study this class carefully, so that you are sure you understand it, then modify it so that it uses:

   (a) text encoding;

   (b) object encoding.

7. Adapt the `Library` application of Chap. 15 so that it keeps permanent records using object encoding.

Check for updates

Packages 19

Outcomes:

By the end of this chapter you should be able to:

- *identify the role of* ***packages*** *in organizing classes*;
- *create and deploy their own packages in Java*;
- *access classes residing in their own packages*;
- *run Java applications from the command line*;
- *deploy Java applications as* ***JAR*** *files*;
- *access libraries that are not part of the standard Java framework to access data on a remote database using the* ***Java Database Connectivity (JDBC)*** *and* ***Hibernate ORM*** *technologies.*

19.1 Introduction

From as early as Chap. 2 of this book you have been familiar with the idea of a *package*, and have been using packages in order to access classes residing in external folders. In this chapter we will take a more in-depth look at Java's package concept and see how you can deploy your own applications.

19.2 Understanding Packages

A **package**, in Java, is a *named collection* of *related classes*. You have already been using packages to access pre-written classes. For example, to store objects in an ordered collection you made use of the `ArrayList` class, which resides in the `util` package. To read and write to files you used classes in the `io` package. To organise the layout of your JavaFX applications you used classes such as `Scene`

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_19

and Group from the scene package. Giving meaningful names to a set of related classes in this way makes it easy for programmers to locate these classes when required. Packages can themselves contain other packages. For example, as well as containing classes for organising the layout of your JavaFX applications, the scene

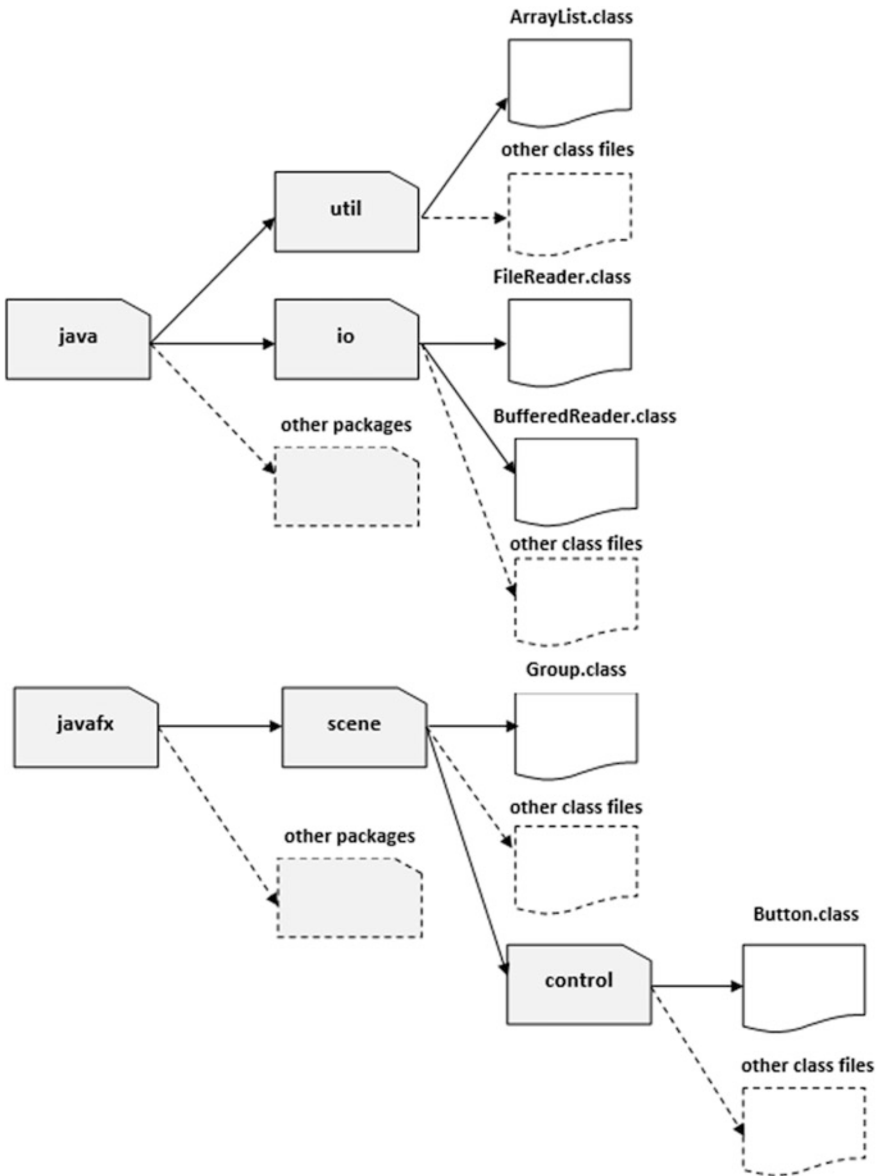


Fig. 19.1 A sample of the java package hierarchy

package also contains the control package, which contains JavaFX control classes such as Button and Label, since this group of classes is still logically related to JavaFX's collection of scene related files.

The package name actually corresponds to the *name of the directory* (or folder as some operating systems call it) in which all the given classes reside. All the core Java packages themselves reside in a global Java directory, named simply java. This directory is not itself a package but a store for other packages. Since Java was launched a few additional global directories have been developed. In particular, the javafx directory contains all the packages and classes required for JavaFX development. Figure 19.1 illustrates part of this hierarchy of packages.

As you can see from Fig. 19.1, packages contain class files (that is the compiled Java byte code), not source files (the original Java instructions). This means the location of the original Java source files is unimportant here. They may be in the same directory as the class files, in another directory or, as in the case of the predefined Java packages, they may even no longer be available.

19.3 Accessing Classes in Packages

Suppose you are writing the code for a new class. You will recall how you can give it access to a class contained within a package. Just referencing the class won't work. For example, let's assume a class you are writing needs a DecimalFormat object. The following will not compile:

```
public class SomeClass
{
   private DecimalFormat someFormatObject; // a problem here
}
```

This won't compile because the compiler won't be able to find a class called DecimalFormat. One way to tell the compiler where this class file resides is, as you already know, to add an **import** statement above the class. This class is in the text package so the following would be appropriate:

```
import java.text.DecimalFormat; // allows compiler to find the DecimalFormat.class file

public class SomeClass
{
   private DecimalFormat someFormatObject; // now this will compile
}
```

Can you see how the **import** statement matches the directory structure we illustrated in Fig. 19.1? Effectively the compiler is being told to look for the DecimalFormat class in the text directory (package), which in turn is in the java directory (whose location is already known to the Java run-time system). The

location of a file is often referred to as the **path** to that file. In the Windows operating systems this path would be expressed as follows:

```
java\text\
```

In other operating systems forward slashes may be used instead of backward slashes. The Java **`import`** statement simply expresses this path but uses dots instead of backward or forward slashes.

The asterisk notation, that we also met in Chap. 2, allows you to have access to *all* class files in the given package. Note that there can only ever be one '.*' in an **`import`** statement and the '.*' must follow a package name. However, as you have already seen in previous chapters, you can have as many **`import`** statements as you require. Here are some examples of valid and invalid **`import`** statements:

```
import java.*.*; // illegal as contains more than one '.*'
import java.*;   // illegal as 'java' is not a package
import java.text.*;  // fine, allows access to all classes in text package
import javafx.scene.control.Button; // fine, allows access to the Button class in control package
```

Although there is no overhead in allowing access to all files in a package via the asterisk notation, we have used the convention of explicitly listing every class imported individually in this book for the sake of clarity.

It is actually possible to access classes from within packages *without* the need for an **`import`** statement. To do this, references to any such classes must be appended onto the package name itself. Returning to the `DecimalFormat` example, we could have removed the **`import`** statement and referred to the package directly in the class as follows:

```
public class SomeClass
{
   /* appending the class name onto the package name avoids the need to import
      the given package */

   private java.text.DecimalFormat someFormatObject;
}
```

The package plus class name is in fact the proper name for this class. An **`import`** statement just provides us with a convenient shorthand so that we do not always have to include the package name with the class name. As you can imagine, having to append the class name onto the package name every time we use a class from a package would be very cumbersome, so the **`import`** statement is preferable. There are times, however, when the long name is necessary.

The long class name can be useful when the class name on its own clashes with the name of another class. For example, let us assume we have developed our own class called `Scanner`—perhaps as part of a warehouse application. Giving this name to the class is not a great idea as there already is a `Scanner` class in the `util` package, but it is possible to choose this name if we wish. Now, let us

assume that the constructor for this class takes a **`double`** value that represents the price of a product. The `ScannerApp1` program below requires both this newly developed `Scanner` class and Java's `Scanner` class in the `util` package.

This *ScannerApp1* program will not compile

```
import java.util.Scanner;// Java's Scanner class

// this program will not compile!
public class ScannerApp1
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in); // Java's Scanner class
        System.out.println("Enter price: ");
        double price = keyboard.nextDouble();
        // the next line will cause a compiler error
        Scanner product = new Scanner (price); // our own Scanner class
        // more code here
    }
}
```

As you can see, we are referencing two `Scanner` classes here: Java's `Scanner` class and our newly developed `Scanner` class. Not surprisingly, this will result in a compiler error! In the case of the `ScannerApp` program, the compiler will assume the correct `Scanner` file is the `Scanner` that has been imported and used to create a keyboard object:

```
Scanner keyboard = new Scanner(System.in); // this will compile ok
```

The second use of `Scanner`, to create an object called product from our own class, will then be the one that will cause the compiler error, as the compiler is expecting this `Scanner` class to be the `Scanner` class from the `util` package, which has no such constructor for receiving a double:

```
Scanner product = new Scanner (price);// this will cause a compiler error
```

To resolve this name clash, we can use the extended package name in the code to differentiate between the two classes. Let's see how to do this in the `ScannerApp2` program below:

This *ScannerApp2* program will compile

```
// note we do not import Scanner from util

public class ScannerApp2
{
    public static void main(String[] args)
    {
       // use full path name to Scanner file in util
       java.util.Scanner keyboard = new java.util.Scanner(System.in);
       System.out.println("Enter price: ");
       double price = keyboard.nextDouble();
       // the next line will not now cause a compiler error
       Scanner product = new Scanner (price); // our own Scanner class
       // more code here
    }
}
```

You can see the full path name has been used for Java's `Scanner` class in the code itself

```
java.util.Scanner keyboard = new java.util.Scanner(System.in);
```

Now we no longer have a name clash with our class and the complier can process references to both classes.

19.4 Developing Your Own Packages

You might be surprised to know that all the classes that you have developed so far already reside in a *single* package. This may seem strange as you didn't instruct the compiler to add your classes to any package. In fact, what actually happens is that if you don't specifically ask your classes to be put in a package, then they all get added to some large unnamed package.

In order to locate and deploy your class files easily, and avoid any name clashes in the future, it would be a good idea to use named packages to organize your classes.

As an example, let's go back to our *Hostel* application from Chaps. 11 and 12 and create a unique package in which to put our class files—we will call this package `hostelApp`.[1] To instruct the compiler that you wish to add the classes that make up this application into a package called `hostelApp`, simply add the following **`package`** command at the top of each of the original source files:

```
package hostelApp;
```

This line instructs the compiler that the class file created from this source file must be put in a package called `hostelApp`. Here, for example, is the `Payment` class with this **`package`** line added:

```
package hostelApp; // add this line to the top of the source file

public class Payment
{
   // as before
}
```

[1] We will stick to the standard Java convention of beginning package names with a lower-case letter.

When you compile this class you will find a directory called hostelApp will have been created and the resulting Payment.class file will be placed into this directory.[2]

All the classes that make up this *Hostel* application (such as Tenant, Hostel and so on) will need to be amended in a similar way:

1. Add the following line to the top of each source file

```
package hostelApp;
```

2. Ensure that the compiled class files are placed in the hostelApp directory.

Now, if we were developing applications in the future that wish to make use of any of the classes in our hostelApp package we could import them like classes from any other package. For example, the TestPackage program below imports the Payment class from the hostelApp package and then creates a Payment object before printing it on the screen:

TestPackage

```
import hostelApp.Payment; // import a class from the hostelApp

public class TestPackage
{
    public static void main(String[] args)
    {
        Payment p = new Payment ("January", 725); // access the Payment class
        System.out.println(p);// calls Payment's toString method
    }
}
```

19.5 Package Scope

Up until now we have declared all our classes to be **public**. This has meant they have been visible to all other classes. When we come to adding our classes into our own packages, this becomes particularly important. This is because *classes can be made visible outside of their package only if they are declared as* **public**. Unless they are declared as **public**, classes by default have what is known as **package** scope. This means that they are visible *only to other classes within the same package*.

[2]If you are developing a class from scratch that you wish to add into a package, your Java IDE can be used so that the package line is inserted into your code for you and the required directory structure created. If you are using your Java IDE to *revisit* classes previously written outside of a package (as in this example), you may need to ensure that the resulting directory structure is reflected in the project you are working in. Refer to your IDE's documentation for details about how to do this.

Not all classes in the package need be declared as **public**. Some classes may be part of the implementation only and the developer may not wish them to be made available to the client. These classes can have **package** scope instead of **public** scope. In this way, packages provide an extra layer of security for your classes.

In the case of our *Hostel* application, we might choose to make the Hostel class **public**, and keep all the other files required in this application hidden within the package by giving them package scope. To give a class package scope, just remove the **public** modifier from in front of the class declaration. For example, returning to the Payment class, we can give this package scope as follows:

```
package hostelApp; // this class is added into the package

class Payment // this class has package scope
{
   // as before
}
```

Now, when the hostelApp package is imported into another class, this other class has access only to the Hostel class; not to classes like Payment which are hidden in the package with package scope. This is demonstrated in the code fragment below:

```
import hostelApp.*; // this imports only the public classes in the package

public class SomeOtherClass
{
   // the next line will not compile as Payment is hidden in the hostelApp package
   Payment p = new Payment("January", 725);
}
```

19.6 Running Applications from the Command Line

Way back in Chap. 1 we discussed the process of compiling and running Java programs. If you remember, we said that if you are working within a Java IDE you will have simple icons to click in order to carry out these procedures. If, however, you are working from a command line, like a DOS prompt for example, you would use the **javac** command (followed by the name of the source file) to compile a source file and **java** (followed by the name of a class) to run an application.[3] As a simple example, here is the very first program we showed you back in Chap. 1:

[3]When installing Java an **environment variable** called PATH should automatically have been set so the operating system can locate the necessary Java tools to compile, run and deploy Java programs. If this has not been done refer to your operating system's instructions for setting this variable The PATH variable should point to the *bin* folder in your JDK folder, for example *C:\ProgramFiles\Java\jdk1.8.0_121\bin.*

HelloWorld

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println ("Hello world");
    }
}
```

Notice this program was not placed in a package. Assuming you are in the directory containing this source code file, you can compile this class using the `javac.exe` tool as follows:

javac HelloWorld.java

This will produce a Java class file (`HelloWorld.class`). Assuming you are now in the directory that contains this class file you can then run this program with the `java.exe` tool as follows:

java HelloWorld

As noted in Chap. 1, when running a class file you do not include the `.class` extension.

When you run a class that resides in a package you must amend this slightly. As an example, let's once again consider the *Hostel* application. When you run an application you must run the class that contains the main method. The appropriate class in our application is the JavaFX `Hostel` class. Remember, we have now added this class to a `hostelApp` package:

```
package hostelApp; // Hostel class part of the hostelApp package
// import statements here
public class Hostel extends Application
{
   // as before
}
```

If we were in the *hostelApp* folder that contained this class we could try running the `Hostel` class from the command line as follows:

java Hostel

Unfortunately, this won't work as the system won't be able to find a class of the given name. In order to run a class that is contained within a package you must append the class name onto the name of the package (with a '.' symbol). You will now need to be in the directory *above* the package directory. So in this case you will need to be in the directory above the *hostelApp* directory. You can then run the `Hostel` class by using the following command:

java hostelApp.Hostel`

If you do not wish to run the Java commands from the directories containing the relevant files you can set an environment variable called the **CLASSPATH** so that it points to the relevant directory (or directories) and then run these commands from any directory. See your operating system's documentation for how to do this.

Before we move on, let's just stop and have a look at the parameter that we always give to `main` methods:

```
public static void main(String[] args)
```

As you know, this means that `main` is given an array of `String` objects as a parameter. How are these `String` objects passed on to `main`? Up until now we have not discussed them at all. Well, values for these strings can be passed to `main` when you run the given class from the command line. Often, as in our previous program, there is no need to pass any such strings and this array of strings is effectively empty. Sometimes, however, it is useful to send in such parameters. They are sent to `main` from the command line by listing the strings, one after the other after the name of the class as follows:

java ClassName firstString secondString otherStrings

As you can see, the strings are separated by spaces. Any number of strings can be sent in this way. For example, if a program were called `ProcessNames`, two names could be sent to it as follows:

java ProcessNames Aaron Quentin

Notice that, were the strings to contain spaces, they must be enclosed in quotes:

java ProcessNames "Aaron Kans" "Quentin Charatan"

These strings will be placed into `main`'s array parameter (`args`), with the first string being at `args[0]`, the second at `args[1]` and so on. The number of strings sent to `main` is variable. The `main` method can always determine the number of strings sent by checking the `length` of the array (`args.length`). The `ProcessNames` class takes the array of strings and displays them on the screen.

ProcessNames

```
public class ProcessNames
{
    public static void main(String[] args)
    {
       if (args.length != 0)// check some arguments have been sent
       {
            // loop through all elements in the 'args' array
            for (int i=0; i<args.length; i++)
            {
                // access individual strings in array
                System.out.println("hello " + args[i]);
            }
       }
    }
}
```

Notice, to keep things simple we have not added this class to a package. We can run this program from the command line as follows:

java ProcessNames "Batman and Robin" Superman

Notice "`Batman and Robin`" needed to be surrounded by quotes as it has spaces in it, whereas `Superman` does not. Running this program would produce the obvious result:

```
hello Batman and Robin
hello Superman
```

19.7 Deploying Your Packages

A very common way of making your packages available to clients is to convert them to JAR files. A JAR file (short for Java Archive) has the extension `.jar` and is simply a compressed file. Most IDEs provide a means of creating JAR files but JAR files can also be created from the command prompt with the **`jar.exe`** tool. This tool is provided with the JDK and also with most standard IDEs. Assuming that we are in the directory above the `hostelApp` package then the correct statement to create a suitable JAR file is:

jar cvf hostel.jar hostelApp

As you can see there are various switches that are used with the `jar` program. The ones used above have the following effect:

c: create a new JAR file;
v: provide full (verbose) output to report on progress;
f: provide a name for the JAR file.

After these switches comes the name of the output file—`hostel.jar` in our case. Finally,we must list the files we wish to be included. In the above example we require only the package directory, `hostelApp`, but you may have additional files here such as sound and image files that are not part of the package.

If you are working in a graphics environment, and there is a JVM installed on your computer, then it is possible to create a JAR file that will run the program by double-clicking on its icon. We call such a JAR file an *executable* JAR file.

While it is possible to use Java tools from the command line to create such executable JAR files, as you could see when we showed you how to create a standard JAR file from the command line, this can be quite verbose and prone to error. So, these days, IDEs will provide very simple tools to both create a JAR file and to make this JAR file executable if you choose. We provide instructions on the accompanying web-site on how to do this for a popular Java IDE.

19.8 Adding External Libraries

Sometimes it is necessary to use libraries that are not part of the standard Java framework. One of the most common examples of this is in the development of applications that require the use of data held on a database which is stored either locally or on another machine on the network. In this section we will briefly explore two technologies—**Java Database Connectivity (JDBC)** and **Hibernate ORM** (Object/Relational Mapping), often referred to simply as *Hibernate*.

It is not our intention here to give a detailed explanation of how these technologies are used, but rather to give a concrete example of how it is possible to use libraries that are not part of the Java framework. This should, however, also serve to whet the appetites of those of you who wish to learn how to access a database via a Java application.

19.8.1 Accessing Databases Using JDBC

Database manufacturers provide JDBC *drivers* by means of which Java programs can access their databases. A driver is a piece of software that enables communication between two programs, or between a software program and a piece of hardware, by translating the output of one program into a form understood by the other one.

In order to use a particular driver we would download it from the manufacturer's website in the form of a `.jar` file. We would then need to ensure that the CLASSPATH points to the location of this file; this is most easily done by adding the `.jar` file to the run-time or compile-time configurations within an IDE such as NetBeans™ or Eclipse™. Instructions for doing this will be found within the documentation for the particular IDE.

In this chapter we are going to be referring to a MySQL™ database. The driver, which is called `Driver.class`, is contained within the `.jar` file. At the time of writing the current version is:

```
mysql-connector-java-5.1.45-bin.jar
```

In today's version of Java, all we have to do to make the driver accessible is to add it to the CLASSPATH as described above. In previous versions we would have needed to include the following code (which, as you see, refers to the driver by its full path name):

```
try
{
   Class.forName("com.mysql.jdbc.Driver");
}

catch(ClassNotFoundException e)
{
}
```

Table 19.1 The *products* table

Field	Data type	Length
serialNumber (*key field*)	char	(25)
make	char	(10)
description	char	(25)
price	decimal	(10, 2)

Java provides a package known as `java.sql`. This package provides the means by which our Java programs can contain commands written in standard SQL (Structured Query Language), which is the well-established means of writing database instructions. In this chapter we are not going to teach you SQL, but will assume you are familiar with some basic commands.

For our example we have set up a little database called *ElectricalStore* that contains a table called *products*. This table is described below (Table 19.1); it is assumed that you are familiar with relational databases and the data types available.

We have populated this database, and, in order to query it, we have developed a class called `ProductQuery`. This class, once an instance of it is created, executes just one SQL statement:

```
select * from products;
```

Those of you who are familiar with SQL will know that this query retrieves all the fields from all the records in the *products* table. The information obtained is then displayed in a text area as you can see from Fig. 19.2.

Electrical Store

Serial Number	Make	Description	Price
1076543	Acme	Vacuum Cleaner	£180.11
3756354	Nadir	Washing Machine	£178.97
1234567	Zenith	Fridge	£150.98
7876161	Zenith	Tumble Drier	£158.99
2876161	Acme	Hair Drier	£57.99

Fig. 19.2 Displaying the information from the *ElectricalStore* database

Take a look at the `ProductQuery` application below, and then we will go through it with you.

ProductQuery

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class ProductQuery extends Application

{
    public static final String URL = "jdbc:mysql://localHost/ElectricalStore?useSSL=true";
    public static final String USERNAME = "Kub";
    public static final String PASSWORD = "SydneyPaper";

    @Override
    public void start(Stage stage)
    {
        // create VBoxes to act as display columns
        VBox data1 = new VBox();
        VBox data2 = new VBox();
        VBox data3 = new VBox();
        VBox data4 = new VBox();

        data1.getChildren().add(new Text("Serial Number\n"));
        data2.getChildren().add(new Text("Make\n"));
        data3.getChildren().add(new Text("Description\n"));
        data4.getChildren().add(new Text("Price\n"));

        // configure the visual components
        HBox root = new HBox(10);
        root.setPadding(new Insets(10));
        root.getChildren().addAll(data1, data2, data3, data4);
        Scene scene = new Scene(root, 300, 150);
        stage.setTitle("Electrical Store");
        stage.setScene(scene);
        stage.show();

        Connection con;
        Statement st;
        ResultSet result;

        try
        {
            // connect to the database
            con = DriverManager.getConnection(URL, USERNAME, PASSWORD);

            // create an SQL statement
            st = con.createStatement();

            // execute an SQL query
            result = st.executeQuery("select * from products");

            while(result.next()) // move to next record
            {
                // retrieve and display first field
                data1.getChildren().add(new Text (result.getString(1)));
                // retrieve and display second field
                data2.getChildren().add(new Text(result.getString(2)));
                // retrieve and display third field
                data3.getChildren().add(new Text(result.getString(3)));
                // retrieve and display fourth field
                data4.getChildren().add(new Text("£" + result.getString(4)));
            }
        }

        catch(SQLException e) // handle the SQLException
        {

        }
    }

    public static void main(String[] args)
    {
       launch(args);
    }
}
```

You can see that we have declared some `String` constants as attributes—these will be explained shortly, when they are used.

In the `start` method we have created four `VBoxes`, one for each field of the data records—these will be lined up horizontally in an `HBox`. We have then declared a `Connection` object, a `Statement` object and a `ResultSet` object, none of which you have previously encountered. These are part of the `java.sql` package; their use will become clear in a moment.

We now declare a **`try`** … **`catch`** block because all of the methods we are going to use to communicate with the database throw `SQLExceptions`. The first thing we need to do within this block is to establish a connection with the database:

```
con = DriverManager.getConnection(URL, USERNAME, PASSWORD);
```

The `getConnection` method of `DriverManager` establishes a connection with the database referred to by the parameter `URL`, which was defined in the attribute declarations as:

```
public static final String URL = "jdbc:mysql://localHost/ElectricalStore?useSSL=true";
```

This is the correct format for the MySQL database called *ElectricalStore* which resides on the local machine and which uses the SSL security protocol. Other databases will require a slightly different format, the details of which can be found in the documentation for that product. Note that *localhost* is the way in which operating systems refer to the local machine—it is in fact an alias for IP (Internet Protocol) address 127.0.0.1, the normal loopback IP. If the database were located on another machine on the network, then this would be replaced by its name or IP address.[4] If the port number is required, this is placed after the name of the database separated by a colon—for example `Electrical Store:3306`. As you can see, the `getConnection` method receives, in addition to the URL (uniform resource locator), the user name and password; if this is not required, there is a version of `getConnection` that accepts the URL only (some databases allow the username and password to be embedded in the URL).

The method returns a `Connection` object, which we have assigned to the attribute `con`. A `Connection` object created in this way has a number of methods that allow communication with the database. One of these methods is called `createStatement`, and it is the next one we use:

```
st = con.createStatement();
```

[4]The system administrator will, of course, have had to set up the correct permissions for the database.

As you saw, we previously declared a `Statement` object, `st`, and this is now assigned the return value of the `createStatement` method. A `Statement` object is used for executing SQL statements and returning their results; in the next line we use its `executeQuery` method:

```
result = st.executeQuery("select * from products");
```

The data returned by executing the query is assigned to a `ResultSet` object, `result`. A `ResultSet` object holds a tabular representation of the data, and a pointer is maintained to allow us to navigate through the records. The `next` method moves the pointer to the next record, returning **`false`** if there are no more records. The individual fields are returned with methods such as `getString`, `getDouble` and `getInt`. You can see how we have used these methods in the `ProductQuery` class:

```
while(result.next()) // move to next record
{
   data1.getChildren().add(new Text(result.getString(1)));
   data2.getChildren().add(new Text(result.getString(2)));
   data3.getChildren().add(new Text(result.getString(3)));
   data4.getChildren().add(new Text("£" + result.getString(4)));
}
```

You can see that, as we are using `VBox`es to display our data, we have created a `Text` object for each string and added this to the correct `VBox`.

The version of `getString` that we have used here takes an integer representing the position of the field—in our example 1 is the *serialNumber*, 2 is *make* and so on. There is also a version of `getString` that accepts the name of the field. So, for example, we could have used, for the second field:

```
data2.getChildren().add(new Text(result.getString(make)));
```

Since all we are doing is displaying the data we used `getString` for the last field, even though it holds numeric data—this saved us the trouble of doing any formatting on it. If we had wished to do any processing with this data, we could have used the `getDouble` method to retrieve it as a **`double`** rather than a `String`.

19.8.2 Accessing Databases Using Hibernate

Hibernate is a more recent technology than JDBC; it allows us to store and retrieve whole objects from a database.

If you create your Hibernate project within Netbeans™ you will find that the necessary libraries will be added to the project. Otherwise you will need to download the files from the Hibernate website. Normally this will be in the form of a `.zip` file which contains a folder called *required*. In this folder you will find the `.jar` files that you will need to add to the CLASSPATH. As with the JDBC example in the previous section, you will need to add the relevant driver to the run-time configuration. We will use the same MySQL™ database as before, so you will need the same driver as in the previous section.

The first thing you will need to do is create a Java class for the objects that we will be dealing with—in our case we will need a `Product` class:

Product

```
public class Product
{
    private String stockNumber;
    private String manufacturer;
    private String item;
    private double unitPrice;

    public Product(String stockNumberIn, String manufacturerIn, String itemIn, double unitPriceIn)
    {
        stockNumber = stockNumberIn;
        manufacturer = manufacturerIn;
        item = itemIn;
        unitPrice = unitPriceIn;
    }

    public Product()
    {

    }

    public String getStockNumber()
    {
        return stockNumber;
    }

     public void setStockNumber(String stockNumberIn)
    {
        stockNumber = stockNumberIn;
    }

    public String getManufacturer()
    {
        return manufacturer;
    }

    public void setManufacturer(String manufacturerIn)
    {
        manufacturer = manufacturerIn;
    }


    public String getItem()
    {
        return item;
    }

    public void setItem(String itemIn)
    {
        item = itemIn;
    }

    public double getUnitPrice()
    {
        return unitPrice;
    }

    public void setUnitPrice(double unitPriceIn)
    {
        unitPrice = unitPriceIn;
    }
}
```

As you can see, the attributes of this class correspond to the fields in the database that you saw in the last section. Hibernate expects there to be `set`- and `get`-methods for these attributes, and also expects there to be an empty constructor.

You might be asking how the application will know which attribute in the Java class corresponds to which field of the database. The answer is that this information needs to be supplied in an XML mapping file, usually named `hibernate.hbm.xml`. Our file looks like this:

hibernate.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
                                        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name = "Product" table = "products">
        <id name = "stockNumber" column = "serialNumber"/>
        <property name = "manufacturer"  column = "make"/>
        <property name = "item" column = "description"/>
        <property name = "unitPrice" type = "double" column = "price"/>
  </class>
</hibernate-mapping>
```

If you are using an IDE wizard to create your Hibernate application, you will find that the first two lines are added into your file automatically—otherwise you can do it manually.

It is assumed here that you know a little about XML—but even if you don't, it is not hard to see how the attributes are mapped onto field names. Notice that the *id* tag specifies which attribute corresponds to the key field, while the *property* tag deals with the other attributes.

There is one more thing we need to do before writing our Hibernate application, which is to write a configuration file (normally called `hibernate.cfg.xml`) that will provide the information needed about the database. Here is ours for the MySQL database we described in the previous section:

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
                                         "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/ElectricalStore?useSSL=true</property>
    <property name="hibernate.connection.username">Kub</property>
    <property name="hibernate.connection.password">SydneyPaper</property>
  </session-factory>
</hibernate-configuration>
```

As before, using a wizard will cause the first two lines to be inserted for you. The wizard will then allow you to add the rest of the information via a design dialogue screen, or you can do it manually.

The example above is self explanatory. We have needed to add only three properties, the URL that points to the database together with the name and password. Other properties might be necessary to add, depending on the system; for

example if the specific driver name were required, that could be added (for a MySQL database) as follows:

```
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
```

Now we come to the application itself. Our little program, which we have called `ProductQuery2`, will do only as much as the one in the previous section, namely to display all the information about the current items held.

Once you have taken a look at it, we will explain what it is all about.

ProductQuery2

```
// accessing a database using Hibernate

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import java.util.ArrayList;
import org.hibernate.query.Query;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

public class ProductQuery2 extends Application
{
    @Override
    public void start(Stage stage)
    {
        // create VBoxes to act as display columns
        VBox data1 = new VBox();
        VBox data2 = new VBox();
        VBox data3 = new VBox();
        VBox data4 = new VBox();

        data1.getChildren().add(new Text("Serial Number\n"));
        data2.getChildren().add(new Text("Make\n"));
        data3.getChildren().add(new Text("Description\n"));
        data4.getChildren().add(new Text("Price\n"));

        HBox root = new HBox(10);
        root.setPadding(new Insets(10));
        root.getChildren().addAll(data1, data2, data3, data4);

        // configure the stage
        Scene scene = new Scene(root, 300, 150);
        stage.setTitle("Electrical Store");
        stage.setScene(scene);
        stage.show();

        // create a Configuration object
        Configuration cfg = new Configuration();

        // link the configuration object to the database properties
        cfg.configure("hibernate.cfg.xml");

        // specify the mapping file
        cfg.addResource("hibernate.hbm.xml");


        // create a Session object to act as an interface between the Java application and Hibernate
        ServiceRegistry serviceRegistry
                 = new StandardServiceRegistryBuilder().applySettings(cfg.getProperties()).build();
        SessionFactory  sessionFactory = cfg.buildSessionFactory(serviceRegistry);
        Session session = sessionFactory.openSession();

        // query the database
        Query query = session.createQuery("from Product");

        // create a list of products from the query
        ArrayList<Product> list = (ArrayList) query.list();

        // display the product details for each product in the list
        for(Product pr : list)
        {
            data1.getChildren().add(new Text(pr.getStockNumber()));
            data2.getChildren().add(new Text(pr.getManufacturer()));
            data3.getChildren().add(new Text(pr.getItem()));
```

```
            data4.getChildren().add(new Text("£" + pr.getUnitPrice()));
        }

        // close the session
        session.close();
        sessionFactory.close();
        StandardServiceRegistryBuilder.destroy(serviceRegistry);

    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

You can see from the code that after creating and configuring the visual components, the first thing we have done is to create a `Configuration` object which read the information from the XML files we created:

```
Configuration cfg = new Configuration();
cfg.configure("hibernate.cfg.xml");
cfg.addResource("hibernate.hbm.xml");
```

We now use this object to create a `Session` object, which is the interface between the Java application and the database. It is the equivalent of a JDBC `Connection` object. To do this you follow the rather verbose steps below:

```
ServiceRegistry serviceRegistry
                 = new StandardServiceRegistryBuilder().applySettings(cfg.getProperties()).build();
SessionFactory sessionFactory = cfg.buildSessionFactory(serviceRegistry);
Session session = sessionFactory.openSession();
```

The `Session` class has a number of methods, one of which is `createQuery`, which returns a Hibernate query. Hibernate has its own query language, **Hibernate Query Language (HQL)** which is similar to SQL; HQL queries are translated into SQL in order to query the database. The only one of these we will show you here is the "From" clause which will retrieve entire objects. You can see how it is used in our application:

```
Query query = session.createQuery("from Product");
```

The Hibernate `Query` class has a method called `list`; this returns a `List` object which we have stored as an `ArrayList` of `Products`:

```
ArrayList<Product> list = (ArrayList) query.list();
```

It is now a simple matter of scrolling through the list of products, displaying all the fields as we do so:

```
for(Product pr : list)
{
     data1.getChildren().add(new Text(pr.getStockNumber()));
     data2.getChildren().add(new Text(pr.getManufacturer()));
     data3.getChildren().add(new Text(pr.getItem()));
     data4.getChildren().add(new Text("£" + pr.getUnitPrice()));
}
```

Before the program terminates, we need to close it down properly. This involves three steps:

```
session.close();
sessionFactory.close();
StandardServiceRegistryBuilder.destroy(serviceRegistry);
```

We mentioned earlier that it is also possible to store whole objects as well as to retrieve them. If you have created a `Product` object, say `pr`, then you could store it with the following lines of code, which should come after the routine for opening a session.

```
Transaction tx = session.beginTransaction();

try
{
    session.save(pr);
    tx.commit();
}

catch(Exception e)
{
    if(tx!=null) tx.rollback();
}

finally
{
    session.close();
    sessionFactory.close();
    StandardServiceRegistryBuilder.destroy(serviceRegistry);
}
```

For a write operation such as this we need to start a transaction, which we do with the `beginTransaction` method of `Session`. We then store our object with the `save` method of `Session` (this has to be in **`try`**. … **`catch`** block). Hibernate does not automatically commit the save to the database, so we need to do this by calling the `commit` method of `Transaction`. Should there be any problem we have rolled back the change within the **`catch`** block. We have conveniently placed our close routines in the **`finally`** clause.

You might be wondering which technology to use, JDBC or Hibernate. There is no rule about this, and opinions differ. However, as a general rule of thumb, if all you want to do is to query an existing database with SQL then JDBC might be simpler. However, if your main aim to write a Java program that stores its data in a database then Hibernate might be the best solution.

19.9 Self-test Questions

1. What role do *packages* have in the development of classes?

2. Identify valid and invalid **import** statements amongst the following list:

   ```
   import java.*;
   import javafx.scene.*;
   import java.util.Scanner;
   import javax.scene.control.Button;
   import java.application.Application;
   import java.text.*.*;
   ```

3. Consider the following outline of a class, used in a computer game, that makes reference to JavaFX's `Button` class:

   ```
   public class GameController
   {
         private Button myButton;
         // more code here
   }
   ```

   At the moment the line referencing the `Button` class will not compile. Identify three different techniques to allow this class with a `Button` attribute to compile.

4. What is the purpose of the **CLASSPATH** environment variable?

5. You were asked to develop a time table application in programming exercise 8 of Chap. 8. Later, in programming exercise 3 of Chap. 14 you were asked to enhance this application with exceptions. Finally you were asked to develop a JavaFX interface for this application in programming exercise 6 of Chap. 17.

   (a) How would you place the classes that make up the time table application into a package called `timetableApp`?
   (b) What is meant by **package** scope and how would you give the `Booking` and `TimeTable` classes package scope?
   (c) How would you run this application from the command line?
   (d) What is the purpose of a JAR file and how would you create a JAR file for the `timetableApp` package form the command line?

6. Explain the fundamental differences between the JDBC and the Hibernate technologies in terms of their approach to accessing databases.

19.10 Programming Exercises

1. Make the changes discussed in this chapter so that the *Hostel* application is now part of a package called `hostelApp`.

2. Run the *Hostel* application from the command line.

3. Use your IDE to create an executable JAR file for your *Hostel* application then run your *Hostel* application by clicking this executable JAR file.

4. Make the changes to the time table application, discussed in self-test question 5 above, so that the application can be run from the command line and by clicking an executable JAR file.

5. Write a program that accepts a list of names from the command line and then sorts and displays these names on the screen. Run this program from the command line with a variety of names.

6. There are several Java packages that we have not yet explored. Browse your Java documentation to find out about what kind of classes these packages offer. For example, the `lang` package contains a class called `Math`, which has a **`static`** method called random designed to generate random numbers. There is also a random number class, `Random`, in the `util` package. Read your Java documentation to find out more about these random number generation techniques. Then write a program that generates five lottery numbers from 1 to 50 using:

   (a) the **`static`** `random` method of the `Math` class in the `lang` package;

   (b) the random number class, `Random`, in the `util` package.

7. In the previous chapter we developed several programs—for example `TextFileTester`—that stored and accessed data by creating files within the application. See if you can convert one of these programs so that the data is stored in a database to which you have access. In order to do this you will need to either have an account on a database system within your organisation, or to be running a database server (such as a MySQL server) locally.

20 Multi-threaded Programs

Outcomes:

By the end of this chapter you should be able to:

- *explain how concurrency is achieved by means of* ***time-slicing***;
- *distinguish between* ***threads*** *and* ***processes***;
- *implement threads in Java*;
- *explain the difference between* ***asynchronous*** *and* ***synchronized*** *thread execution*;
- *explain the terms* ***critical section*** *and* ***mutual exclusion****, and describe how Java programs can be made to implement these concepts*;
- *explain how* ***busy waiting*** *can be avoided in Java programs*;
- *provide a* ***state transition diagram*** *to illustrate the thread life-cycle*;
- *describe how the* `javafx.concurrent` *package is used to produce multi-threaded JavaFX applications*;
- *use the* `Task` *class and the* `Service` *class from the above package in JavaFX applications*;
- *use the above classes to create animated applications in JavaFX.*

20.1 Introduction

In this chapter you are going to learn how to make a program effectively perform more than one task at the same time—this is known as **multi-tasking**, and Java provides mechanisms for achieving this within a single program.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_20

20.2 Concurrent Processes

If you have been using computers for no more than a couple of decades then you will probably think nothing of the fact that your computer can appear to be doing a number of things at the same time. For example, a large file could be downloading from the web, while you are listening to music and typing a letter into your word processor. However, those of us who were using desktop computers in the 1980s don't take this for granted! We can remember the days of having to wait for our document to be printed before we could get on with anything else—the idea of even having two applications like a spreadsheet and a database loaded at the same time on a personal computer would have been pretty exciting.

In recent years dual core and quad core computers have become available on a wide scale—such computers have more than one processor, so it does not seem quite so extraordinary that they can perform more than one task at a time. However, with a quad core computer for example it is certainly possible to perform more than four tasks at once, and indeed multi-tasking has been possible for many years on machines with a single processor—and at first sight this does seem rather extraordinary. The way this is achieved is by some form of **time-slicing**; in other words the processor does a little bit of one task, then a little bit of the next and so on —and it does this so quickly it appears that it is all happening at the same time.

A running program is usually referred to as a **process**; two or more processes that run at the same time are called **concurrent** processes. Normally, when processes run concurrently each has its own area in memory where its program code and data are kept, and each process's memory space is protected from any other process. All this is dealt with by the operating system; modern operating systems such as Windows™ and Unix™ have a process management component whose job it is to handle all this.

20.3 Threads

We have just introduced the idea of a number of programs—or processes—operating concurrently. There are, however, times when we want a *single* program to perform two or more tasks at the same time. Whereas two concurrent programs are known as processes, each separate task performed by a single program is known as a **thread**. A thread is often referred to as a **lightweight process**, because it takes less of the system's resources to manage threads than it does to manage processes. The reason for this is that threads do not have completely separate areas of memory; they can share code and data areas. Managing threads, which also work on a time-slicing principle, is the job of the Java runtime environment working in conjunction with the operating system.

Let's illustrate this by developing a very simple program, which continuously displays the numbers from 0 to 9 in a console window as shown in Fig. 20.1.

The program will provide a simple graphical interface to start and stop the numbers being displayed, as shown in Fig. 20.2.

Output - Ed4Chapter22 (run)

```
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
```

Fig. 20.1 Output from the simple number display application

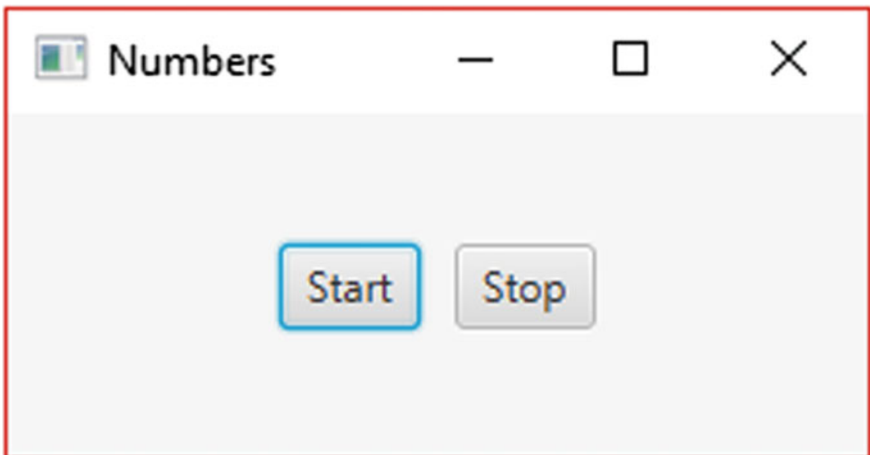


Fig. 20.2 Interface for the simple number display application

Intuitively you might think that the way to achieve the desired result would be to have a loop that keeps displaying the numbers, and which is controlled by some **boolean** variable, so that the loop continues while the variable is set to **true**, and stops when it is set to **false**.

So, with a **boolean** variable called go, you might expect the code for the Start button to look like this:

```
startButton.setOnAction(e ->  {
                                  go = true;
                                  int count = 0;
                                  while(go)
                                  {
                                         System.out.print(count);
                                         count++;
                                         if(count > 9) // reset the counter if it has reached 9
                                         {
                                               count = 0;
                                               System.out.println(); // start a new line
                                         }
                                  }
                              }
                        );
```

And the code for the stop button to look like this:

```
stopButton.setOnAction(e -> go = false);
```

If you were to run such a program, you would find that pressing the Start button would indeed start the numbers displaying—but pressing the Stop button, or clicking the cross hairs to try and stop the program wouldn't work—what would happen is that the application would eventually become unresponsive.

Can you see what's wrong here? If the mouse is clicked on the Start button then, as we have seen, the loop is started. But now the application is tied up executing the loop, so nothing else can happen. It doesn't matter how often you click on the Stop button, the program will never get the chance to process this event, because it is busy executing the loop.

What we need is to set up a separate thread that can busy itself with the loop, while another thread carries on with the rest of the program. Luckily Java provides us with a Thread class that allows us to do exactly that.

20.4 The *Thread* Class

The Thread class provides a number of different methods that allow us to create and handle threads in our Java programs. The Thread class implements an interface called Runnable which has just one method, run. The code for this method determines the action that takes place when the thread is started.

The DisplayNumbers program below shows how we can write our application with a separate thread that deals with the loop.

DisplayNumbers

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class DisplayNumbers extends Application
{
    private boolean go;

    @Override
    public void start(final Stage stage)
    {
        Button startButton = new Button("Start");
        Button stopButton = new Button("Stop");

        startButton.setOnAction(e ->
                    {
                           go = true;
                           // create a separate thread
                           Thread thread1 = new Thread(()->
                               {
                                   int count = 0;
                                   while(go)
                                   {
                                         System.out.print(count);
                                         count++;
                                         if(count > 9)
                                         {
                                               // reset the counter and start a new line
                                               count = 0;
                                               System.out.println();
                                         }
                                   }
                               });
                           thread1.start();
                     });

        stopButton.setOnAction(e -> go = false);

        HBox root = new HBox(10);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(startButton, stopButton);
        Scene scene = new Scene(root, 250, 100);
        stage.setScene(scene);
        stage.setTitle("Numbers");
        stage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Let's take a closer look at the code for the `setOnAction` method of the Start button:

```
startButton.setOnAction(e ->   {
                                   go = true;
                                   Thread thread1 = new Thread(()->
                                       {
                                            int count = 0;
                                            while(go)
                                            {
                                                 System.out.print(count);
                                                 count++;
                                                 if(count > 9)
                                                 {
                                                      count = 0;
                                                      System.out.println();
                                                 }
                                            }
                                       });
                                   thread1.start();
                              });
```

After setting go to **true** (which we are able to do within the lambda expression because go is an attribute of the class) we create a new thread. Since the Thread class implements Runnable we need to code its run method, and since Runnable is a functional interface, we can code its run method by using a lambda expression that provides the instructions for displaying the numbers. So it is now the responsibility of this new thread to display the numbers according to the instructions in its run method.

Once we have done this, we start the thread by calling its start method; we don't call the run method directly—the start method does this for us.

Once started, the thread executes and, via time-slicing, the application thread is also executed, so that it is possible to listen for events such as the Stop button being pressed. As you can see, when this happens go is set to **false**, and when the execution returns to our new thread the loop terminates.

The next section provides some more detail about how all this is achieved.

20.5 Thread Execution and Scheduling

As we explained earlier, concurrency, with a single processor, is achieved by some form of time-slicing. Each process or thread is given a little bit of time—referred to as a **quantum**—on the CPU, then the next process or thread takes its turn and so on.

Now, as you can imagine, there are some very complex issues to consider here. For example, what happens if a process that currently has the CPU cannot continue because it is waiting for some input, or perhaps is waiting for an external device like a printer to become available? When new processes come into existence, when do they get their turn? Should all processes get an equal amount of time on the CPU or should there be some way of prioritizing?

The answers to these questions are not within the domain of this book. However, it is important to understand that the responsibility for organizing all this lies with the operating system; in the case of multi-threaded Java programs this takes place in conjunction with the JVM. Different systems use different **scheduling algorithms** for deciding the order in which concurrent threads or processes are allowed CPU time. This is hidden from the user, and from the programmer. In the case of an application such as our counter program, all we can be sure about is the fact that one thread has to complete a quantum on the CPU before another thread gets its turn—we cannot, however, predict the amount of time that will be allocated to each thread.

There is quite a simple way to illustrate this. Let's add the following code to the instructions for the setOnAction method of the start button:

```
Thread thread2 = new Thread(()-> {
                                        int count = 0;
                                        while(go)
                                        {
                                              // print characters A to J
                                              System.out.print((char) (65 + count));
                                              count++;
                                              if(count > 9)
                                              {
                                                    count = 0;
                                                    System.out.println();
                                              }
                                        }
                                   });

                                   thread2.start()
```

So now we have created two threads which will run concurrently: `thread1` will display the numbers from 0 to 9 as before, while `thread2` will print the characters from “A” to “J” (you can see how we have achieved this by type casting from **`int`** to **`char`**—“A” has the Unicode value of 65).

Fig. 20.3 Two threads running concurrently

A typical output fragment from this program is shown in Fig. 20.3. This gives a very good illustration of how the execution switches randomly between threads, with the output switching unpredictably between numbers and letters.

20.6 Synchronizing Threads

In Sect. 20.5 we explained that under normal circumstances the behaviour of two or more threads executing concurrently is not co-ordinated, and we are not able to predict which threads will be allocated CPU time at any given moment. Unco-ordinated behaviour like this is referred to as **asynchronous** behaviour.

It is, however, often the case that we require two or more concurrently executing threads or processes to be co-ordinated—and if they were not, we could find we had some serious problems. There are many examples of this. One of the most common is that of a **producer–consumer** relationship, whereby one process is continually producing information that is required by another process. A very simple example of this is a program that copies a file from one place to another. One process is responsible for reading the data, another for writing the data. Since the two processes are likely to be operating at different speeds, this would normally be implemented by providing a *buffer*, that is a space in memory where the data that has been read is queued while it waits for the write process to access it and then remove it from the queue.

It should be fairly obvious that it could be pretty disastrous if the read process and the write process tried to access the buffer at the same time—both the data and the indices could easily be corrupted. In a situation like this we would need to treat the parts of the program that access the buffer as **critical sections**—that is, sections that can be accessed only by one process at a time.

Implementing critical sections is known as **mutual exclusion**, and Java provides a mechanism for the implementation of mutual exclusion in multi-threaded programs. In this book we are not going to go into any detail about how this is implemented, because the whole subject of concurrent programming is a vast one, and is best left to texts that deal with that topic. What we intend to do here is simply to explain the mechanisms that are available in Java for co-ordinating the behaviour of threads.

Java provides for the creation of a **monitor**, that is a mechanism by which a method can be accessed by only one thread at a time. This entails the use of the modifier `synchronized` in the method header. For instance, a `Buffer` class in the above example might have a `read` method declared as:

```
public synchronized Object read()
{
    .....
}
```

Because it is synchronized, as soon as some object invokes this method a **lock** is placed on it; this means that no other object can access it until it has finished executing. This can, however, cause a problem known as **busy waiting**. This means that the method that is being executed by a particular thread has to go round in a loop until some condition is met, and as a consequence the CPU time is used just to keep the thread going round and round in this loop until it times out—not very efficient! As an example of this, consider the `read` and `write` methods that we talked about in the example above. The `read` method would not be able to place any data in the buffer if the buffer were full—it would have to loop until some data was removed by the `write` method; conversely, the `write` method would not be able to obtain any data if the buffer were empty—it would have to wait for the `read` method to place some data there.

Java provides methods to help us avoid busy waiting situations. The `Object` class has a method called `wait`, which suspends the execution of a thread (taking it away from the CPU) until it receives a message from another thread telling it to wake up. The object methods `notify` and `notifyAll` are used for the purpose of waking up other threads. Sensible use of these methods allow programmers to avoid busy waiting situations.

20.7 Thread States

A very useful way to summarize what you have learnt about threads is by means of a **state transition diagram**. Such a diagram shows the various states that an object can be in, and the allowed means of getting from one state to another—the **transitions**. The state transition diagram for a thread is shown in Fig. 20.4.

As we have said, much of the thread's life-cycle is under the control of the operating system and the JVM; however some transitions are also under the control of the programmer. In Fig. 20.4 the transitions that are controlled by the operating system and the JVM are italicized; those that the programmer can control are in plain font.

As you have seen, a thread is brought into existence by invoking its `start` method. At this point it goes into the **ready** state. This means it is waiting to be allocated time on the CPU; this decision is the responsibility of the operating system and JVM. Once it is **dispatched** (that is given CPU time), it is said to be in the **running** state. Once a thread is running, a number of things can happen to it:

- It can simply timeout and go back to the **ready** state.
- The programmer can arrange for the `sleep` method to be called, causing the thread to go into the **sleeping** state for a given period of time. When this time period has elapsed the thread wakes up and goes back to the ready state.

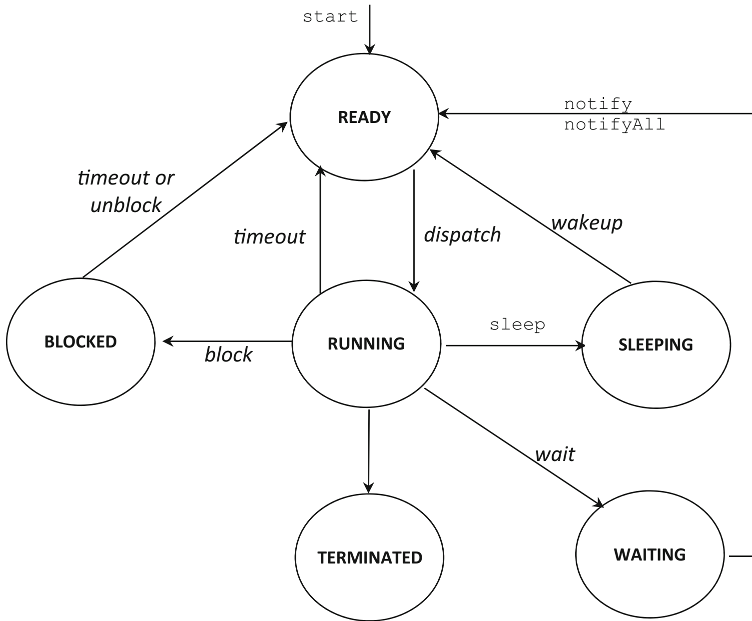


Fig. 20.4 The state transition diagram for a thread

- The programmer can use the `wait` method to force the thread to go into the **waiting** state until a certain condition is met. Once the condition is met, the thread will be informed of this fact by a `notify` or `notifyAll` method, and will return to the ready state.
- A thread can become **blocked**; this is normally because it is waiting for some input, or waiting for an output device to become available. The thread will return to the ready state when either the normal timeout period has elapsed or the input/output operation is completed.
- When the `run` method finishes the thread is terminated.

Let's look at how the programmer can influence the behaviour of a thread by calling the `sleep` method. This is a **`static`** method, and calling it with the class name will cause the currently executing thread to sleep for a specified number of milliseconds.

If we wanted our number display program to leave an interval of one second between each number being displayed we could adapt the method for the Start button as follows—the additional code is emboldened:

```
startButton.setOnAction(e ->
                    {
                           go = true;

                           Thread thread1 = new Thread(()->
                           {
                                   int count = 0;
                                   while(go)
                                   {
                                         System.out.print(count);
                                         try
                                         {
                                             Thread.sleep(1000);
                                         }
                                         catch(InterruptedException ex)
                                         {
                                         }
                                         count++;
                                         if(count > 9)
                                         {
                                               // reset the counter and start a new line
                                               count = 0;
                                               System.out.println();
                                         }
                                   }
                           });
                           thread1.start();
                     });
```

The `sleep` method throws an `InterruptedException`, which is a checked exception and so needs to be placed in a **`try`** … **`catch`** block as shown.

20.8 Multithreading and JavaFX

When we launch a JavaFX application the `init` method is called in the main thread (or launcher thread), and then, when the `start` method is called, the application itself runs in a separate thread called the **application thread**, which is also where the `stop` method runs.

If we want to write multi-threaded JavaFX applications, a special package—`javafx.concurrent`—is provided for this purpose. The package contains an interface called `Worker` that creates background tasks that can communicate with the user interface. Two classes implement this interface, `Task` and `Service`.

20.8.1 The `Task` Class

In our previous application we created a simple thread that only used the console for output, and we were able to get away with not using the special classes. But now we will adapt that program so that the display occurs in a `TextArea` as shown in Fig. 20.5.

We are going to use the `Task` class to implement this. `Task` is an abstract class, so we need either to use an anonymous class, or create a custom class that extends `Task`, which is what we have done below:

NumbersTask

```
import javafx.scene.control.TextArea;
import javafx.concurrent.Task;

public class NumbersTask extends Task<Void>
{
    private boolean go;
    private TextArea display;

    public  NumbersTask(TextArea displayIn)
    {
        display = displayIn;
    }

    @Override
    protected Void call()
    {
        go = true;
        int count = 0;

        while(go)
        {
            display.appendText("" + count);
            try
            {
                Thread.sleep(100); //force the thread to sleep for 100 milliseconds
            }
            catch(InterruptedException e)
            {
            }

            count++;
            if(count > 9) // reset the counter if it has gone over 9
            {
                count = 0;
                display.appendText("\n"); // start a new line
            }
        }
        return null;
    }

     public void finish()
     {
        go = false;
     }
}
```

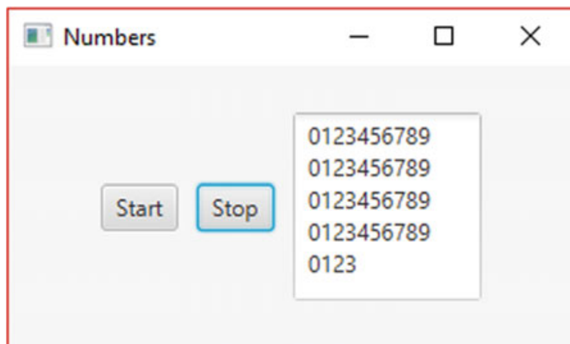


Fig. 20.5 The simple number display application outputting to a text area

`Task` is a generic class; its type is the return type of its **`abstract`** method, `call`. In our case there is no return value, so the type will be `Void`, as explained in Chap. 13.

An application that uses our `NumbersTask` class will need to inform it where to display the output. Therefore, a reference to a `TextArea` is sent into the class as

an argument, and is assigned to a variable `display`, which has been declared as an attribute of the class. We have declared another attribute, `go`, which is the **boolean** variable that will control the loop.

The `call` method, which has to be overridden, will contain the instructions for the task. You can see that it is very similar to the previous console version, except that the output is now directed to the text area. Notice that we need a **return** statement, and as it is of type `Void`, the return value is **null**.

Finally we have defined another method, `finish`, which simply sets `go` to **false**.

The program below uses our `NumbersTask` class.

DisplayNumbersInTextArea

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class DisplayNumbersInTextArea extends Application
{
  @Override
  public void start(final Stage stage)
  {
     Button startButton = new Button("Start");
     Button stopButton = new Button("Stop");
     TextArea displayArea = new TextArea();
     displayArea.setMaxSize(100, 100);
     displayArea.setEditable(false);

     NumbersTask task = new NumbersTask(displayArea);

     startButton.setOnAction(e ->  {
                                      Thread thread1 = new Thread(task);
                                      thread1.start();
                                   });

     stopButton.setOnAction(e -> task.finish());

     HBox root = new HBox(10);
     root.setAlignment(Pos.CENTER);
     root.getChildren().addAll(startButton, stopButton, displayArea);

     Scene scene = new Scene(root, 300, 150);
     stage.setScene(scene);
     stage.setTitle("Numbers");
     stage.show();
  }

  public static void main(String[] args)
  {
     launch(args);
  }
}
```

As you can see, we create a new instance of our `NumbersTask` class, and then, in the code for the start button, we create a new thread with a form of the constructor that accepts a `Task` object whose `call` method provides the code for its `run` method. We then start the thread as before.

The code for the stop button simply calls the `finish` method that we defined in the `NumbersTask` class.

Now, when you implement this program there is something that you will notice. Pressing the Start button starts the display and pressing the stop button ends it as expected. But then if you press the start button again, nothing happens. This is

because a Task object is designed to run only once. Once it has completed its job, it doesn't run again—for that you need to create a Service.

20.8.2 The Service Class

The Service class creates a Task object via its createTask method. When a new thread is created it calls this method to generate a task; hence a task can effectively run an infinite amount of times.

Our NumbersService class is shown below:

NumbersService

```
import javafx.scene.control.TextArea;
import javafx.concurrent.Task;
import javafx.concurrent.Service;

public class NumbersService extends Service<Void>
{
        private TextArea display;
        private boolean go;

        public  NumbersService (TextArea displayIn)
        {
              display = displayIn;
        }

        @Override
        protected Task<Void> createTask()
        {
              return new Task<Void> ()
              {
                    @Override
                    protected Void call() throws Exception
                    {
                        go = true;
                        int count = 0;

                        while(go)
                        {
                            display.appendText("" + count);

                            try
                            {
                                Thread.sleep(100); //force the thread to sleep
                            }
                            catch(InterruptedException e)
                            {
                            }

                            count++;
                            if(count > 9) // reset the counter
                            {
                                count = 0;
                                display.appendText("\n");
                            }
                        }
                        return null;
                    };
               };
        }

        public void finish()
        {
               display.appendText("\n");
               go = false;
        }
}
```

In the case of a Service, the method that has to be implemented is createTask. This returns a Task object, which is specified here by overriding its

call method as before. In this case we have achieved this by means of an anonymous class. The code for the call method is the same as before.

The only other thing to mention here is that we have tidied up the output by adding a newline statement in the finish method, so that the display starts on a new line after having been stopped.

Our previous program is adapted below to make use of the NumbersService class:

DisplayNumbersInTextAreaUsingService

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;


public class DisplayNumbersInTextAreaUsingService extends Application
{
    @Override
    public void start(final Stage stage)
    {
        Button startButton = new Button("Start");
        Button stopButton = new Button("Stop");
        TextArea displayArea = new TextArea();
        displayArea.setMaxSize(100, 100);
        displayArea.setEditable(false);


        NumbersService service = new NumbersService(displayArea);


        startButton.setOnAction(e ->
                    {
                        Thread thread1 = new Thread(service.createTask());
                        thread1.start();
                    }
                           );

        stopButton.setOnAction(e -> service.finish());


        HBox root = new HBox(10);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(startButton, stopButton, displayArea);

        Scene scene = new Scene(root, 300, 150);
        stage.setScene(scene);
        stage.show();
    }


    public static void main(String[] args)
    {
        launch(args);
    }

}
```

If you implement this program you will see that the display can be started and stopped as many times as you want.

20.8.3 Automating the *ChangingFace* Application

JavaFX provides some advanced and sophisticated routines for producing animations, which we have not had room to cover in this text. However, because animations—from the simple to the complex—consist of continuously performing a task, then some degree of threading will be required.

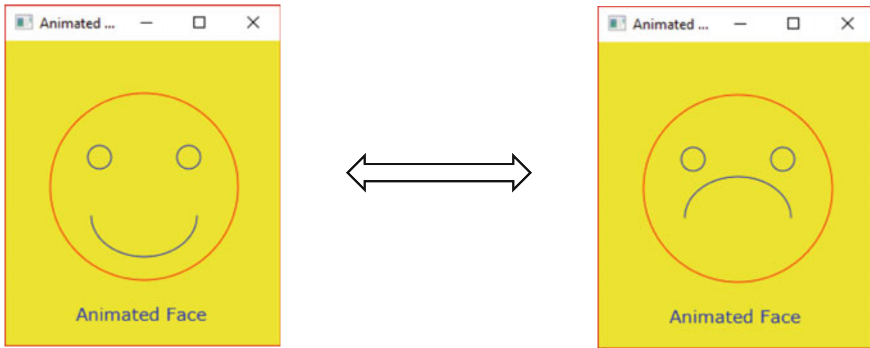


Fig. 20.6 Automating the *ChangingFace* application

Let's apply this principle to our ChangingFace application from Chap. 10, and "automate" the task of changing from the smiling face to the frowning face so that it every second it changes from one to the other as shown in Fig. 20.6.

Once again we have created a custom class as shown below:

FaceTask

```
import javafx.concurrent.Task;
import javafx.scene.shape.Arc;
import javafx.stage.Stage;

public class FaceTask extends Task<Void>
{
    private Arc arc;
    private Stage stage;

    public FaceTask(Arc arcIn, Stage stageIn)
    {
        arc = arcIn;
        stage = stageIn;
    }

    @Override
    protected Void call()
    {
        while(stage.isShowing())
        {
               arc.setLength(-180); // smiling mouth
               try
               {
                     Thread.sleep(1000); //force the thread to sleep for 1 second
               }
               catch(InterruptedException e)
               {
               }

               arc.setLength(180);  // frowning mouth

               try
               {
                    Thread.sleep(1000); //force the thread to sleep for 1 second
               }
               catch(InterruptedException e)
               {
               }
        }
        return null;
    }
}
```

If you recall from Chap. 10, the mouth is formed from an object of type `Arc`, and therefore our class requires a reference to the mouth to be sent in from the main application, in order that it can make the periodic change to it. It also needs a reference to the stage on which the scene graphic is placed—we will explain why it needs this in a moment. An `Arc` object and a `Stage` object are therefore declared as attributes of the class and these are given a value via the constructor

The `call` method is self-explanatory—the arc is re-drawn every second, first clockwise and then anticlockwise so that mouth smiles and frowns repeatedly.

Because we have not provided a means to stop the thread, we want this to happen when the stage is closed (that is, the cross-hairs have been pressed by the user), so that the application terminates normally. To do this we have arranged for the loop to check on each iteration that the stage is still showing—hence the need for a reference to the stage. For this purpose we have used the `isShowing` method of `Stage`

The following program uses our `FaceTask` to create the animation.

AnimatedFace

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.text.Text;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.Group;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.text.Font;

public class AnimatedFace extends Application
{
    @Override
    public void start(final Stage stage)
    {
        Circle face = new Circle(118, 125, 80);
        face.setFill(Color.YELLOW);
        face.setStroke(Color.RED);

        Circle rightEye = new Circle(80, 100, 10);
        rightEye.setFill(Color.YELLOW);
        rightEye.setStroke(Color.BLUE);

        Circle leftEye = new Circle(156, 100, 10);
        leftEye.setFill(Color.YELLOW);
        leftEye.setStroke(Color.BLUE);

        Arc mouth = new Arc(118, 150, 45, 35, 0, -180);
        mouth.setFill(Color.YELLOW);
        mouth.setStroke(Color.BLUE);
        mouth.setType(ArcType.OPEN);

        Text caption = new Text(60, 240, "Animated Face");
        caption.setFont(Font.font ("Verdana", 15));
        caption.setFill(Color.BLUE);

        Group root = new Group(face, rightEye, leftEye, mouth, caption);

        Scene scene = new Scene(root, Color.YELLOW);

        stage.setScene(scene);
        stage.setHeight(300);
        stage.setWidth(250);
        stage.setTitle("Animated Face");
        stage.show();

       Thread thread1 = new Thread(new FaceTask(mouth, stage));
       thread1.start();
    }

    public static void main(String[] args)
    {
       launch(args);
    }

}
```

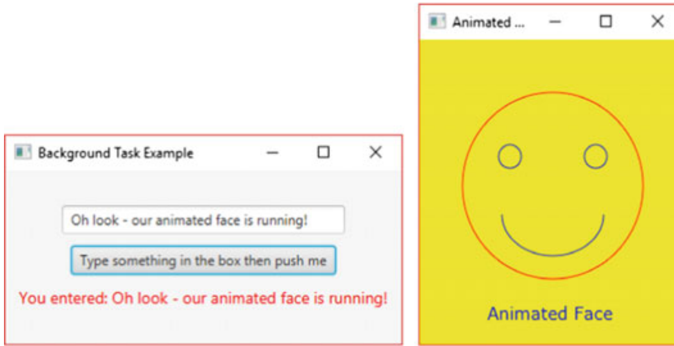


Fig. 20.7 The animated face running in the background

20.8.4 Running a Task in the Background

Whenever we want a task to run in the background we need to place it in a separate thread. To illustrate this we will keep it simple and use two applications that we have already developed—our `PushMe` application from Chap. 10 and the `AnimatedFace` that we just developed, which will continuously run in the background. This is shown in Fig. 20.7.

The following program achieves this result:

BackgroundTaskExample

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
import javafx.scene.Scene;
import javafx.geometry.Pos;
import javafx.scene.Group;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Circle;

public class BackgroundTaskExample extends Application
{
    @Override
    public void start(final Stage primaryStage)
    {
        // create and configure a text field for user entry
        TextField pushMeTextField = new TextField();
        pushMeTextField.setMaxWidth(250);

        // create and configure a label to display the output
        Label pushMeLabel= new Label();
        pushMeLabel.setTextFill(Color.RED);
        pushMeLabel.setFont(Font.font("Ariel", 14));

        // create and configure a label which will cause the text to be displayed
        Button pushMeButton = new Button();
        pushMeButton.setText("Type something in the box then push me");
        pushMeButton.setOnAction(e -> pushMeLabel.setText("You entered: " + pushMeTextField.getText()));

        // create and configure a VBox to hold our components
        VBox root = new VBox();
        root.setSpacing(10);
        root.setAlignment(Pos.CENTER);
```

```
        //add the components to the VBox
        root.getChildren().addAll(pushMeTextField, pushMeButton, pushMeLabel);

        // create a new scene
        Scene scene = new Scene(root, 350, 150);
        primaryStage.setScene(scene);

        primaryStage.setTitle("Background Task Example");
        primaryStage.show();
        createBackgroundTask(primaryStage);
    }


    private void createBackgroundTask(Stage stageIn)
    {
        // create and configure the main circle for the face
        Circle face = new Circle(118, 125, 80); // face
        face.setFill(Color.YELLOW);
        face.setStroke(Color.RED);

        // create and configure the circle for the right eye
        Circle rightEye = new Circle(80, 100, 10);
        rightEye.setFill(Color.YELLOW);
        rightEye.setStroke(Color.BLUE);

        // create and configure the circle for the left eye
        Circle leftEye = new Circle(156, 100, 10);
        leftEye.setFill(Color.YELLOW);
        leftEye.setStroke(Color.BLUE);

        // create and configure a smiling mouth
        Arc mouth = new Arc(118, 150, 45, 35, 0, -180);
        mouth.setFill(Color.YELLOW);
        mouth.setStroke(Color.BLUE);
        mouth.setType(ArcType.OPEN);

        // create and configure the text
        Text caption = new Text(60, 240, "Animated Face");
        caption.setFont(Font.font ("Verdana", 15));
        caption.setFill(Color.BLUE);

        Group root = new Group(face, rightEye, leftEye, mouth, caption);

        Scene scene = new Scene(root, Color.YELLOW);

        // create and configure a secondary stage
        Stage secondaryStage = new Stage();
        secondaryStage.setScene(scene);
        secondaryStage.setHeight(300);
        secondaryStage.setWidth(250);
        secondaryStage.setTitle("Animated Face");

        // position the secondary stage relative to the primary stage
        secondaryStage.setX(stageIn.getX() + 400);
        secondaryStage.setY(stageIn.getY());

        secondaryStage.show();

        // create a new thread
        Thread thread1 = new Thread(new FaceTask(mouth, secondaryStage));
        thread1.start();
    }

    public static void main(String[] args)
    {
       launch(args);
    }
}
```

It should be quite easy to see what we have done here. After creating the scene for the `PushMe` class and adding it to the primary stage, we have called a helper method, `createBackgroundTask`, which creates a secondary stage on which the face will appear. The new thread is then created, using our custom `FaceTask` class. The secondary stage is positioned relative to the primary stage, hence the need for a reference to the primary stage to be sent to `createBackgroundTask` as an argument.

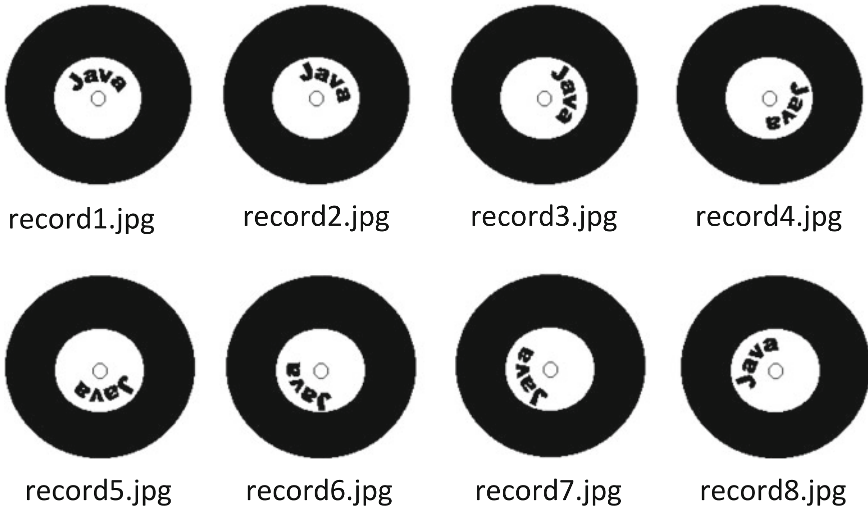


Fig. 20.8 The images used for the record animation

20.8.5 Animation Using a Series of Images

Prior to the advent of digital technology, the traditional way of producing animations in film was to display a continuous series of images, which, to the eye, gave the impression of movement.

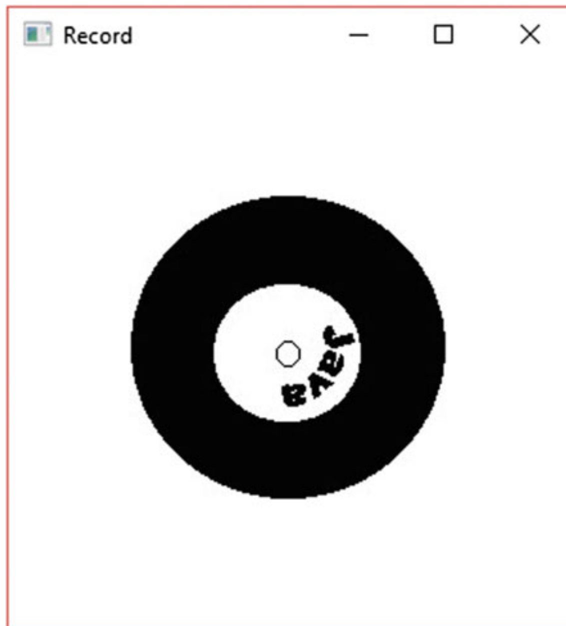


Fig. 20.9 The *Record* application

So to end this chapter, let's have some fun and replicate that technique by depicting an old-style vinyl record going round on a turn-table. There are eight images involved in our animation, as shown in Fig. 20.8.

Figure 20.9 shows a snapshot of the application in action.

The code for the application is presented below. You will see that we have taken a slightly different approach here; we have not created a custom class, but instead have written an anonymous class to define our `Task`.

Record

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.control.Label;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Background;
import javafx.scene.layout.VBox;
import javafx.concurrent.Task;

public class Record extends Application
{
    // declare some constants
    private static final int NUMBER_OF_IMAGES = 8;
    private static final int SLEEP_TIME = 100;

    private Label label = new Label();

    @Override
    public void start (Stage stage)
    {
        VBox root = new VBox();
        root.setBackground(Background.EMPTY);
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(label);

        Scene scene = new Scene(root, 300, 300);
        stage.setScene(scene);
        stage.setTitle("Record");

        stage.show();
        begin(stage);
    }

    private void begin(Stage stageIn)  // helper method
    {
        // create an image from a file and add it to a label
        ImageView imageView = new ImageView();
        label.setGraphic(imageView);

        Thread thread1 = new Thread(new Task<Void>() // anonymous class
        {
            String imageFileName;
            int currentImage = 1;
            Image image;

            @Override
            protected Void call()
            {
                while(stageIn.isShowing())
                {
                    // create the name of the next image to be used
                    imageFileName = "record" + currentImage + ".jpg";

                    image = new Image(imageFileName);
                    imageView.setImage(image);

                    try
                    {
                        Thread.sleep(SLEEP_TIME);
                    }
                    catch(InterruptedException e)
                    {
                    }

                    currentImage++; // next image
                    if(currentImage == NUMBER_OF_IMAGES + 1)
                    {
                        currentImage = 1;
                    }
                }
                return null;
            }
        });
```

```
        thread1.start();
    }


    public static void main(String[] args)
    {
        launch(args);
    }
}
```

To make the code easier to read we have placed everything to do with the creation and operation of the thread in a helper method called `begin`:

```
private void begin(Stage stageIn)
{
        // create an image from a file and add it to a label
        ImageView imageView = new ImageView();
        label.setGraphic(imageView);

        Thread thread1 = new Thread(new Task<Void>() // anonymous class
        {
            String imageFileName;
            int currentImage = 1;
            Image image;

            @Override
            protected Void call()
            {
                while(stageIn.isShowing())
                {
                    // create the name of the next image to be used
                    imageFileName = "record" + currentImage + ".jpg";

                    image = new Image(imageFileName);
                    imageView.setImage(image);

                    try
                    {
                        Thread.sleep(SLEEP_TIME);
                    }

                    catch(InterruptedException e)
                    {
                    }

                    currentImage++; // next image
                    if(currentImage == NUMBER_OF_IMAGES + 1)
                    {
                        currentImage = 1;
                    }
                }
                return null;
            }
        });
        thread1.start();
}
```

The method begins by creating an empty `ImageView` and adding it to a label. Then a new `Thread` is created with a `Task` object as its argument; in this case the `Task` is created and specified as an anonymous class.

The code for the `call` method of the `Task` should not be too difficult to understand. We have incremented a counter (`currentImage`) on each iteration of the loop, and each time the name of the image file is reset accordingly. The file name therefore changes from `record1.jpg` to `record2.jpg` and so on, until the final image is reached and the counter is reset to 1. On each iteration the `Image` and `ImageView` are set to the correct image using this file name.

The speed of the rotation is set by causing the thread to sleep for a given number of milliseconds (`SLEEP_TIME`). We have chosen 100 ms, but of course you can experiment with different values.

Once again we have used the `isShowing` method of `Stage` to terminate the thread when the application is closed.

20.9 Self-test Questions

1 Explain how concurrency is achieved by means of *time-slicing*.

2 Distinguish between *threads* and *processes*.

3 What is the difference between *asynchronous* and *synchronized* thread execution?

4 What is meant by the terms *critical section* and *mutual exclusion*? How are Java programs made to implement these concepts?

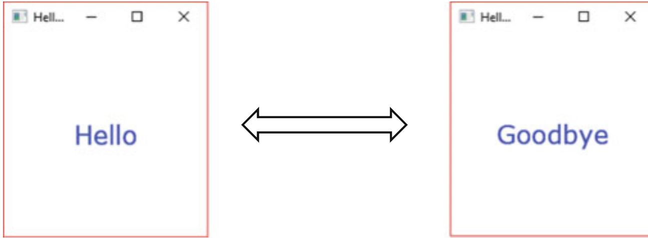
5 Explain how *busy waiting* can be avoided in Java programs.

6 Which two classes exist in the `javafx.concurrent` package to support multithreading in JavaFX applications? What is the main difference between these classes?

7 The application below has had its `call` method replaced by a comment.

```
import javafx.application.Application;
import javafx.concurrent.Task;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.text.Text;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.layout.Background;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;

public class HelloGoodbye extends Application
{
    private Text text = new Text(60, 240, "Hello");

    public void start (Stage stage)
    {
        text.setFont(Font.font ("Verdana", 25));
        text.setFill(Color.BLUE);
        VBox root = new VBox();
        root.setBackground(Background.EMPTY);
        root.setAlignment(Pos.CENTER);
        root.getChildren().add(text);
        Scene scene = new Scene(root, 200, 200);
        stage.setScene(scene);
        stage.setTitle("Hello-Goodbye");
        stage.show();
        begin(stage);
    }

    private void begin(Stage stageIn)
    {
        Thread thread1 = new Thread(new Task<Void>()
        {
             protected Void call()
            {
                    //code goes here
            }
        });
        thread1.start();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Replace the comment with code so that the application continuously displays the words “Hello” and “Goodbye” as shown below, changing once a second.



20.10 Programming Exercises

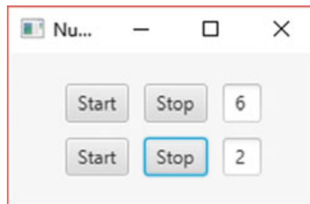
1. Implement some of the programs from this chapter. The images that you need for the `Record` class can be downloaded from the website. Try to design some animations of your own.

2. Implement the application that you completed in question 7 of the self-test questions.

3. Try making some modifications to the applications developed in Sects. 20.8.1 and 20.8.2 which continuously display number sequences.

   One possible modification would be to alter the `NumbersTask` or `NumbersService` class to allow the sleep interval to be passed as a parameter when an object is instantiated.

   Another might be to continuously display single digits in a text field (rather than a text area), and perhaps have two more threads operating at the same time, with different sleep intervals. An example is shown below:

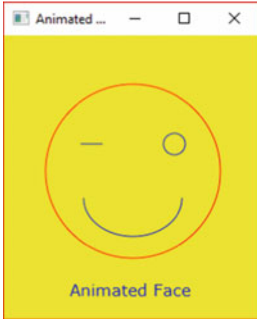


4. Make some alterations to the animated face application that we developed in Sect. 20.8.3. Here are some suggestions:

   (a) The sleep interval could be sent into the `FaceTask` as a parameter.

   (b) The mouth could change colour each time it changes expression.

(c) Instead of the mouth changing, one eye could wink as shown below:



You would need to modify both the `FaceTask` class and the `AnimatedFace` class so that one eye was no longer a circle, but an arc drawn through 360°. Setting the *y* radius to zero will produce a straight line.

(d) See if you can create a face in which both the eye winks and the mouth changes expression, but at different intervals. For this purpose you would need two versions of the `FaceTask` class.

21 Advanced Case Study

Outcomes:

By the end of this chapter you should be able to:

- *specify system requirements by developing a* ***use case model***;
- *annotate a* ***composition*** *association on a UML diagram*;
- *specify* ***enumerated types*** *in UML and implement them in Java*;
- *develop test cases from* ***behaviour specifications*** *found in the use case model*;
- *use the* `TabPane` *class to create an attractive user interface*;
- *add* ***tool tips*** *to JavaFX components.*

21.1 Introduction

You have covered quite a few advanced topics now in this second semester. In this chapter we are going to take stock of what you have learnt by developing an application that draws upon all these topics. We will make use of Java's package notation for bundling together related classes; we will implement interfaces; we will catch and throw exceptions; we will make use of the collection classes in the `java.util` package and we will store objects to file. We will also make use of many JavaFX components to develop an attractive graphical interface.

As with the case study we presented to you in the first semester, we will discuss the development of this application from the initial stage of requirements analysis, through to final implementation and testing stages. Along the way we will look at a few new concepts.

© Springer Nature Switzerland AG 2019
Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_21

21.2 System Overview

The application that we will develop will keep track of planes using a particular airport. So as not to overcomplicate things, we will make a few assumptions:

- there will be no concept of *gates* for arrival and departure—passengers will be met at a runway on arrival and be sent to a runway on departure;
- planes entering airport airspace and requesting to land are either called into land on a free runway, or are told to join a queue of circling planes until a runway becomes available;
- once a plane departs from the airport it is removed from the system.

21.3 Requirements Analysis and Specification

Many techniques are used to determine system requirements. Among others, these include interviewing the client, sending out questionnaires to the client, reviewing any documentation if a current system already exists and observing people carrying out their work. A common way to document these requirements in UML is to develop a **use case model**. A use case model consists of **use case diagrams** and **behaviour specifications**.

A *use case diagram* is a simple way of recording the *roles* of different users within a system and the services that they require the system to deliver. The users (people or other systems) of a system are referred to as **actors** in use case diagrams and are drawn as simple stick characters. The roles these actors play in the system are used to annotate the stick character. The services they require are the so-called *use cases*. For example, in an ATM application an actor may be a customer and one of the use cases (services) required would be to withdraw cash. A very simple use case diagram for our application is given in Fig. 21.1.

Figure 21.1 depicts the actors in this application (air traffic controllers and information officers) and the services these actors require (registering a flight, listing arrivals and so on). Once a list of use cases has been identified, *behaviour specifications* are used to record their required functionality. A simple way of recording behaviour specifications is to give a simple textual description for each use case. Table 21.1 contains behaviour specifications for each use case given in Fig. 21.1. Note that the descriptions are always given from the users' point of view.

As the system develops, the use case descriptions may be modified as detailed requirements become uncovered. These descriptions will also be useful when testing the final application, as we will see later.

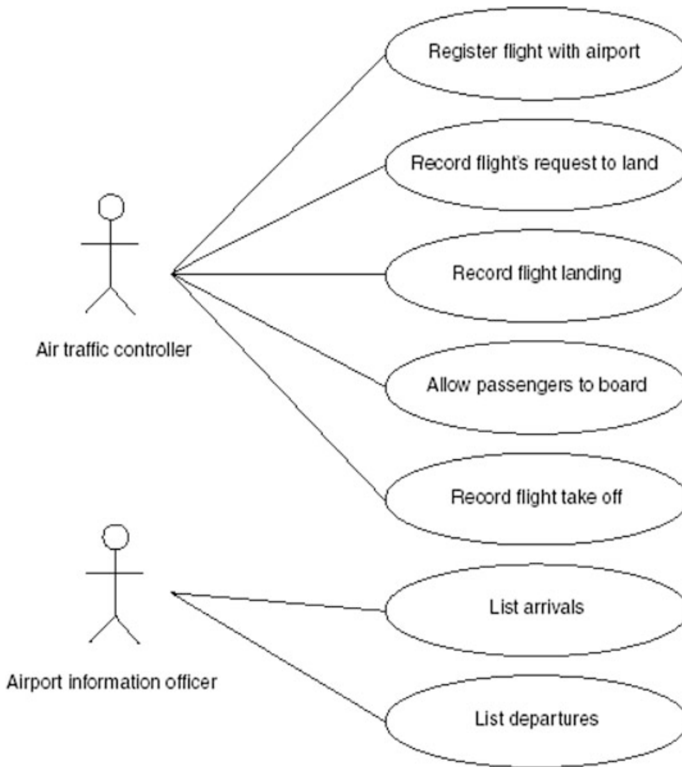


Fig. 21.1 A use case diagram for the airport application

Table 21.1 Behaviour specifications for the airport application

Register flight with airport	An air traffic controller registers an incoming flight with the airport by submitting its unique flight number, and its city of origin. If the flight number is already registered by the airport, the software will signal an error to the air traffic controller
Record flight's request to land	An air traffic controller records an incoming flight entering airport airspace, and requesting to land, by submitting its flight number. As long as the plane has previously registered with the airport, the air traffic controller is given an unoccupied runway number on which the plane will have permission to land. If all runways are occupied however, this permission is denied and the air traffic controller is informed to instruct the plane to circle the airport. If the plane has not previously registered with the airport, the software will signal an error to the air traffic controller
Record flight landing	An air traffic controller records a flight landing on a runway at the airport by submitting its flight number and the runway number. If the plane was not given permission to land on that runway, the software will signal an error to the air traffic controller

(continued)

Table 21.1 (continued)

Allow Passengers to board	An air traffic controller allows passengers to board a plane currently occupying a runway by submitting its flight number, and its destination city. If the given plane has not yet recorded landing at the airport, the software will signal an error to the air traffic controller
Record flight take off	An air traffic controller records a flight taking off from the airport by submitting its flight number. If there are planes circling the airport, the first plane to have joined the circling queue is then given permission to land on that runway. If the given plane was not at the airport, the software will signal an error to the air traffic controller
List arrivals	The airport information officer is given a list of planes whose status is either due-to-land, waiting-to-land, or landed
List departures	The airport information officer is given a list of planes whose status is currently waiting-to-depart (taking on passengers)

21.4 Design

The detailed design for this application is now presented in Fig. 21.2. It introduces some new UML notation. Have a look at it and then we will discuss it.

As you can see from Fig. 21.2, an `Airport` class has been introduced to represent the functionality of the system as a whole. The **`public`** methods of the `Airport` class correspond closely to the use cases identified during requirements analysis and specification. Notice we have provided two constructors. One that will allow us to create an empty `Airport` object and another that allows us to provide a filename (as a `String`) and load data stored in the given file. The **`private`** methods of the `Airport` class are there simply to help implement the functionality of the class.

The requirements made clear that there would be *many* planes to process in this system. Since the airport exists regardless of the number of planes at the airport, the relationship between the `Airport` and `Plane` class is one of containment, as indicated with a hollow diamond. It makes sense to consider the collection classes in the `java.util` package at this point. As we record planes in the system, and process these planes, we will always be using a plane's flight number as a way of identifying an individual plane. A `Map` is the obvious collection to choose here, with flight numbers the *keys* of the `Map` and the planes associated with these flight numbers as *values* of the `Map`.

The one drawback with a `Map`, however, is that it is not ordered on input. When considering which plane in a circling queue of planes to land, ordering is important, as the first to join the queue should be the first to land. So we have also introduced a `List` to hold the flight numbers of circling planes. Notice that the contained `Plane` type requires `equals` and `hashCode` methods to work effectively with these collection classes.

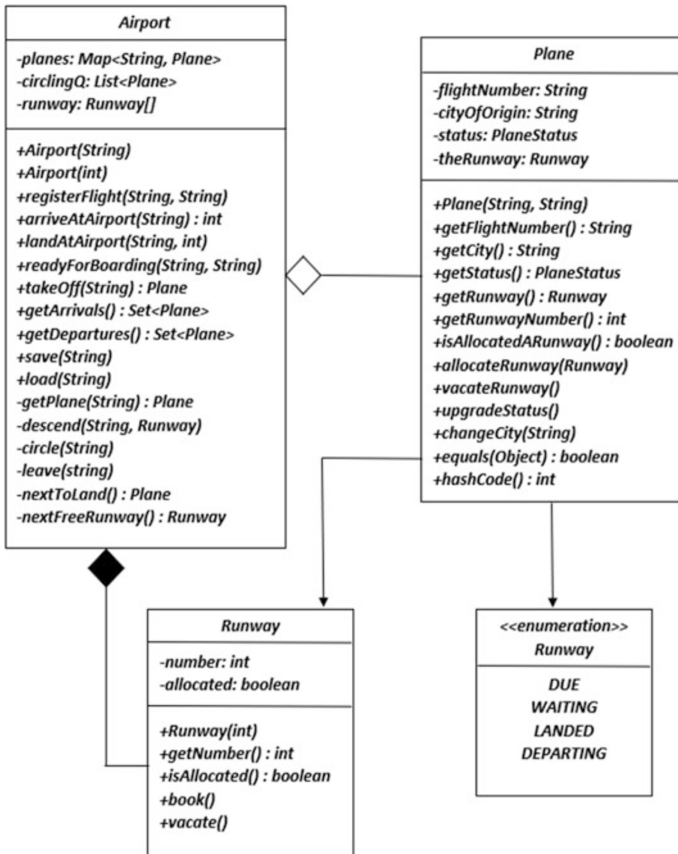


Fig. 21.2 Detailed design for the airport application

The airport will also consist of a number of runways. In fact the airport cannot exist without this collection of runways. The airport is said to be *composed of* a number of runways as opposed to *containing* a number of planes. Notice that the UML notation for **composition** is the same as that for containment, except that the diamond is filled rather than hollow. We use an array to hold this collection of `Runway` objects.

Turning to the contained classes, the `Runway` class provides methods to allow for the runway number to be retrieved, and for a runway to be booked and vacated. The `Plane` class also has access to a `Runway` object, to allow a plane to be able to book and vacate runways. You can see that as well as each plane being associated with a runway, the plane also has a flight number, a city and a status associated with it. The arrows from the `Plane` class to the `PlaneStatus` and `Runway` classes indicate the direction of the association. In this case a `Plane` object can send messages to a `Runway` and `PlaneStatus` object, but not vice versa.

The status of a plane is described in the `PlaneStatus` diagram. This diagram is the UML notation for an *enumerated type*, which is a type we have not met before.

21.5 Enumerated Types in UML

A type that consists of a few possible values, each with a meaningful name, is referred to as an **enumerated type**. The status of a plane is one example of an enumerated type. This status changes depending upon the plane's progress to and from the airport:

- when a plane registers with the airport, it is *due* to land;
- when a plane arrives in the airport's airspace, it is *waiting* to land (this plane may be told to come in and land, or it may have to circle the airport until a runway becomes available);
- when a plane touches down at the airport, it has *landed*;
- when a plane starts boarding new passengers, it is *departing* the airport.

You can see from the design of the system that such a type is captured in UML by marking this type with `<<enumeration>>` as follows (Fig. 21.3).

We need to mark this UML diagram with `<<enumeration>>` so that it is not confused with a normal UML class diagram. With a normal UML class diagram, attributes and methods are listed in the lower portion. With an enumerated type diagram, the possible values of this type are given in the lower portion of the diagram, with each value being given a meaningful name. An attribute that is allocated a `PlaneStatus` type, such as `status` in the `Plane` class, can have any one of these values.

This completes our design analysis, so now let's turn our attention to the Java implementation.

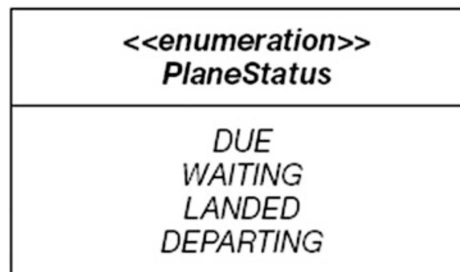


Fig. 21.3 The UML design of the enumerated *PlaneStatus* type

21.6 Implementation

Since we are developing an application involving several classes, it makes sense to bundle these classes together into a single package. We will call this package airportSys. This means that all our classes will begin with the following **package** statement:

```
package airportSys;
```

It is a good idea to hide implementation level exceptions (such as NumberFormatException) from users of the application and, instead, always throw some general application exception. In order to be able to do this, we define our own general AirportException class.

AirportException

```
package airportSys; // add to package

/**
  * Application Specific Exception
  *
  * @author Charatan and Kans
  * @version 1st August 2018
  */
public class AirportException extends RuntimeException
{
  /**
    * Default Constructor
    */
  public AirportException ()
  {
     super("Error: Airport System Violation");
  }

  /**
    * Constructor that accepts an error message
    */
  public AirportException (String msg)
  {
     super(msg);
  }
}
```

Notice that, as with all the classes we shall develop here, we have added Javadoc comments into the class definition. Now let's consider the remaining classes. First of all, we will look at the implementation of the enumerated PlaneStatus type.

21.6.1 Implementing Enumerated Types in Java

In order to define an enumerated type such as PlaneStatus, the **enum** keyword is used. The PlaneStatus type can now be implemented simply as follows:

```
// this is how to definine an enumerated type in Java
public enum PlaneStatus
{
   DUE, WAITING, LANDED, DEPARTING
}
```

You can see how easy it is to define an enumerated type. When defining such a type, do not use the **class** keyword, use the **enum** keyword instead. The different values for this type are then given within the braces, separated by commas.

These values create class constants, with the given names, as before. The type of each class constant is PlaneStatus and variables can now be declared of this type. For example, here we declare a variable of the PlaneStatus type and assign it one of these class constant values:

```
PlaneStatus status; // declare PlaneStatus variable
status = PlaneStatus.DEPARTING; // assign variable a class constant
```

The variable status can take no other values, apart from those defined in the enumerated PlaneStatus type. Each enumerated type you define will also have an appropriate toString method generated for it, so values can be displayed on the screen:

```
System.out.println("Value = " + status);
```

Assuming we created this variable as above, this would display the following:

Value = DEPARTING

As well as a toString method, a few other methods are generated for you as well, and the **switch** statement can be used in conjunction with enumerated type variables. We will see examples of these features when we look at the code for the other classes in this application.

Of course, we must remember to add this PlaneStatus type into our airportSys package:

The *PlaneStatus* type

```
package airportSys; // add to package

/**
  * Enumerated plane status type.
  *
  * @author Charatan and Kans
  * @version 1st August 2018
  */
public enum PlaneStatus
{
    DUE, WAITING, LANDED, DEPARTING
}
```

21.6.2 The *Runway* Class

Here is the code for the `Runway` class, take a look at it and then we will discuss it.

Runway

```
package airportSys; // add class to package
import java.io.Serializable;

/**
 * This class is used to store details of a single runway.
 *
 * @author Charatan and Kans
 * @version 1st August 2018
 */
public class Runway implements Serializable
{
 // attributes
 private int number;
 private boolean allocated;

 /**
  * Constructor sets the runway number
  * @param       numberIn             Used to set the runway number
  * @throws      AirportException     When the runway number is less than 1
  */
 public Runway (int numberIn)
 {
   if (numberIn <1)
   {
      throw new AirportException ("invalid runway number "+numberIn);
   }
    number = numberIn;
    allocated = false; // runway vacant initially
 }

 /**
   * Returns the runway number
   */
 public int getNumber()
 {
    return number;
 }

 /**
   * Checks if the runway has been allocated
   * @return     Returns true if the runway has been allocated and false otherwise
   */
 public boolean isAllocated()
 {
    return allocated;
 }

 /**
   * Records the runway as being booked
   */
 public void book()
 {
   allocated = true;
 }

 /**
  * Records the runway as being vacant
  */
 public void vacate()
 {
    allocated = false;
 }
}
```

There is not much that needs to be said about this class. As we may wish to save and load objects from our system, we have to remember to indicate that this class is `Serializable`.

```
public class Runway implements Serializable
```

Notice that we have defined this as a **`public`** class so that it is accessible outside of the package. We did this as a runway is a generally useful concept in many applications; declaring this class **`public`** allows it to be re-used outside of the `airportSys` package. In fact, we have declared most of our classes **`public`** for this reason.

21.6.3 The *Plane* Class

Here is the code for the `Plane` class. Have a close look at it and then we will discuss it.

Plane

```
package airportSys;

import java.io.Serializable;

/**
 * This class stores the details of a single plane
 *
 * @author Charatan and Kans
 * @version 2nd August 2018
 */
public class Plane implements Serializable
{

  // attributes
  private String flightNumber;
  private String city;
  private PlaneStatus status;
  private Runway theRunway; // to implement Runway association

  // methods

  /**
   * Constructor sets initial flight details of the plane requesting registration
   *
   * @param   flightIn      The flight number of the plane to register
   * @param   cityOfOrigin  The city of origin of the plane to register
   */
  public Plane(String flightIn, String cityOfOrigin)
  {
    flightNumber = flightIn;
    city = cityOfOrigin;
    status = PlaneStatus.DUE; // initial plane status set to DUE
    theRunway = null; // indicates no runway allocated
  }

  /**
   * Returns the plane's flight number
   */
  public String getFlightNumber()
  {
    return flightNumber;
  }

  /**
   * Returns the city associated with the flight
   */
  public String getCity()
  {
    return city;
  }

  /**
   * Returns the current status of the plane
   */
  public PlaneStatus getStatus()
  {
    return  status;
  }

  /**
   * Returns the runway allocated to this plane or null if no runway allocated
   */
  public Runway getRunway()
```

```
{
  return  theRunway;
}

/**
 * Returns the runway number allocated to this plane
 * @throws    AirportException if no runway allocated
 */
public int getRunwayNumber()
{
  if (theRunway == null)
  {
    throw new AirportException ("flight "+flightNumber+" has not been allocated a runway");
  }
  return theRunway.getNumber();
}

/**
 * Checks if the plane is allocated a runway
 * @return    Returns true if the plane has been allocated a runway
 *            and false otherwise
 */
public boolean isAllocatedARunway()
{
  return theRunway!=null;
}

/**
 * Allocates the given runway to the plane
 *
 * @throws     AirportException if runway parameter is null or runway already allocated
 */
public void allocateRunway(Runway runwayIn)throws AirportException
{
  if (runwayIn == null) // check runway has been sent
  {
    throw new AirportException ("no runway to allocate");
  }
  if (runwayIn.isAllocated())
  {
      throw new AirportException ("runway already allocate");
  }
  theRunway = runwayIn;
  theRunway.book();
}

/**
 * De-allocates the current runway
 *
 * @throws    AirportException if no runway allocated
 */
public void vacateRunway()
{
  if (theRunway==null)
  {
     throw new AirportException ("no runway allocated");
  }
  theRunway.vacate();
}

/**
 * Returns the String representation of the plane's status
 */
public String getStatusName()
{
    return status.toString();
}


/**
 * Upgrades the status of the plane.
 */
public void upgradeStatus()
{
  switch(status)
  {
     case DUE: status =PlaneStatus.WAITING; break;
     case WAITING: status =PlaneStatus.LANDED; break;
     case LANDED: status =PlaneStatus.DEPARTING; break;
     case DEPARTING: throw new AirportException("Cannot upgrade DEPARTING status");
  }
}

/**
 * Changes the city associated with the plane
 */
public void changeCity (String destination)
```

```
  {
    city = destination;
  }

  /**
  * Returns a string representation of a plane
  */
  @Override
  public String toString()
  {
    String out =    "number: "+flightNumber+   "\tcity: "+city+ "\tstatus: "+status;
    if (theRunway!=null)
    {
            out = out +"\trunway: "+theRunway;
    }
    return out;
  }


  /**
   * Checks whether the plane is equal to the given object
   */
  @Override
  public boolean equals(Object objIn)
  {
    if (objIn!=null)
    {
           Plane p = (Plane)objIn;
           return p.flightNumber.equals(flightNumber);
    }
    else
    {
           return false;
    }
 }

  /**
   * Returns a hashcode value
   */
  @Override
  public int hashCode()
  {
   return flightNumber.hashCode();
  }
}
```

Again, most of the points we raised with the `Runway` class are relevant to this `Plane` class. It needs to be `Serializable` and it is declared **`public`**.

Since `Plane` objects will be used in collection classes we have provided this class with an `equals` and a `hashCode` method. You can see that both of these methods make use of the plane's flight number.

In addition you should look at the way in which we dealt with the `status` attribute. During class design we declared this attribute to be of the enumerated `PlaneStatus` type, so it has been implemented as follows:

```
private PlaneStatus status;
```

We can then assign this attribute values from the enumerated `PlaneStatus` type. For example, in the constructor, we initialize the status of a plane to `DUE`:

```
public Plane(String flightNumberIn, String cityOfOrigin)
{
    flightNumber = flightNumberIn;
    city = cityOfOrigin;
    status = PlaneStatus.DUE;
    theRunway = null;
}
```

The `getStatus` method returns the value of the `status` attribute, so the appropriate return type is `PlaneStatus`:

```
public PlaneStatus getStatus()
{
    return status;
}
```

The `upgradeStatus` method is interesting as it demonstrates how the **`switch`** statement can be used with enumerated type variables such as `status`:

```
public void upgradeStatus()
{
    switch(status) // this is an enumerated type variable
    {
       // 'case' statements can check the different enumerated values
       case DUE: status = PlaneStatus.WAITING; break;
       case WAITING: status = PlaneStatus.LANDED; break;
       case LANDED: status = PlaneStatus.DEPARTING; break;
       case DEPARTING: throw new AirportException("Cannot upgrade DEPARTING status");
    }
}
```

Here we are upgrading the status of a plane as it makes its way to, and eventually from, the airport. Notice that the value of the `status` attribute is checked in the **`case`** statements, but this value is *not* appended onto the `PlaneStatus` class name. For example:

```
// just use a status name in 'case' test
case DUE: status = PlaneStatus.WAITING; break;
```

However, in all other circumstances, such as assigning to the `status` attribute, the enumerated value *does* have to be appended onto the `PlaneStatus` class name:

```
// use class + status name in all other circumstances
case DUE: status = PlaneStatus.WAITING; break;
```

You can see that we should not be upgrading the status of a plane if its current status is `DEPARTING`, so an exception is thrown in this case:

```
case DEPARTING: throw new AirportException ("Cannot upgrade DEPARTING status");
```

Before we leave this class, also notice that by adding a runway attribute, `theRunway`, into the `Plane` class we can send messages to (access methods of) a `Runway` object, for example:

```
public void allocateRunway(Runway runwayIn)
{
    // some code here
    theRunway.book(); // 'book' is a 'Runway' method
}
```

21.6.4 The *Airport* Class

The Airport class encapsulates the functionality of the system. It does not include the interface to the application. As we have done throughout this book, the interface of an application is kept separate from its functionality. That way, we can modify the way we choose to implement the functionality without needing to modify the interface, and vice versa. Examine it closely, being sure to read the comments, and then we will discuss it.

Airport

```
package airportSys;

import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;
import java.io.IOException;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.ObjectInputStream;

/**
 * Class to provide the functionality of the airport system
 *
 * @author Charatan and Kans
 * @version 4th August 2018
 */
public class Airport
{

  // attributes
  private Map<String, Plane> planes;  // registered planes
  private List<String> circlingQ; //flight numbers of circling planes
  private Runway []runway; // runways allocated to the airport

  // methods

  /**
   * This Constructor allows airport data to be loaded from a file
   *
   * @param   filenameIn The name of the file
   * @throws  IOException if problems with opening and loading given file
   * @throws  ClassNotFoundException if objects in file not of the right type
   */
  public Airport(String filenameIn)throws IOException, ClassNotFoundException
  {
      load(filenameIn);
  }


  /**
   * This Constructor creates an empty collection of planes,
   * and allocates runways to the airport
   *
   * @param   numIn The number of runways
   * @throws  AirportException if negative runway number used
   */
  public Airport (int numIn)
  {
    try
    {
        // intialise runways
        runway = new Runway [numIn];
        for (int i = 0; i<numIn; i++)
        {
          runway[i] = new Runway (i+1);
        }
        // initially no planes allocated to airport
        planes = new HashMap<>();
        circlingQ = new ArrayList<>();
    }
    catch (Exception e)
    {
        // notice throwing an excpetion from a catch clause
        throw new AirportException("Invalid Runway Number set, application closing");
    }
  }

  /**
   * Registers a plane with the airport
   *
   * @param   flightIn The plane's flight number
```

```
 * @param   cityOfOrigin The plane's city of origin
 * @throws  AirportException if flight number already registered.
 */
public void registerFlight (String flightIn, String cityOfOrigin)
{
   if (planes.containsKey(flightIn))
   {
      throw new AirportException ("flight "+flightIn+" already registered");
   }
   Plane newPlane = new Plane (flightIn, cityOfOrigin);
   planes.put(flightIn, newPlane);
}

 /**
 * Records a plane arriving at the airport
 *
 * @param   flightIn The plane's flight number
 * @throws  AirportException if plane not previously registered
 *            or if plane already arrived at airport
 */
public int arriveAtAirport (String flightIn)
{
   Runway vacantRunway = nextFreeRunway(); // get next free runway
   if (vacantRunway != null) // check if runway available
   {
      descend(flightIn, vacantRunway); // allow plane to descend on this runway
      return vacantRunway.getNumber(); // return booked runway number
   }
   else // no runway available
   {
      circle(flightIn); // plane must join circling queue
      return 0; // indicates no runway available to land
   }
}

/**
 * Records a plane landing on a runway
 *
 * @param   flightIn The plane's flight number
 * @param   runwayNumberIn The runway number the plane is landing on
 * @throws  AirportException if plane not previously registered
 *                           or if the runway is not allocated to this plane
 *                           or if plane has not yet signalled its arrival at the aiport
 *                           or if plane is already recorded as having landed.
 */
public void landAtAirport (String flightIn, int runwayNumberIn)
{
  Plane thisPlane = getPlane(flightIn); // throws AirportException if not registered
  if (thisPlane.getRunwayNumber()!= runwayNumberIn)
  {
    throw new AirportException ("flight "+flightIn+" should not be on this runway");
  }
  if (thisPlane.getStatus()== PlaneStatus.DUE)
  {
    throw new AirportException ("flight "+flightIn+" not signalled its arrival");
  }
  if (thisPlane.getStatus().compareTo(PlaneStatus.WAITING)>0)
  {
    throw new AirportException ("flight "+flightIn+" already landed");
  }
  thisPlane.upgradeStatus(); // upgrade status from WAITING to LANDED
}

/**
 * Records a plane boarding for take off
 *
 * @param   flightIn The plane's flight number
 * @param   destination The city of destination
 * @throws  AirportException  if plane not previously registered
 *                            or if plane not yet recorded as landed
 *                            or if plane already recorded as ready for take off
 */
public void readyForBoarding(String flightIn, String destination)
{
  Plane thisPlane = getPlane(flightIn);// throws AirportException if not registered
  if (thisPlane.getStatus().compareTo(PlaneStatus.LANDED)<0)
  {
    throw new AirportException ("flight "+flightIn+" not landed");
  }
  if (thisPlane.getStatus()== PlaneStatus.DEPARTING)
  {
    throw new AirportException ("flight "+flightIn+" already registered to depart");
```

```
    }
    thisPlane.upgradeStatus(); // upgrade status from LANDED to DEPARTING
    thisPlane.changeCity(destination); // change city of origin to city of destination
  }

  /**
   * Records a plane taking off from the aiprort
   *
   * @param   flightIn The plane's flight number
   * @throws  AirportException    if plane not previously registered
   *                              or if plane not yet recorded as landed
   *                              or if the plane not previously recorded as taken off
   */
  public Plane takeOff (String flightIn)
  {
    leave(flightIn); // remove from plane register
    Plane nextFlight = nextToLand(); // return next circling plane to land
    if (nextFlight != null) // check circling flight exists
    {
      Runway vacantRunway = nextFreeRunway();
      descend(nextFlight.getFlightNumber(), vacantRunway); // allocate runway to circling plane
      return nextFlight; // send back details of next plane to land
    }
    else // no circling planes
    {
      return null;
    }
  }


  /**
   * Returns the set of planes due for arrival
   */
  public Set<Plane> getArrivals()
  {
      Set<Plane> planesOut = new HashSet<>();
      Set<String> items = planes.keySet();
      for(String thisFlight: items)
      {
        Plane thisPlane = planes.get(thisFlight);
        if (thisPlane.getStatus() != PlaneStatus.DEPARTING)
        {
              planesOut.add(thisPlane); //add to set
        }
      }
      return planesOut;
   }

  /**
   * Returns the set of planes due for departure
   */
  public Set<Plane> getDepartures()
  {
      Set<Plane> planesOut = new HashSet<>();
      Set<String> items = planes.keySet();
      for(String thisFlight: items)
      {
        Plane thisPlane = planes.get(thisFlight);
        if (thisPlane.getStatus()==PlaneStatus.DEPARTING)
        {
              planesOut.add(thisPlane); // add to set
        }
      }
      return planesOut;
  }

  /**
   * Returns the number of runways
   */
  public int getNumberOfRunways()
  {
    return runway.length;
  }

  /**
   * Saves airport object to file
   *
   * @param  fileIn The name of the file
   * @throws IOException if problems with opening and saving to given file
   */
  public void save(String fileIn)throws IOException
  {
      // notice try-with-resources to ensure file closes safely
      try ( FileOutputStream fileOut = new FileOutputStream(fileIn);
            ObjectOutputStream objOut = new ObjectOutputStream (fileOut))
      {
          objOut.writeObject(planes);
```

```
        objOut.writeObject(circlingQ);
        objOut.writeObject(runway);
    }
}

/**
 * Loads airport object from file
 *
 * @param   fileName The name of the file
 * @throws  IOException if problems with opening and loading given file
 * @throws  ClassNotFoundException if objects in file not of the right type
 */
public void load (String fileName) throws IOException, ClassNotFoundException
{
    // notice try-with-resources to ensure file closes safely
    try ( FileInputStream fileInput = new FileInputStream(fileName);
          ObjectInputStream objInput = new ObjectInputStream (fileInput))
    {
        planes = (Map<String, Plane>) objInput.readObject();
        circlingQ = (List<String>) objInput.readObject();
        runway = (Runway[])objInput.readObject();
    }
}


 // helper method to find next free runway
 private Runway nextFreeRunway()
 {
    for (Runway nextRunway : runway)
    {
        if (!nextRunway.isAllocated())
        {
            return nextRunway;
        }
    }
    return null;
 }

/**
 * Returns the registered plane with the given flight number
 *
 * @throws  AirportException if flight number not yet registered.
 */
 private Plane getPlane(String flightIn)
 {
     if (!planes.containsKey(flightIn))
     {
        throw new AirportException ("flight "+flightIn+" has not yet registered");
     }
     return planes.get(flightIn);
  }

/**
 * Records a plane descending on a runway
 *
 * @param   flightIn The plane's flight number
 * @param   runwayIn The runway the plane will be landing on
 * @throws  AirportException    if plane not previously registered
 *                              or if plane already arrived at airport
 *                              or if plane already allocated a runway
 */
 private void descend (String flightIn, Runway runwayIn)
 {
  Plane thisPlane = getPlane(flightIn);// throws AirportException if not registered
  if (thisPlane.getStatus().compareTo(PlaneStatus.WAITING)>0)

  {
      throw new AirportException
        ("flight "+flightIn+" already at airport has status of "+thisPlane.getStatusName());
  }
  if (thisPlane.isAllocatedARunway())
  {
     throw new AirportException
        ("flight "+flightIn+" has already been allocated runway "+thisPlane.getRunwayNumber());
  }
  thisPlane.allocateRunway(runwayIn);
  if (thisPlane.getStatus()==PlaneStatus.DUE) // updraged status from DUE to WAITING
  {
      thisPlane.upgradeStatus();
  }
}

/**
 * Records a plane joining the planes circling the airport
 *
 * @param   flightIn The plane's flight number
 * @throws  AirportException if plane not previously registered
```

```
    *                          or if plane already arrived
    */
   private void circle (String flightIn)
   {
     Plane thisPlane = getPlane(flightIn); // throws AirportException if not registered
     if (thisPlane.getStatus()!= PlaneStatus.DUE)
     {
       throw new AirportException ("flight "+flightIn+" already at airport");
     }
     thisPlane.upgradeStatus(); // updraged status from DUE to WAITING
     circlingQ.add(flightIn);
   }


   /**
    * Records a plane taking off from the aiprort
    *
    * @param   flightIn The plane's flight number
    * @throws  AirportException if plane not plane not not previously registered
    *                 or if plane not yet recorded as landed
    *                 or if the plane has not previously been recorded as ready for take off
    */
   private void leave (String flightIn)
   {
     // get plane associated with given flight number
     Plane thisPlane = getPlane(flightIn);// throws AirportException if not registered
     // throw exceptions if plane is not ready to leave airport
     if (thisPlane.getStatus().compareTo(PlaneStatus.LANDED)<0)
     {
       throw new AirportException ("flight "+flightIn+" not yet landed");
     }
     if (thisPlane.getStatus()==PlaneStatus.LANDED)
     {
         throw new AirportException ("flight "+flightIn+" must register to board");
     }
     // process plane leaving airport
     thisPlane.vacateRunway(); // runway now free
     planes.remove(flightIn); // remove plane from list
   }

   /**
    * Locates next circling plane to land
    *
    * @return      Returns the next circling plane to land
    *              or null if no planes
    */
   private Plane nextToLand()
   {
     if (!circlingQ.isEmpty()) // check circling plane exists
     {
       String flight =  circlingQ.get(0);
       circlingQ.remove(flight);
       return getPlane(flight); // could throw exception of not in list
     }
     else // no circling plane
     {
       return null;
     }
   }

}
```

There is not a lot that is new here, but we draw your attention to a few implementation issues.

First, notice we have provided two constructors, as specified in Fig. 21.2. The first receives the name of a file and loads data from this given file (using the **private** `load` method to be found later in this `Airport` class); the associated exceptions are passed on if an error occurs during this process:

```
public Airport(String filenameIn)throws IOException, ClassNotFoundException
{
      load(filenameIn); // call private method to load airport data
}
```

The second constructor receives the number of runways associated with this airport and initialises all data to be empty.

```
public Airport (int numIn)
{
  try
  {
      // intialise runways
      runway = new Runway [numIn];
      for (int i = 0; i<numIn; i++)
      {
        runway[i] = new Runway (i+1);
      }
      // initially no planes allocated to airport
      planes = new HashMap<>();
      circlingQ = new ArrayList<>();
  }
  catch (Exception e)
  {
      // notice we have thrown our user-defined exception from this catch clause
      throw new AirportException("Invalid Runway Number set, application closing");
  }
}
```

Within this constructor you can see that we catch a general exception, in case something goes wrong when allocating the array, and throw our application-specific exception when this occurs. This can be useful if we wish to suppress the name of Java specific expectations and stick to the names of our own user-defined exceptions.

Most of the other **`public`** methods simply check for a list of exceptions, and then upgrade the plane's status as it makes its way to and eventually from the airport.

Here, for example, is the method that records a plane that has previously landed at the airport, being ready to board new passengers for a new destination:

```
/**
   * Records a plane boarding for take off
   *
   * @param   flightIn The plane's flight number
   * @param   destination The city of destination
   * @throws  AirportException if plane not previously registered
   *                             or if plane not yet recorded as landed
   *                             or if plane already recorded as ready for take off
   */
  public void readyForBoarding(String flightIn, String destination)
  {
    Plane thisPlane = getPlane(flightIn);// throws AirportException if not registered
    if (thisPlane.getStatus().compareTo(PlaneStatus.LANDED)<0)
    {
      throw new AirportException ("flight "+flightIn+" not landed");
    }
    if (thisPlane.getStatus()== PlaneStatus.DEPARTING)
    {
      throw new AirportException ("flight "+flightIn+" already registered to depart");
    }
    thisPlane.upgradeStatus(); // upgrade status from LANDED to DEPARTING
    thisPlane.changeCity(destination); // change city of origin to city of destination
  }
```

The first thing we need to do in this method is to check whether or not an `AirportException` needs to be thrown. The `Javadoc` comments make clear that there are three situations in which we need to throw such an exception.

First, an exception needs to be thrown if the given flight number has not been registered with the airport. At some point we also need to retrieve the `Plane` object from this flight number. Calling the helper method `getPlane` will do both of these things for us, as it throws an `AirportException` if the flight is not registered.

```
// retrieves plane or throws AirportException if flight is not registered
Plane thisPlane = getPlane(flightIn);
```

To check for the remaining exceptions we need to check that the plane currently has the appropriate status to start taking on passengers. The `getStatus` method of a plane returns the status of a plane for us. We know from the previous section that this method returns a value of the enumerated type `PlaneStatus`.

As well as having a `toString` method generated for you when you declare an enumerated type such as `PlaneStatus`, a `compareTo` method (to allow for comparison of two enumerated values) is also generated. This method works in exactly the same way as the `compareTo` method you met when looking at `String` methods. That is, it returns 0 when the two values are equal, a number less than 0 when the first value is less than the second value and a number greater than 0 when the first value is greater than the second value. One enumerated type value is considered *less than* another if it is listed before that value in the original type definition. So, in our example, `DUE` is *less than* `WAITING`, which is *less than* `LANDED` and so on. If a plane has a status that is less than `LANDED` it has not yet landed, so cannot be ready to board passengers—an `AirportException` is thrown:

```
// use 'compareTo' method to compare two status values
if (thisPlane.getStatus().compareTo(PlaneStatus.LANDED)<0)
{
   throw new AirportException ("flight "+flightIn+" not yet landed");
}
```

We also need to throw an `AirportException` if the plane already has a status of `BOARDING`. Although the `compareTo` method can be used to check for equality as well, with most classes it is common to use an `equals` method to do this. An `equals` method is generated for any enumerated type, such as `PlaneStatus`, that you define. However, because of the way enumerated types are implemented in Java, the simple equality operator (==) can also be used to check for equality:

```
// equality operator can be used to check if 2 enumerated values are equal
if (thisPlane.getStatus()== PlaneStatus.DEPARTING)
{
   throw new AirportException ("flight "+flightIn+" already registered to depart");
}
```

Having checked for exceptions, we can now indicate that this plane is ready for boarding by upgrading its status (from `LANDED` to `DEPARTING`), and by recording the flights new destination city:

```
// we have cleared all the exceptions so we can update flight details now
thisPlane.upgradeStatus(); // upgrades status from LANDED to DEPARTING
thisPlane.changeCity(destination); // changes city to destination city
```

The inequality operator (`!=`) can be used with enumerated types, to check for inequality of two enumerated type values. An example of this can be seen in the implementation of the `arrivals` method:

```
/**
 * Returns the set of planes due for arrival
 */
public Set<Plane> getArrivals()
{
   Set<Plane> planesOut = new HashSet<Plane>(); // create empty set
   Set<String> items = planes.keySet(); // get all flight numbers
   for(String thisFlight: items) // check status of all
   {
      Plane thisPlane = planes.get(thisFlight);
      if (thisPlane.getStatus() != PlaneStatus.DEPARTING)
      {
         planesOut.add(thisPlane); // add to set
      }
   }
   return planesOut;
}
```

Here we create an empty set of planes. We then add planes into this set if they do not have a status of DEPARTING:

```
// use inequality operator to check if status does not equal some value
if (thisPlane.getStatus() != PlaneStatus.DEPARTING)
{
   planesOut.add(thisPlane);
}
```

We have used an enhanced for loop in this method, but you might consider using a forEach loop. We leave this as end of chapter exercises for you.

Before we leave this section, let us take a look at the save and load methods that allow us to save and load the attributes in our application. We have three attributes here, planes (the Map of registered planes), circlingQ (the List of flight numbers of the planes circling the airport) and runway (the array of runways).

Since we have declared our Plane and Runway classes to be Serializable, and because enumerated types such as PlaneStatus and collection classes such as Map and List are already Serializable, it is a simple matter to write these objects to a file, and read them from a file. Here is the save method:

```
/**
 * Saves airport object to file
 *
 * @param     fileIn The name of the file
 * @throws    IOException if problems with opening and saving to given file
 */
 public void save(String fileIn)throws IOException
 {
     // notice try-with-resources to ensure file closes safely
     try ( FileOutputStream fileOut = new FileOutputStream(fileIn);
           ObjectOutputStream objOut = new ObjectOutputStream (fileOut))
     {
         objOut.writeObject(planes);
         objOut.writeObject(circlingQ);
         objOut.writeObject(runway);
     }
 }
```

You can see we have used a *try-with-resources* construct here to ensure the file is closed once the method terminates.

Here is the `load` method:

```
/**
 * Loads airport object from file
 *
 * @param   fileName The name of the file
 * @throws  IOException if problems with opening and loading given file
 * @throws  ClassNotFoundException if objects in file not of the right type
 */
public void load (String fileName) throws IOException, ClassNotFoundException
{
    // notice try-with-resources to ensure file closes safely
    try ( FileInputStream fileInput = new FileInputStream(fileName);
          ObjectInputStream objInput = new ObjectInputStream (fileInput))
    {
        planes = (Map<String, Plane>) objInput.readObject();
        circlingQ = (List<String>) objInput.readObject();
        runway = (Runway[])objInput.readObject();
    }
}
```

Again, we have used a *try-with-resources* construct to ensure our file is closed upon termination. Notice that when we load the attributes from file, we must indicate their type. The collection class types need to be marked using the generics mechanism:

```
// indicate the type of each collection using generics mechanism
planes = (Map<String, Plane>) objInput.readObject();
circlingQ = (List<String>) objInput.readObject();
runway = (Runway[])objInput.readObject();
```

There is nothing particularly new in the remaining methods. Take a look at the comments provided to follow their implementation.

21.7 Testing

In Chaps. 11 and 12 we looked at the concepts of unit testing and integration testing. We have left unit testing to you as a practical task, but we will spend a little time here considering integration testing. A useful technique to devise test cases during integration testing is to review the behaviour specifications of use cases, derived during requirements analysis.

Remember, a use case describes some useful service that the system performs. The behaviour specifications capture this service from the point of view of the user. When testing the system you take the place of the user, and you should ensure that the behaviour specification is observed.

Often, there are several routes through a single use case. For example, when registering a plane, either the plane could be successfully registered, or an error is indicated. Different routes through a single use case are known as different

scenarios. During integration you should take the place of the user and make sure that you test *each* scenario for *each* use case. Not surprisingly, this is often known as **scenario testing**. As an example, reconsider the "*Record flight's request to land*" use case:

An air traffic controller records an incoming flight entering airport airspace, and requesting to land at the airport, by submitting its flight number. As long as the plane has previously registered with the airport, the air traffic controller is given an unoccupied runway number on which the plane will have permission to land. If all runways are occupied however, this permission is denied and the air traffic controller is informed to instruct the plane to circle the airport. If the plane has not previously registered with the airport an error is signalled.

From this description three scenarios can be identified:

Scenario 1
An air traffic controller records an incoming plane entering airport airspace and requesting to land at the airport, by submitting its flight number, and is given an unoccupied runway number on which the plane will have permission to land.

Scenario 2
An air traffic controller records an incoming plane entering airport airspace and requesting to land at the airport, by submitting its flight number. The air traffic controller is informed to instruct the plane to circle the airport as all runways are occupied.

Scenario 3
An air traffic controller records an incoming plane entering airport airspace and requesting to land at the airport, by submitting its flight number. An error is signalled as the plane has not previously registered with the airport.

Similar scenarios can be extracted for each use case. During testing we should walk through each scenario, checking whether the outcomes are as expected.

21.8 Design of the JavaFX Interface

Figure 21.4 illustrates the interface design we have chosen for the *Airport* application. A few new JavaFX features have been highlighted.

Apart from the features highlighted in Fig. 21.4, the remaining JavaFX components will be familiar to you. The three new JavaFX features are: a layout component known as a **tabbed pane**; some text that appears when you keep your cursor over a component—known as that component's **tool tip** and a `Stage` with

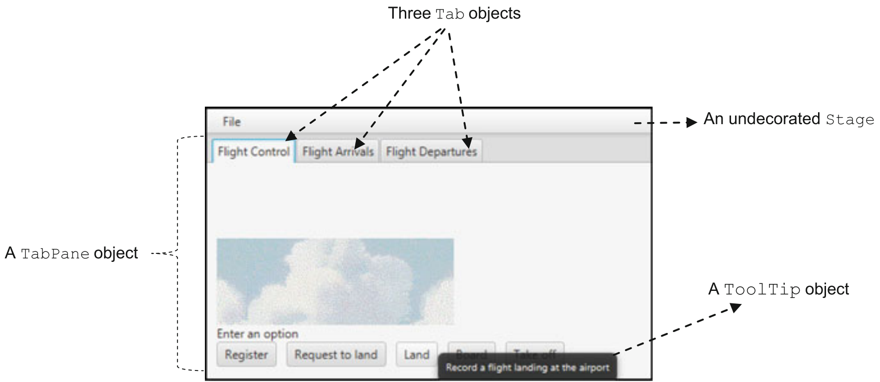


Fig. 21.4 Some new JavaFX features in the *Airport* JavaFX interface

no icons for minimising/maximising/closing—known as an **undecorated** stage. We will return to look at tool tips and undecorated stages later in the chapter but we will look at a tabbed pane now—it is implemented in JavaFX using a `TabPane` class.

21.9 The *TabPane* Class

The `TabPane` class provides a very useful JavaFX component for organizing the user interface. You can think of a `TabPane` component as a collection of overlapping tabbed "cards", on which you place other user interface components. A particular card is revealed by clicking on its **tab**. This allows certain parts of the interface to be kept hidden until required, thus reducing screen clutter.

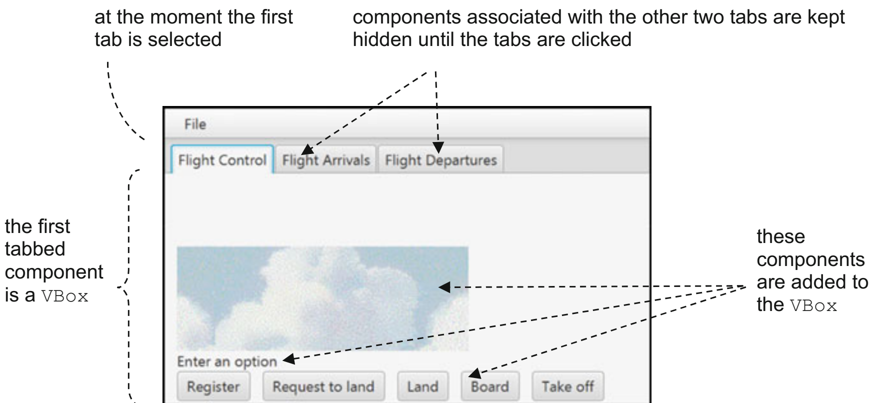


Fig. 21.5 A `TabPane` allows parts of the interface to be revealed selectively

A TabPane component can consist of any number of tabbed cards. Each card is actually a *single* component of your choice. If you use a container component such as a VBox, you can effectively associate many components with a single tab (see Fig. 21.5).

We can construct a TabPane component by calling the empty constructor as follows:

```
TabPane tabbedPane = new TabPane();
```

A TabPane can be associated with any number of individual tabbed cards. To do this we use the Tab class for each individual tabbed card. We need three tabbed cards, one for the main screen shown in Fig. 21.5, one for the arrivals information and one for the departures information:

```
Tab tab1 = new Tab("Flight Control"); // main flight control tab
Tab tab2 = new Tab("Flight Arrivals"); // arrivals tab
Tab tab3 = new Tab("Flight Departures"); // departures tab
```

The strings provided to the constructors are the titles displayed on each tab.

We can now add tabbed components to the TabPane. When adding a tabbed component to a TabPane, you call the getTabs method to access the link to the collection of tabbed cards and then the tabs themselves can be set using the addAll method:

```
tabbedPane.getTabs().addAll(tab1, tab2, tab3);
```

The order in which the tabs are given to the addAll method is the order in which they will appear on the screen.[1] As we mentioned, each tabbed card is associated with a single component. So to add multiple components to a tab we can use a container such as a VBox or a HBox. For example, to create the main tab shown in Fig. 21.5, we have 3 components highlighted—a cloud image (named imageView in the code), a label (named label in the code) and a collection of buttons (stored in a HBox named controls in the code). We add all three of these to a VBox (named box in the code):

```
// add cloud image, label and button collection to VBox
box.getChildren().addAll(imageView,label, controls);
```

[1]By default, the tabs you add will appear at the top left of the TabPane (as in Fig. 21.4). A setSide method can be used to choose an alternative side (the top right, the bottom, the left or the right).

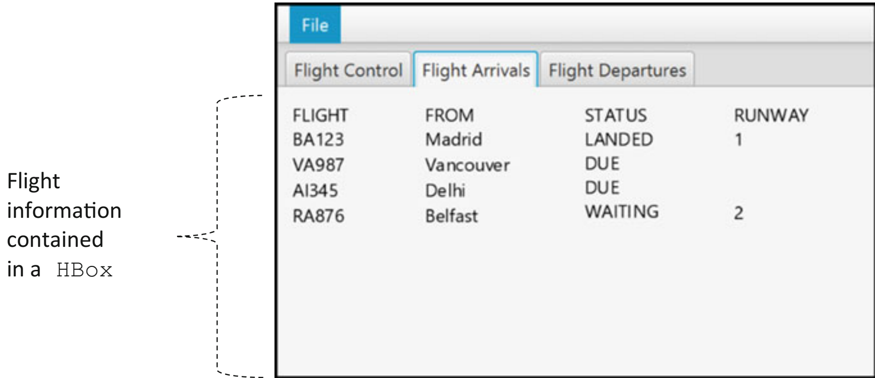


Fig. 21.6 Both "Flight Arrivals" and "Flight Departures" tabs consist of flight information in a `HBox`

We then use the `setContent` method of a tabbed card to add this `VBox` component to a given tab:

```
// add VBox to tab1
tab1.setContent(box);
```

The "Flight Arrivals" and "Flight Departures" each contain a `HBoxes` (each of which contains a series of `VBox` columns) to display arrivals and departure information respectively. Figure 21.6 shows the airport GUI after selecting the "Flight Arrivals" tab.

21.10 The *AirportFrame* Class

We now present the complete code for the JavaFX class. Take a look at it and then we will point out a few features:

AirportFrame

```
package airportSys; // add class to package

import java.util.Set;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.application.Platform;
import javafx.geometry.Insets;
```

```
import javafx.geometry.Pos;
import javafx.scene.Scene;
 mport javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Button;
import javafx.scene.control.ButtonType;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.scene.control.TextInputDialog;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.control.Tooltip;
import javafx.scene.layout.Border;
import javafx.scene.layout.BorderStroke;
import javafx.scene.layout.BorderStrokeStyle;
import javafx.scene.layout.BorderWidths;
import javafx.scene.layout.CornerRadii;
import javafx.scene.paint.Color;
import javafx.stage.StageStyle;

/**
 * Class to provide the JavaFX interface of the airport system
 *
 * @author Charatan and Kans
 * @version 6th August 2018
 */

public class AirportFrame extends Application
{
    // declare Airport object
    private Airport myAirport;
    // additional data required for the airport system
    private int numberOfRunways ;
    private final String FILENAME = "airport.dat";

    // create arrival and departure visual components that need global access

    // arrivals information displayed in a HBox
    private HBox arrivals = new HBox(50);
    // include columns for arrivals information
    private   VBox arrivalsColumn1 = new VBox();
    private   VBox arrivalsColumn2 = new VBox();
    private   VBox arrivalsColumn3 = new VBox();
    private   VBox arrivalsColumn4 = new VBox();
    // departures information displayed in a HBox
    private   HBox departures = new HBox(60);
    // include columns for departures information
    private   VBox departuresColumn1 = new VBox();
    private   VBox departuresColumn2 = new VBox();
    private   VBox departuresColumn3 = new VBox();

    // methods

    /**
     * The start method to initialise the screen and the airport data
     *
     * @param stage The Stage object
     */
    @Override
    public void start(Stage stage)
    {
        // check if data is to be loaded from file
        Alert alert = new Alert(  AlertType.INFORMATION, "Do you want to restore your data?",
                                  ButtonType.YES, ButtonType.NO);
        String response = alert.showAndWait().get().getText();
        if (response.equals("Yes")) // load data from file
        {
            try
            {
                myAirport = new Airport(FILENAME); // call file loading constructor
                listArrivals(); // update arrivals tab
                listDepartures(); // update departures tab
                showInfo("Planes loaded");
            }
            catch (Exception e) // file loading errors
            {
                showError("File Opening error");
                System.exit(1); // indicates exit with error
            }
```

```
        }
        else // initialise an empty airport
        {
            numberOfRunways = getNumberOfRunways(); // request number of runways
            try
            {
                myAirport = new Airport (numberOfRunways); // create an empty Airport object
            }
            catch (AirportException ae) // error initialising Airport object
            {
                showError(ae.getMessage());
                System.exit(1); // indicates exit with error
            }
            catch (Exception e) // in case of any unforseen error
            {
                showError(e.getMessage());
                System.exit(1); // indicates exit with error
            }
        }
        // set up three Tab objects in a TabPane
        TabPane tabbedPane = new TabPane();
        Tab tab1 = new Tab("Flight Control"); // main flight control tabs
        Tab tab2 = new Tab("Flight Arrivals"); // arrivals tab
        Tab tab3 = new Tab("Flight Departures"); // departures tab
        tabbedPane.getTabs().addAll(tab1, tab2, tab3);
        // ensure tabs remain open
        tab1.setClosable(false);
        tab2.setClosable(false);
        tab2.setClosable(false);

        // creat a VBox to hold all scene components
        VBox root = new VBox();

        // set up menu bar and items
        MenuBar bar = new MenuBar();
        bar.setMinHeight(25);
        Menu item = new Menu("File");
        Menu saveAndContinueOption = new Menu("Back-up and continue");
        Menu saveAndExitOption = new Menu("Back-up and exit");
        Menu exitWithoutSavingOption = new Menu("Exit without backing-up");
        item.getItems().addAll(saveAndContinueOption, saveAndExitOption, exitWithoutSavingOption);
        bar.getMenus().add(item);

        // create and customise a VBox to organise flight control screen
        VBox box = new VBox();
        box.setPadding(new Insets(10));
        box.setMinHeight(215);
        // add Vbox to tab1
        tab1.setContent(box);
        box.setAlignment(Pos.BOTTOM_LEFT);
        // create a cloud image
        Image image = new Image("clouds.png");
        ImageView imageView = new ImageView(image);
        // create an instructional label
        Label label = new Label("Enter an option");
        // create a HBox to hold main flight control buttons
        HBox controls = new HBox(10);
        // create flight control buttons and add tooltips
        Button button1 = new Button("Register");
        button1.setTooltip(new Tooltip("Register a flight with the airport"));
        Button button2 = new Button("Request to land");
        button2.setTooltip(new Tooltip("Record a flight requesting to land at the airport"));
        Button button3 = new Button("Land");
        button3.setTooltip(new Tooltip("Record a flight landing at the airport"));
        Button button4 = new Button("Board");
        button4.setTooltip(new Tooltip("Record a landed flight ready for boarding new passengers"));
        Button button5 = new Button("Take off");
        button5.setTooltip(new Tooltip("Record a flight leaving the airport"));
        // add buttons to HBox
        controls.getChildren().addAll(button1, button2, button3, button4, button5);
        // add cloud image, label and button collection to VBox
        box.getChildren().addAll(imageView,label, controls);
        try
        {
            // code button responses by calling private methods
            button1.setOnAction(e -> register());
            button2.setOnAction(e -> requestToLand());
            button3.setOnAction(e -> land());
            button4.setOnAction(e -> board());
            button5.setOnAction(e -> takeOff());
            // code menu responses
            saveAndContinueOption.setOnAction(e -> save(FILENAME));
            saveAndExitOption.setOnAction(e ->{
                                                    save(FILENAME);
                                                    Platform.exit();
                                                  });
            exitWithoutSavingOption.setOnAction(e -> exitWithoutSaving());
```

```
        }
        catch(Exception e) // for any unforseen errors
        {
            showError("Invalid Operation");
        }

        // customise look of arrivals tab
        arrivals.setPadding(new Insets(10));
        arrivals.getChildren().addAll(     arrivalsColumn1, arrivalsColumn2, arrivalsColumn3,
                                           arrivalsColumn4);
        tab2.setContent(arrivals);
        // customise look of departures tab
        departures.setPadding(new Insets(10));
        departures.getChildren().addAll(departuresColumn1, departuresColumn2, departuresColumn3);
        tab3.setContent(departures);
        // customise root object and add menu and tabbed pane
        root.setBorder( new Border(new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID,
                        new CornerRadii(0), new BorderWidths(2))));
        root.getChildren().addAll(bar, tabbedPane);
        // customise frame
        Scene scene = new Scene(root,450, 275);
        stage.setScene(scene);
        stage.setTitle("Airport System");
        stage.initStyle(StageStyle.UNDECORATED); // for undecorated frame

        stage.show();
    }

    /**
     *  Private method to request and return the number of runways
     */
    private int getNumberOfRunways()
    {
        TextInputDialog dialog = new TextInputDialog();
        dialog.setHeaderText("Enter number of runways");
        dialog.setTitle("Runway Information Request");

        String response = dialog.showAndWait().get();

        if (!response.equals("")) // check for empty string
        {
            return Integer.parseInt(response);
        }
        else
        {
            return -1; // to indicate no runway set
        }
    }

    /**
     *  Private method to register new flight with the airport
     */
    private void register()
    {
        String flightNo, city;
        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Enter flight number");
            dialog.setTitle("Registration form");
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            dialog = new TextInputDialog();
            dialog.setHeaderText("Enter city of origin");
            dialog.setTitle("Registration form");
            city = dialog.showAndWait().get();

            // throws AirportException if no city entered
            checkIfEmpty(city, "No city entered");

            // register flight
            myAirport.registerFlight(flightNo, city);
            showInfo("confirmed:\nflight "+flightNo +" registered from "+city);
        }
        catch(AirportException ae) // catch airport exceptions
        {
            showError(ae.getMessage());
        }
        listArrivals();  // update arrivals tab
    }

    /**
```

```
     *  Private method to record a flight's request to land
     */
    private void requestToLand()
    {
        String flightNo, message;

        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Enter flight number");
            dialog.setTitle("Request to land form");
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            // record flight's request to land and get runway number
            int runway = myAirport.arriveAtAirport(flightNo);
            // check runway number
            if (runway == 0)
            {
                message = "no runway available, circle the airport";
            }
            else
            {
               message = " land on runway "+runway;
            }
            showInfo("confirmed:\nflight "+flightNo + message);
        }
        catch(AirportException ae) // catch airport exceptions
        {
            showError(ae.getMessage());
        }
        listArrivals(); // update arrivals tab
    }

    /**
     *  Private method to record a flight landing at the airport
     */
    private void land()
    {
        String flightNo, runwayIn;
        int runway;

        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Enter flight number");
            dialog.setTitle("Landing form");
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            dialog = new TextInputDialog();
            dialog.setHeaderText("Enter runway number");
            dialog.setTitle("Landing form");
            runwayIn = dialog.showAndWait().get();

            // throws AirportException if no runway enetered
            checkIfEmpty(flightNo, "No flight number entered");

            // convert runway to an integer
            runway = Integer.parseInt(runwayIn);
            // record flight landing
            myAirport.landAtAirport(flightNo, runway);
            showInfo("confirmed:\nflight "+flightNo +" landed on runway "+runway);
        }
        catch (AirportException ae) // catch airport exceptions
        {
           showError(ae.getMessage());
        }
        listArrivals(); // update arrivals tab
    }

    /**
     *  Private method to register a flight boarding passengers at the airport
     */
    private void board ()
    {
        String flightNo, city;

        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Fight number");
            dialog.setTitle("Boarding form");
```

```
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            dialog = new TextInputDialog();
            dialog.setHeaderText("Enter destination city");
            dialog.setTitle("Boarding form");
            city = dialog.showAndWait().get();

            // throws AirportException if no city entered
            checkIfEmpty(city, "No city entered");

            // record flight boarding
            myAirport.readyForBoarding(flightNo, city);
            showInfo("confirmation:\nflight "+flightNo+" boarding to "+city);
        }
        catch (AirportException ae) // catch airport exceptions
        {
           showError(ae.getMessage());
        }
        listArrivals(); // update arrivals tab
        listDepartures(); // update departures tab
    }

    /**
     *  Private method to register a flight leaving the airport
     */
    private void takeOff ()
    {
        String flightNo;

        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Fight number");
            dialog.setTitle("Take off form");
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            // record flight taking off
            myAirport.takeOff(flightNo);
            showInfo("confirmation:\nflight "+flightNo+" Removed from system");
        }
        catch (AirportException ae) // catch airport exceptions
        {
           showError(ae.getMessage());
        }
        listDepartures(); // update departures tab
    }

    /**
     *  Private method to update arrivals tab information
     */
    private void listArrivals()
    {
      // get arrivals information
      Set<Plane> arrivalsList = myAirport.getArrivals();
      // clear current arrivals information
      arrivalsColumn1.getChildren().clear();
      arrivalsColumn2.getChildren().clear();
      arrivalsColumn3.getChildren().clear();
      arrivalsColumn4.getChildren().clear();
      arrivalsColumn1.getChildren().add(new Text("FLIGHT"));
      arrivalsColumn2.getChildren().add(new Text("FROM"));
      arrivalsColumn3.getChildren().add(new Text("STATUS"));
      arrivalsColumn4.getChildren().add(new Text("RUNWAY"));
      // re-populate arrivals information
      for (Plane thisPlane: arrivalsList)
      {
        arrivalsColumn1.getChildren().add(new Text(thisPlane.getFlightNumber()));
        arrivalsColumn2.getChildren().add(new Text(thisPlane.getCity()));

        arrivalsColumn3.getChildren().add(new Text(thisPlane.getStatusName()));
        try
        {
            // throws exception if no runway set
            arrivalsColumn4.getChildren().add(
                                        new Text(Integer.toString(thisPlane.getRunwayNumber())));
        }
        catch(Exception e) // catch exception and leave runway column blank
        {
             arrivalsColumn4.getChildren().add(new Text(""));
        }
     }
```

```
}

/**
 *  Private method to update departures tab information
 */
private void listDepartures()
{
  // get departures information
  Set<Plane> departuresList = myAirport.getDepartures();
  // clear current departures information
  departuresColumn1.getChildren().clear();
  departuresColumn2.getChildren().clear();
  departuresColumn3.getChildren().clear();
  departuresColumn1.getChildren().add(new Text("FLIGHT"));
  departuresColumn2.getChildren().add(new Text("TO"));
  departuresColumn3.getChildren().add(new Text("RUNWAY"));
  // re-populate departures information
  for (Plane thisPlane: departuresList)
  {
    departuresColumn1.getChildren().add(new Text(thisPlane.getFlightNumber()));
    departuresColumn2.getChildren().add(new Text(thisPlane.getCity()));

    try
    {
        // throws exception if no runway set
        departuresColumn3.getChildren().add(
                                      new Text(Integer.toString(thisPlane.getRunwayNumber())));
    }
    catch(Exception e) // catch exception and leave runway column blank
    {
        departuresColumn3.getChildren().add(new Text(""));
    }
 }
}

/**
 *  Private method to exit application without saving data
 */
private void exitWithoutSaving()
{
    Alert alert = new Alert(  AlertType.WARNING, "Are you sure? Your work could be lost.",
                              ButtonType.YES, ButtonType.CANCEL);
    alert.setTitle("Confirmation reqired");
    String response = alert.showAndWait().get().getText();

    if(response.equals("Yes"))
    {
        Platform.exit();
    }
}

/**
 *  Private method to load airport data from a file
 *  @param   fileName The name of the file to open
 */
private void open(String fileName)
{
    try
    {
        myAirport.load(fileName); // may throw an exception
        listArrivals(); // update arrivals tab
        listDepartures(); // update departures tab
        showInfo("Planes Loaded");
    }
    catch (Exception e) // catch file related exceptions
    {
        showError("File Opening error");
        System.exit(1); // indicates exit with error
    }
}

/**
 *  Private method to save airport data to a file
 *  @param   fileName The name of the file to save
 */
     private void save(String fileName)
{
    try
    {
        myAirport.save(fileName); // may throw exception
        showInfo("Planes saved");
    }
    catch (Exception e) // catch file related exceptions
    {
        showError("Error saving data");
    }
```

```
    }

    /**
     *  Private method to show an error message
     *  @param   msg The error message
     */
    private void showError(String msg)
    {
        Alert alert = new Alert(AlertType.ERROR);
        alert.setHeaderText("Airport Error Alert");
        alert.setContentText(msg);
        alert.showAndWait();
    }

    /**
     *  Private method to show an information message
     *  @param   msg The information message
     */
    private void showInfo(String msg)
    {
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setHeaderText("Airport Information Alert");
        alert.setContentText(msg);
        alert.showAndWait();
    }

    /**
     *  Private method to check if a string is empty
     *  @param  s The string to check
     *  @param  errorMessage The error message to include in an exception
     *  @throws AirportException if string is empty
     */
    private void checkIfEmpty(String s, String errorMsg)
    {
        if (s.equals(""))
        {
            throw new AirportException (errorMsg);
        }
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

As you can see, although this class has more code than those we have met before, it follows a familiar pattern. Most of the code will therefore be familiar to you and the `Javadoc` comments and additional supplementary comments should be sufficient to follow what we have done here. We will just draw your attention to one or two JavaFX features that we have decided to incorporate into our implementation that will be new to you and that we mentioned in the introduction.

First, we have added **tool tips** to our buttons. A tool tip is an informative description of the purpose of a GUI component. This informative description is revealed when the user places the cursor over the component. Figure 21.7 shows the tool tip that is revealed when the cursor is placed over the "Land" button.

Adding a tool tip to a JavaFX component is easy; just use the `setToolTipText` method and provide a tool tip message as a parameter to the `ToolTip` class:

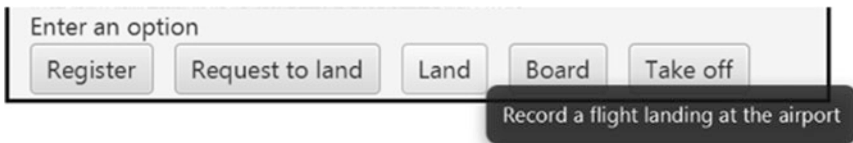


Fig. 21.7 A tool tip is revealed when the mouse is placed over the "Land" button

```
// create a Land button
Button button3 = new Button("Land");
// add a tool tip to the Land button using the setToolTip method
button3.setTooltip(new Tooltip("Record a flight landing at the airport"));
```

We also mentioned that we have made our frame an undecorated frame. An undecorated frame has no icons to minimise or maximise (or close) the frame. This ensures the frame remains the same size and this cannot be altered. To do this we use the `initStyle` method of our `stage` object and pass an enumerated type constant `StageStyle.UNDECORATED` as a parameter:

```
stage.initStyle(StageStyle.UNDECORATED);
```

Finally we draw your attention to a few things related to the functionality of this class.

When the application opens the user is given the option to load data from a file via an `Alert` dialog (see Fig. 21.8).

If the user chooses to load data from a file, the airport interface (as given in Fig. 21.5) is activated once the data is loaded. If the user chooses not to load data from a file, the system needs to be initialised by asking the user for the number of runways associated with this airport. A `TextInputDialog` is used for this purpose (see Fig. 21.9).

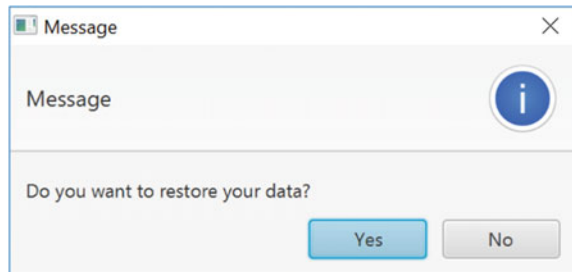


Fig. 21.8 An `Alert` dialog to allow users to load data from a file

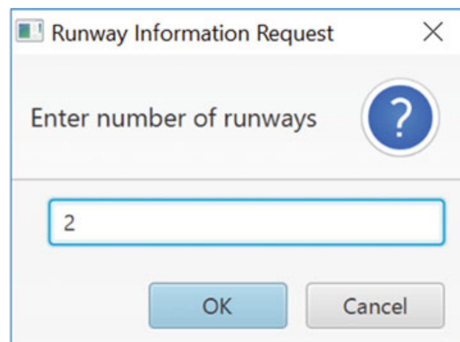


Fig. 21.9 A `TextInputDialog` to allow users to specify the number of runways

Again, the airport interface (as given in Fig. 21.5) is activated once the user has entered the number of runways. The user can then use the buttons on the flight control tab to register, request to land, land, board and take off flights at the airport —with arrivals and departures information being always available in the arrivals and departures tabs.

We have created two **`private`** methods, `listArrivals` and `listDepartures`. These methods update the information in the arrivals and departures tabs respectively. Whenever the airport data is modified (i.e. when a button is pressed in the main control tab) these methods need to be called to update the arrivals and departures tabs so when these tabs are revealed they always show the current state of flights in the system.

For example, when a flight registers at the airport the arrivals tab only needs to be updated before we exit the method. Here is the outline of the code for processing the register button response:

```
/**
 *  Private method to register new flight with the airport
 */
private void register()
{
        // code to register flight at airport here

        listArrivals();  // update arrivals tab
}
```

However, when a flight is ready for boarding, it needs to be removed from the arrivals tab and added to the departures tab, so both `listArrivals` and `listDepartures` need to be called.

```
/**
 *  Private method to register a flight boarding passengers at the airport
 */
private void board()
{
        // code to record flight as ready for boarding

        listArrivals();  // update arrivals tab
        listDepartures();  // update departures tab
}
```

When a flight takes off from the airport, it only needs to be removed from the departures tab:

```
/**
 *  Private method to record a flight leaving the airport
 */
private void takeOff()
{
        // code to record flight leaving the airport

        listDepartures();  // update departures tab only
}
```

Finally, the user can use the file menu to back-up the data to a file and continue with the application, to back-up data to a file and exit the application or simply exit the application without backing up the data (see Fig. 21.10).

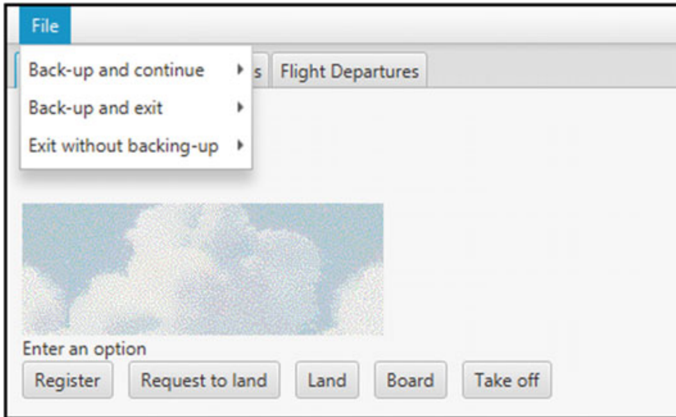


Fig. 21.10 Menu options to allow users to back-up data and exit the application as required

21.11 Self-test Questions

1. In Sect. 21.6 we developed scenarios for the use case "*Register flight arrival*". Develop scenarios for all the other use cases in Table 21.1.
2. What is the difference between *containment* and *composition* in UML?
3. Consider an enumerated type, `Light`. This type can have one of three values: RED, AMBER and GREEN. It will be used to display a message to students, indicating whether or not a lecturer is available to be seen.

   (a) Specify this type in UML.
   (b) Implement this type in Java.
   (c) Declare a `Light` variable, `doorLight`;
   (d) Write a **`switch`** statement that checks `doorLight` and displays "I am away" when `doorLight` is RED, "I am busy" when `doorLight` is AMBER and "I am free" when `doorLight` is GREEN.

4. Identify the benefits offered by the `TabPane` component.
5. How can the tool tip "*This button stops the game*" be added to a `Button` called `stop`?
6. Develop test plans for the `Runway`, `Plane`, `Airport` and `AirportFrame` classes.

21.12 Programming Exercises

Copy, from the accompanying web-site, the classes that make up the airport application and then tackle the following exercises:

1. Develop `toString` methods for the `Runway`, `Plane` and `Airport` classes and then develop testers for these classes.
2. Run and test the classes in the airport application by making use of your testing programs of programming exercise 1 above and following your test plans you devised in self-test question 6.
3. Make any further enhancements that you wish to the airport application. For example, you may wish to consider

   (a) adding a fourth "Help" tab that displays text describing how to use the airport application;
   (b) rather than use an enhanced **`for`** loop in the `getArrivals` and `getDepartures` methods of the `Airport` class make use of **`forEach`** loops;
   (c) instead of a `TextInputDialog` to enter the runway number in the `land` method of the `AirportFrame` class, use a `ChoiceDialog` with a list of runways pre-populated in a drop-down box;
   (d) identify methods, such as `nextFreeRunway` and `nextToLand` in the `Airport` class, that could return **`null`** values and then modify the code in the airport application to make use of the `Optional` type to avoid returning these **`null`** values;
   (e) design your own skin for the `AirportFrame` by creating a cascading style sheet.

4. Create an executable JAR file to run the airport application.

Check for updates

The Stream API

22

Outcomes:

By the end of this chapter you should be able to:

- *explain the term* ***stream*** *in the context of the java stream API*;
- *identify the advantages of stream processing over iteration*;
- *describe the three stages involved in processing a stream*;
- *explain what is meant by* ***lazy evaluation***;
- *create streams from scratch, from collections and from files*;
- *use a variety of intermediate methods to process streams*;
- *use appropriate methods to terminate streams*;
- *explain the difference between* ***stateless*** *and* ***stateful*** *operations*;
- *explain how streams can be created in* ***parallel mode***;
- *identify the possible pitfalls with parallel stream processing and explain how to avoid them.*

22.1 Introduction

In Chap. 15 you studied collections, and learnt how to process collections by iterating through the list in order to sort, select and retrieve items. Now, there is nothing wrong with that way of doing things. However, when it comes to processing large volumes of data, iterating through all the items can be rather a slow process, and is not necessarily the most efficient way of doing things.

For this reason, Java 8 came packaged with a new API, the **stream API**, to be found in `java.util`. This API provides classes—in particular the `Stream` class—that enable us to process collections by making use of the multi-tasking abilities of modern machines. Methods of the `Stream` class allow this processing to take place behind the scenes, without having to trouble the programmer.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_22

In Java, a stream (not to be confused with the input-output streams discussed in Chap. 18) is a collection of items that is created in memory, and which ceases to exist once the processing is completed.

There are three stages to processing a stream—these are described below:

Stage 1
We create a stream either from scratch, or from an existing collection.

Stage 2
We specify *intermediate operations*. These operations transform the initial stream into other streams. This is done in one or more steps.

Stage 3
We apply a *terminal operation*, which produces a result. The terminal operation forces the execution of the preceding operations—once the terminal operation has been applied the stream can no longer be used. Attempting to do so will cause an exception at runtime.

One important aspect of stream processing that we should point out is that the intermediate operations are not executed until the terminal operation is invoked. Each intermediate operation creates a new stream and this stream is stored along with the function provided. These streams accumulate as the pipeline is traversed, and once the terminal operation is called each function is performed one by one. This is referred to as **lazy evaluation**.

Before taking a look at some concrete examples, there is one other important aspect of streams that we should mention at this juncture. Programming streams is all about *what* is to be done rather than *how* it is to be done. Do you remember in Chap. 19 that we showed you how to communicate with a remote database via a java program? We showed you there how to embed SQL (Structured Query Language) code into our java applications. In SQL we have statements that look like this:

```
select serialNumber from Products where make = 'Acme';
```

This would display all the serial numbers of products in a table which are manufactured by "Acme".

Now, even if you know nothing about SQL, it should be apparent that this instruction simply states what we want to achieve, and says nothing about how it is to be achieved—a similar statement in Java, working for example on an `ArrayList`, would require us to write a loop for iterating through all the elements. SQL is a fourth generation language—also called a **declarative** language—which means that we are able simply to state the result we require, rather than tell the computer how to do it.

Streams in Java do the same thing. They remove from the programmer the burden of writing the code for iteration and selection and concentrate instead on what the code is required to achieve.

So now let's look at some examples.

22.2 Streams Versus Iterations: Example Program

To understand how all this works we will study a simple example of two programs which analyse a collection of objects—the first one uses the methods of the collection classes that we are familiar with, and the second one uses streams. Our collection will consist of objects from the `Product` class that we developed in Chap. 19. This class is reproduced below—you will notice that in this version we have overridden the `toString` method in order to make it easy to display the full details of each product.

Product

```
public class Product
{
    private String stockNumber;
    private String manufacturer;
    private String item;
    private double unitPrice;

    public Product(String stockNumberIn, String manufacturerIn, String itemIn, double unitPriceIn)
    {
        stockNumber = stockNumberIn;
        manufacturer = manufacturerIn;
        item = itemIn;
        unitPrice = unitPriceIn;
    }

    public Product()
    {

    }

    public String getStockNumber()
    {
        return stockNumber;
    }

     public void setStockNumber(String stockNumberIn)
    {
        stockNumber = stockNumberIn;
    }

    public String getManufacturer()
    {
        return manufacturer;
    }

    public void setManufacturer(String manufacturerIn)
    {
        manufacturer = manufacturerIn;
    }


    public String getItem()
    {
        return item;
    }

    public void setItem(String itemIn)
    {
        item = itemIn;
    }

    public double getUnitPrice()
    {
        return unitPrice;
    }

    public void setUnitPrice(double unitPriceIn)
    {
        unitPrice = unitPriceIn;
    }

    @Override
    public String toString()
    {
        return stockNumber + " " + manufacturer + " " + item + " " + unitPrice;
    }
}
```

So our first program will add a few products to a list, then—in the way we are used to—iterate through the list to display all the products; it will then repeat this process, but this time filter the products to exclude the more expensive items (those costing 170 or more). Finally it will display the number of items that remain.

QueryWithoutUsingStreams

```
import java.util.ArrayList;
import java.util.List;


public class QueryWithoutUsingStreams
{
    public static void main(String[] args)
    {

        List<Product> productList = new ArrayList<>(); // create a list of products
        int count = 0;

        // add four products to the list
        productList.add(new Product("1076543", "Acme", "Vacuum Cleaner", 180.11));
        productList.add(new Product("3756354", "Nadir", "Washing Machine", 178.97));
        productList.add(new Product("1234567", "Zenith", "Fridge", 151.98));
        productList.add(new Product("7876161", "Zenith", "Tumble Drier", 159.99));

        System.out.println("ALL ITEMS");

        // display all items
        for(Product pr : productList)
        {
            System.out.println(pr);
        }

        System.out.println();

        System.out.println("ITEMS COSTING LESS THAN 170");

        // display items costing less than 170
        for(Product pr : productList)
        {
            if(pr.getUnitPrice() < 170)
            {
                System.out.println(pr);
                count++;
            }
        }

        System.out.println();
        System.out.println("There are " + count + " items costing less then 170");

    }
}
```

The output from this program is, as expected:

```
ALL ITEMS
Acme Vacuum Cleaner 180.11
Nadir Washing Machine 178.97
Zenith Fridge 151.98
Zenith Tumble Drier 159.99

ITEMS COSTING LESS THAN 170
Zenith Fridge 151.98
Zenith Tumble Drier 159.99

There are 2 items costing less then 170
```

Now let's see how we do exactly the same thing using streams:

QueryUsingStreams

```
import java.util.ArrayList;
import java.util.List;


public class QueryUsingStreams
{
    public static void main(String[] args)
    {
        List<Product> productList = new ArrayList<>(); // create a list of products

        // add four products to the list
        productList.add(new Product("1076543", "Acme", "Vacuum Cleaner", 180.11));
        productList.add(new Product("3756354", "Nadir", "Washing Machine", 178.97));
        productList.add(new Product("1234567", "Zenith", "Fridge", 151.98));
        productList.add(new Product("7876161", "Zenith", "Tumble Drier", 159.99));

        // display all items
        System.out.println("ALL ITEMS");
        productList.stream().forEach(pr -> System.out.println(pr));

        System.out.println();

        // filter the list and display items costing less than 170
        System.out.println("ITEMS UNDER 170");
        productList.stream().filter(pr -> pr.getUnitPrice() < 170).forEach(pr -> System.out.println(pr));

        // count items costing less than 170
        long count = productList.stream().filter(pr -> pr.getUnitPrice() < 170).count();

        System.out.println();
        System.out.println("There are " + count + " items costing less then 170");
    }
}
```

Let's start by looking at the line of code that displays all the items:

```
productList.stream().forEach(pr -> System.out.println(pr));
```

This starts to give you an idea of how we pipeline the stream operations. Here we have begun the process (stage 1) by creating a steam with the `stream` method which is provided as part of the `Collection` interface implemented by `ArrayList` (as explained in Chap. 15). In this case, all we want to do is display the entire list, so there are no intermediate operations (stage 2) and we proceed directly to termination (stage 3). The termination operation we use here is `forEach`, which performs the required action for each item in the list. You were introduced to the `forEach` method in Chap. 15 when it was used directly with the Java collection types, but it is also available to Java streams—more detail about `forEach` in the sections that follow.

We should point out here that the above line of code could have utilized the double colon notation that we introduced in Chap. 13. So we could have written:

```
productList.stream().forEach(System.out::println);
```

In the programs that follow we will use this notation where possible.

Now let's look at the way we have filtered our results to display only items costing under 170:

```
productList.stream().filter(pr -> pr.getUnitPrice() < 170).forEach(pr -> System.out.println(pr));
```

Here you see the use of an intermediate operation which uses the `filter` method. You can see how we send in the criteria on which to filter the items as a lambda expression—again, more about this in the following sections. You should note that any intermediate method, such as `filter`, returns another stream; this stream is again terminated with the `forEach` method.

Finally we have used the following line of code to count the filtered stream:

```
long count = productList.stream().filter(pr -> pr.getUnitPrice() < 170).count();
```

The `count` method, which is again a termination method, returns a **`long`**, which we use to display the number of items in the resulting stream, which in this case will consist of items costing under 170.

22.3 Creating Streams

In the previous section we created our stream from an existing `ArrayList`, using the `stream` method. It is also possible to create a stream from scratch as in the following example:

StreamFromValues

```
import java.util.stream.Stream;

public class StreamFromValues
{
    public static void main(String[] args)
    {
        // create stream from values
        Stream<String> colours = Stream.of("Purple", "Blue", "Red", "Yellow", "Green");

        // filter the list and display strings of length 5 or more
        colours.filter(c -> c.length() >= 5).forEach(System.out::println);
    }
}
```

Here we have created a named `Stream` object (holding `Strings`) and used the static `Stream` method called `of` to create the stream from a list of values.

We have filtered the stream to contain only strings consisting of five characters or more, se we get the following output:

Purple
Yellow
Green

It is also possible to create a stream from an array, by using the static `stream` method of the `Arrays` class that resides in `java.util`:

StreamFromArray

```
import java.util.Arrays;
import java.util.stream.Stream;

public class StreamFromArray
{
    public static void main(String[] args)
    {
        // create an array of Products
        Product[] productList = {
                                    new Product("1076543", "Acme", "Vacuum Cleaner", 180.11),
                                    new Product("3756354", "Nadir", "Washing Machine", 178.97),
                                    new Product("1234567", "Zenith", "Fridge", 151.98)
                                };

        // create a stream from the array
        Stream<Product> products = Arrays.stream(productList);

        products.forEach(System.out::println);
    }
}
```

If you want to create an empty stream (of `Strings`, for example), you can do so as follows:

```
Stream<String> s = Stream.empty();
```

A stream can also be created from a file, with the help of the `Files` class which resides in `java.nio.files`. We will demonstrate this with the file *Poem.txt*, which we used in Chap. 17, and which contains the following text:

The moving finger writes and having writ
Moves on; nor all thy piety nor wit
Shall lure it back to cancel half a line,
Nor all thy tears wash out a word of it.

In the program below, the static `lines` method of `Files` is used to convert the text to a stream of `Strings`, each one being a line of text in the file.

StreamFromFile

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

class StreamFromFile
{
    public static void main(String[] args)
    {
        Stream<String> fileStream = Stream.empty(); // create empty stream

        try
        {
            fileStream = Files.lines(Paths.get("Poem.txt")); // file in current directory
        }

        catch (IOException ex)
        {
        }

        fileStream.forEach(System.out::println);
    }
}
```

Notice that `lines` requires an object of type `Path`. To create this object, we have used the `get` method of `Paths` (also found in `java.nio.file`)—this method creates a `Path` object by joining together the strings that are sent in as parameters.[1] We have sent in only the file name, so the program will look in the current directory.

The output is as follows:

```
The moving finger writes and having writ
Moves on; nor all thy piety nor wit
Shall lure it back to cancel half a line,
Nor all thy tears wash out a word of it.
```

22.4 Intermediate Operations

Intermediate operations transform one stream to another stream. So far the only intermediate method we have encountered is `filter`, which selects which items will be included in the new stream, based on the criteria sent into the method. Intermediate methods make use of a couple of the “out of the box” interfaces that we encountered in Chap. 13. We remind you of these below in Table 22.1

Table 22.1 Reminder of the `Predicate` and `Function` interfaces

Functional Interface	Abstract method name	Parameter types	Return type
`Predicate <T>`	`test`	`T`	`boolean`
`Function <T, R>`	`apply`	`T`	`R`

[1]More detail about the `Paths` class and `Path` interface can be found on the Oracle™ website.

The filter method requires a Predicate—its abstract method, test, will expect to receive an object of the type of item held (Product or String in our previous examples) and return a **boolean**, so we send in the appropriate lambda expression as we have done in the examples you have seen:

```
filter(pr -> pr.getUnitPrice() < 170
```

and

```
filter(c -> c.length() >= 5)
```

Next we introduce three other intermediate methods: distinct, sorted, and map. The program below creates a stream, then pipelines these three methods as well as the filter method before terminating the stream with the forEach method.

IntermediateExamples

```
import java.util.stream.Stream;

public class IntermediateExamples
{
    public static void main(String[] args)
    {
       Stream<String> colours
          = Stream.of("Purple", "Blue", "Red", "Yellow", "Green", "Yellow", "Purple", "Orange", "Black");

       colours.filter(c -> c.length() > 4).distinct().sorted().map(c -> c.substring(0, 2))
            .forEach(System.out::println);

    }
}
```

You can see that we have, in this case, created a stream containing duplicates. The distinct method, which does not require any parameters, simply transforms the stream into a new stream with the duplicates removed. The sorted method, also requiring no parameters, as its name suggests, produces a stream sorted on "natural" order (for example numerical or alphabetical order). The map method expects an item of type Function, and transforms each element to another element according to the lambda expression sent into its apply method. In our example, each string is converted to a string containing only the first two characters of the original.

The output from this program is:

```
Bl
Gr
Or
Pu
Ye
```

Next we will look at the flatMap method. This method again accepts a function, but will produce a stream with the original contents broken down according to the criteria we specify in the lambda expression. We will demonstrate this by making use of the stream that we created from a file in our

StreamFromFile program above. The resulting stream contained four lines of text. In the program that follows we will break this down to a stream consisting of the individual words (that is, the elements that are separated by a space).

FlatMapExample

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.stream.Stream;

public class FlatMapExample
{
    public static void main(String[] args)
    {
        Stream<String> fileStream = Stream.empty(); // create empty stream

        try
        {
            fileStream = Files.lines(Paths.get("Poem.txt")); // file in current directory
        }

        catch (IOException ex)
        {
        }

        // create a stream of individual words
        fileStream.flatMap(s -> Arrays.stream(s.split(" "))).forEach(System.out::println);
    }
}
```

Like map, flatMap receives a function, but in this case the type received by the function's apply method is restricted to a Stream type by the use of an upper bound (as explained in Chap. 13). In our example the split method of String is employed to transform the original stream to another stream broken into the individual words contained in each line. The output from this program is:

The
moving
finger
writes
and
having
writ
Moves
on;
nor
all
thy
piety
nor
wit
Shall
lure
it
back
to

cancel
half
a
line,
Nor
all
thy
tears
wash
out
a
word
of
it.

Finally we can look at three more operations, the `limit` operation, the `skip` operation and the `peek` operation. The `limit` method takes an integer—*n* for example—and transforms the original stream into a stream containing only the first *n* items. So, for example, in our `IntermediateExamples` program above we could add a `limit` operation into the pipeline as follows:

```
colours.filter(c -> c.length() > 4).distinct().sorted()
      .map(c -> c.substring(0, 2)).limit(2).forEach(System.out::println);
```

In this case the final output would be:

Bl
Gr

The `skip` operation does the opposite of `limit`, and discards the first *n* elements. So replacing `limit(2)` with `skip(2)` in the pipeline above would gives us:

Or
Pu
Ye

The `peek` method is very useful for debugging—it enables us to look into the stream at a given point, but unlike `forEach` it does not terminate the stream.

For example, in the following line of the `QueryUsingStreams` example above:

```
long count = productList.stream().filter(pr - > pr.getUnit-
Price() < 170).count();
```

we could have added a `peek` method as follows:

```
long count = productList.stream().peek(s -> System.out.println(s)).
                                       filter(pr -> pr.getUnitPrice() < 170).count();
```

This would print all the items in the stream so far.

Table 22.2 Reminder of the `consumer` interface

Functional Interface	Abstract method name	Parameter types	Return type
`Consumer<T>`	`accept`	`T`	`void`

22.5 Operations for Terminating Streams

So far we have encountered two terminal operations, `count` and `forEach`.

As we have seen, the `count` method does not require any arguments and simply returns a **`long`**, representing the number of elements in the stream.

The `forEach` method receives a `Consumer` type as its argument (reminder in Table 22.2). The `accept` method of `Consumer` receives an item of the type held by the stream—the lambda expression then defines how that item will be processed. All that the lambda expressions in the above examples have done is to display each item on the screen.

22.5.1 More Examples

Now we can explore some other terminal methods. The following program demonstrates a few of these:

TerminalExamples

```
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;


public class TerminalExamples
{
    public static void main(String[] args)
    {
        // find the maximum of a stream of integers
        Optional<Integer>  maximumInt =  Stream.of(1, 2, 3, 11, 7, 8, 10).max(Comparator.naturalOrder());
        System.out.println("The maximum integer is " + maximumInt.get());

        // find the "minimum" of a stream of strings
        Optional<String>  minimumString =
                                Stream.of("banana", "apple", "apple", "orange").min(Comparator.naturalOrder());
        System.out.println("The first string alphabetically is " + minimumString.get());

        // find cheapest product from a stream of products
        Optional<Product> cheapestProduct =
                                Stream.of(
                                               new Product("1076543", "Acme", "Vacuum Cleaner", 180.11),
                                               new Product("3756354", "Nadir", "Washing Machine", 178.97),
                                               new Product("1234567", "Zenith", "Fridge", 151.98),
                                               new Product("7643210", "Wizz", "Dish Washer", 219.99)
                                          ).min(Comparator.comparingDouble(Product::getUnitPrice));
        System.out.println("The cheapest product is " + cheapestProduct.get());

        // find the first in the list in a stream of doubles
        Optional<Double>  firstDouble =  Stream.of(1.6, 2.7, 6.8).findFirst();
        System.out.println("The first double in the list is " + firstDouble.get());

        // find the sum of a stream of integers
        Optional<Integer>  sumOfIntegers =  Stream.of(1, 2, 3, 4, 5).reduce((x, y) -> x + y);
        System.out.println("The sum of the integers is " + sumOfIntegers.get());

        // find if a specific item is in the stream
        boolean  appleExists =  Stream.of("banana", "pear", "apple", "orange").anyMatch(s -> s.equals("apple"));
        if(appleExists)
        {
                System.out.println("apple is in the list");
        }

    }
}
```

The first three examples in this program use the max and min functions, which do as their names suggest. They both return items of type Optional, and the native type is extracted with the get method. They both require a Comparator object as an argument (Comparators were introduced in Chap. 15). The first two streams in our example contain Integers and Strings respectively, and the method can therefore receive a Comparator.naturalOrder() argument, as in the first example:

```
Optional<Integer>  maximumInt =  Stream.of(1, 2, 3, 11, 7, 8, 10).max(Comparator.naturalOrder());
```

In the third example the stream holds items of our own Product class, and we want to know the cheapest. We therefore have to consider the unitPrice attribute, which is of type **double**, so we use the comparingDouble method of Comparator with the correct lambda expression:

```
Optional<Product> cheapestProduct = Stream.of(
                                               new Product("1076543", "Acme", "Vacuum Cleaner", 180.11),
                                               new Product("3756354", "Nadir", "Washing Machine", 178.97),
                                               new Product("1234567", "Zenith", "Fridge", 151.98),
                                               new Product("7643210", "Wizz", "Dish Washer", 219.99)
                                          ).min(Comparator.comparingDouble(Product::getUnitPrice));
```

The next example in our program demonstrates the findFirst method, which, as you might expect, finds the first item in the stream:

```
Optional<Double>  firstDouble =  Stream.of(1.6, 2.7, 6.8).findFirst();
```

In addition to findFirst there is also a findAny method, which returns a random item in the stream.

The next example makes use of the reduce method.

```
Optional<Integer>  sumOfIntegers =  Stream.of(1, 2, 3, 4, 5).reduce((x, y) -> x + y);
```

The reduce method performs operations on the elements according to the pattern defined in the lambda expression. In this example, the method returns the sum of the elements. It returns an item of type Optional, because there is no valid result if the stream is empty.

If you want to avoid using Optional, there is another version of reduce that you can use. In our case it would look like this:

```
int sumOfIntegers = Stream.of(1, 2, 3, 4, 5).reduce(0, (x, y) -> x + y);
```

The first parameter is referred to as an identity. It is defined like this:

identity + *x* = *x*

In the case of integer addition, the appropriate identity is zero. This is added to the total—the result is unaffected by adding zero, but if the stream is empty, there is nonetheless a valid result.

If, for example, we were concatenating a stream of strings, we would use "" as our identity.

The final method that we demonstrate in our program is `anyMatch`:

```
boolean  appleExists = Stream.of("banana", "pear", "apple", "orange").anyMatch(s -> s.equals("apple"));
```

It determines if any item in the stream matches the value defined by the lambda expression—in this case it checks to see it the word "apple" is in the stream. `anyMatch` returns a **`boolean`** value.

There are two operations similar to `anyMatch`. `allMatch` determines if all the elements are of a particular value, while `noneMatch` does the opposite to `anyMatch`.

The output from our program is:

```
The maximum integer is 11
The first string alphabetically is apple
The cheapest product is 1234567 Zenith Fridge 151.98
The first double in the list is 1.6
The sum of the integers is 15
apple is in the list
```

22.5.2 Collecting Results

As you have already found out, once we terminate a stream, the stream is no longer available. But it is very likely that we might want to save the results of processing a stream for later use and for this purpose a special terminal method, `collect`, is available.

In the program that follows we have created a stream of country names from an `ArrayList`, then we have sorted the stream, and collected the results into a new `List`.

CollectionExample

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectionExample
{
    public static void main(String[] args)
    {
        // create an ArrayList of strings
        List<String> countryList = new ArrayList<>();

        countryList.add("Germany");
        countryList.add("France");
        countryList.add("Nigeria");
        countryList.add("Canada");
        countryList.add("India");

        // create a stream from the ArrayList
        Stream<String> countryStream = countryList.stream();

        // sort the stream data and save the result in a new ArrayList
        List<String> sortedList =  countryStream.sorted().collect(Collectors.toList());

        // display the sorted list
        for(String item : sortedList)
        {
            System.out.println(item);
        }
    }
}
```

The line of code that we are interested in is this one:

```
List<String> sortedList = countryStream.sorted().collect(Collectors.toList());
```

We are using the `collect` method of `Stream` to collect our items and place them into a list. The version of `collect` that we are using here requires an object of type `Collector`. Java provides a `Collectors` class that has static methods that generate `Collector` objects. Here we are using the `toList` object to create a list.

If we had wanted a set then we would have used `toSet` as follows:

```
Set<String> sortedList = countryStream.sorted().collect(Collectors.toSet());
```

It is also possible to collect the data into a map, using the `toMap` method. This method requires two functions (sent in as lambda expressions) which define the key and the value of the map. In our example, if we required each element of the map to comprise the initial letter and name of each country, we would do the following:

```
Map<Character, String> map =  countryStream.sorted().collect(Collectors.toMap(s -> s.charAt(0), s -> s));
```

22.6 Concatenating Streams

Combining two streams of the same type is easy. The `Stream` class has a **`static`** method called `concat`, so to join, for example, two streams of `Strings`—`stream1` and `stream2`—we simply do the following:

```
Stream<String> combined = Stream.concat(stream1, stream2);
```

22.7 Infinite Streams

The idea of creating an infinite object might seem rather an alien concept to a programmer—but it is quite possible to create an infinite stream of items and truncate the stream to, say, the first one hundred elements. This is possible because, as we explained in Sect. 22.1, the stream operations don't come into effect until the terminal operation is encountered, so the stream is created only with the finite number of elements that are required.

Some examples will make this clear. The program below shows three examples which we will discuss once you have had a look at it.

InfiniteStreams

```
import java.util.stream.Stream;

public class InfiniteStreams
{
    public static void main(String[] args)
    {
        // an infinite stream of strings
        Stream<String> echo = Stream.generate(() -> "Hello world");
        echo.limit(10).forEach(s -> System.out.println(s));

        // an infinite stream of random numbers
        Stream<Double> randomNumbers = Stream.generate(() -> Math.random());
        randomNumbers.limit(10).forEach(System.out::println);

        // an infinite sequence of integers
        Stream<Integer> sequence = Stream.iterate(1, n -> n+2);
        sequence.limit(5).forEach(System.out::println);
    }
}
```

There are two static methods of `Stream` which we can use to create infinite streams: `generate` and `iterate`. They make use, respectively, of the `Supplier` and `UnaryOperator` generic interfaces that you learnt about in Chap. 13, and which are reproduced for you here in Table 22.3 as a reminder.

Table 22.3 Reminder of the `Supplier` and `unaryoperator` interfaces

Functional Interface	Abstract method name	Parameter types	Return type
`Supplier<T>`	`get`	`none`	`T`
`UnaryOperator<T>`	`apply`	`T`	`T`

The first two examples in the program above make use of the `generate` method. As it requires a `Supplier` as argument, the lambda expression requires no input, and simply outputs what is to be repeated in the stream. The first example creates a stream of `Strings` ("Hello world"), which is then limited to the first 10 items; so the final result is that "Hello world" is displayed ten times:

```
Stream<String> echo = Stream.generate(() -> "Hello world");
echo.limit(10).forEach(System.out::println);
```

The second example is similar, but makes use of the `random` method of the `Math` class so that 10 random numbers are displayed.

```
Stream<Double> randomNumbers = Stream.generate(() -> Math.random());
randomNumbers.limit(10).forEach(System.out::println);
```

The final example uses the `iterate` method:

```
Stream<Integer> sequence = Stream.iterate(1, n -> n+2);
sequence.limit(5).forEach(System.out::println);
```

The `iterate` method receives two arguments. The first, which is of the same type as the stream items, is referred to as a **seed**; the second is a `UnaryOperator`. The lambda expression for the `UnaryOperator` defines the operation that is to take place on the seed and then on each subsequent item. In our example we start off with 1 (the seed), then continuously add 2. In this case we have limited the final stream to the first 5 items, with the result that the numbers 1, 3, 5, 7, 9 are displayed.

22.8 Stateless and Stateful Operations

Certain stream operations, when executed, examine individual items in the stream and perform an action on the item without having to worry about any of the other members of the stream. Take for example the `filter` operation. When making the decision about whether to include a particular item in the new stream, it is not necessary to think about the other items—for example, if the `filter` method has to include only strings that have more than five characters, the string "yellow" is always going to be included, irrespective of what else is in the stream.

Compare this to a method like `sorted`. In this case the position of an item depends upon the other items in the stream, so that the method has got to in some way remember the items that have already been processed.

Operations like `filter` that do not have to remember what has gone before are called **stateless** operations; in contrast, methods like sorted are referred to as **stateful**. The following intermediate operations are stateful:

```
distinct
sorted
limit
skip
```

The others are all stateless.

22.9 Parallelism

As we explained in the introduction, stream processing makes use of the multi-tasking and multi-processing capabilities of the system as a whole, and that this goes on behind the scenes. It is important to emphasise here that multi-tasking applies to the internal execution of the individual operations; the operations in the pipeline are, under normal circumstances, executed in sequence. This is important, because without careful thought, allowing operations to be carried out in random order could lead to very different results from the ones intended. Consider, for example, a stream of strings consisting of the words *foot*, *feet*, *feet*, *folder*, *foot*, *feeling*. Now imagine carrying out two operations on these—a `distinct` operation, and a `map` operation that reduces each item to its first two characters. It should be easy to see that carrying out these operations in different orders will produce two different results.

It is nonetheless possible to have streams in which the intermediate operations are parallelized. This is done either by creating a parallel stream with the `parallelStream` operation of `Collection`:

```
Stream<Product>  para = productList.parallelStream();
```

or by converting an existing sequential stream to parallel mode with the `parallel` method of `Stream`:

```
Stream<Product>  para = sequentialStream.parallel();
```

It should, however, be evident from the above discussion that extreme care should be exercised when using parallel streams. Firstly all the operations should be *stateless*. Secondly they must be able to be executed in arbitrary order.

22.10 Self-test Questions

1. What are the advantages of using streams to process collections, compared to iteration?
2. Describe the three stages involved in processing a stream.
3. Explain what is meant by *lazy evaluation*.
4. What is the difference between stateless and stateful operations? Give examples of both.
5. Why is it necessary to exercise caution when it comes to processing streams in parallel mode? What steps should be taken to avoid problems?
6. Explain why the following lines in a program would create a problem at runtime:

```
Stream<String> colours = Stream.of("Purple", "Blue", "Red", "Yellow", "Green", "Yellow", "Purple", "Black");

colours.filter(c -> c.length() > 4).distinct().sorted().forEach(System.out::println);
colours.filter(c -> c.length() > 4).distinct().sorted().count();
```

22.11 Programming Exercises

1. Implement some of the programs from this chapter, and experiment with using different stream methods.
2. Write a short program the uses the `skip` method and the `limit` method to extract a substream from a stream, from a start position to an end position.
3. Use the `iterate` method of `Stream` to display the first 5 square numbers.
4. In Sect. 8.8.1 of Chap. 8 we developed a `Bank` class. Take a look at the following methods and see if you can rewrite them so that they use stream processing instead of iteration:

(a) The `getItem` method

You will need to filter the stream so that it contains one item, and collect the new stream into a list. As a `BankAccount` has to be returned, you will have to return the first (and only) item in the list. Also, bear in mind that a null value needs to be returned if the requested account does not exist.

(b) The `removeAccount` method

In this case, you need to filter out the item in question. As you have to save the resulting stream to the original list, which is an `ArrayList`, and the `toList` method simply returns a `List`, you will need to type cast. Again you will have to figure out a way of reporting on whether or not the particular account number exists.

5. Look at the case study from the previous chapter. Try rewriting some or all of the methods in that case study that made use of iteration so that they make use of stream processing instead.

Working with Sockets

23

Outcomes:

By the end of this chapter you should be able to:

- *explain the terms* ***client***, ***server***, ***host*** *and* ***port***;
- *describe the* ***client–server*** *model*;
- *explain the function of a* ***socket***;
- *distinguish between the Java* `Socket` *and* `ServerSocket` *classes and explain their function*;
- *write a simple client–server application using sockets*;
- *write a server application that supports multiple clients*;
- *write multi-threaded client–server applications*;
- *create client and server applications that utilise a JavaFX interface.*

23.1 Introduction

In this chapter we are going to explore the way in which Java can be used to write programs that communicate over a network. We should say from the outset that here we are dealing only with communication over a local area network (LAN). Communication over wide networks, and in particular the Internet, involve issues that are beyond the scope of this text. In particular, communication over the Internet is now fraught with security issues, and we are not able to address such concerns here. In previous editions of this book, we covered applets, which are java programs that run in a browser. We are no longer going to deal with applets as browsers have by and large stopped supporting them because of the enormous security questions that they can pose.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_23

Network programs rely very much on the concept of a **client** and a **server**. A server program provides some sort of service for other programs—clients—normally located on a different machine. The service it provides could be one of many things—it could send some files to the client; it could send web pages to the client; it could read some data from the local machine and send that across, maybe having done some processing first; it could perform a complex calculation; it could print some material on a local printer. The possibilities are endless.

It should be noted that the distinction between a client and server can become blurred: a program acting as a client in one situation could also act as a server in another, and vice versa. It is also important to note that it is often the case that a machine, rather than a program, is referred to as a server. This usually happens when a machine is dedicated to running a particular server program—typically a file server—and does very little else. Strictly speaking we should refer to the machine on which a server runs as the **host**.

In general, communication between a client and a server could be over a local area network, a wide area network, or over the Internet. Server programs that offer a service via the Internet have to obey a particular set of rules or protocols to ensure that the client and server are "speaking the same language". Common examples are File Transfer Protocol (FTP) for servers that send files, and Hypertext Transfer Protocol (HTTP) for services that send web pages to a client.

However, as we have stated above, here we will be dealing here only with local area networks which communicate by implementing **sockets**—special programs that allow data to pass between two applications running on different computers.

23.2 Sockets

In Chap. 18 you were introduced to the idea of a *stream*—a channel of communication between the computer's main memory and some external device such as a disk (not to be confused with streams for processing collections, that we covered in Chap. 22). In that chapter you were shown how Java provides high-level classes that hide the programmer from the low-level details of how data is stored on a disk or other device. Just as the external storage of data is a complicated business, so too is the transmission of data across a network.

A **socket** is a software mechanism that is able to hide the programmer from the detail of how data is actually transmitted, in a not dissimilar way to that in which the high-level file handling classes protect the programmer from the details of external storage. Sockets were originally developed for the Unix™ operating system and they enabled the programmer to treat a network connection as just another stream to which data can be written, and from which it can be read. Sockets have since been developed for other operating systems such as Windows™, and fortunately for Java.

In order to understand sockets it is also necessary to understand the concept of a **port**. A machine on a network is referred to by its IP (Internet Protocol) address. However, any particular host can perform a number of different functions, and therefore needs to be able to distinguish between different types of request, such as email requests, file transfer requests, requests for web pages and so on. This is accomplished by assigning each type of request a special number known as a port. Many port numbers are now internationally recognized, and so all computers will agree on their meaning. For example, a request on port 80 will always be expected to be an HTTP request; port 21 is for FTP (File Transfer Protocol) requests. A client program can therefore assume that server programs will be using these ports for those particular services.

All sockets must be capable of doing the following:

- connect to a remote machine;
- send data;
- receive data;
- close a connection.

A socket which is to be used for a server must additionally be able to:

- bind to a port (that is to associate the server with a port number);
- listen for incoming data;
- accept connections from a remote server on the bound port.

The Java `Socket` class has methods that correspond to the first four of the above; the `ServerSocket` class provides methods for the last three.

23.3 A Simple Server Application

The server we are going to build is going to offer a very simple service to a client; it will wait to receive two integers, and then it will send back the sum of those two integers. Clearly this would not in reality be a very useful server—a real-world server would be offering a far more complex service—perhaps performing some very complicated processing, or retrieving data from a database running on the same machine, or maybe printing on a printer local to the server. However, our simple addition server demonstrates the principles of a client–server protocol very nicely.

A program such as this would typically be launched from a command line, or perhaps launched as a service on startup—many services run in the background, and it is often the case that the user has little awareness of their existence. In order to monitor the behaviour of our server we have organised it so that it reports its behaviour to a console.

The `AdditionServer` class is presented below—have a look at it and then we'll take you through it.

AdditionServer

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class AdditionServer
{
    public static void main(String[] args)
    {
        final int port = 8901;

        // declare a "general" socket and a server socket
        Socket connection;
        ServerSocket listenSocket;

        // declare low level and high level objects for input
        InputStream inStream;
        DataInputStream inDataStream;

        // declare low level and high level objects for output
        OutputStream outStream;
        DataOutputStream outDataStream;

        // declare other variables
        String client;
        int first, second, sum;
        boolean connected;
        while(true)
        {
            try
            {
                // create a server socket
                listenSocket = new ServerSocket(port);
                System.out.println("Listening on port " + port);

                // listen for a connection from the client
                connection = listenSocket.accept();
                connected = true;
                System.out.println("Connection established");

                // create an input stream from the client
                inStream = connection.getInputStream();
                inDataStream = new DataInputStream(inStream);

                // create an output stream to the client
                outStream = connection.getOutputStream ();
                outDataStream = new DataOutputStream(outStream);

                // wait for a string from the client
                client = inDataStream.readUTF();
                System.out.println("Address of client: " + client);
                while(connected)
                {
                    // read an integer from the client
                    first = inDataStream.readInt();
                    System.out.println("First number received: " + first);

                    // read an integer from the client
                    second = inDataStream.readInt();
                    System.out.println("Second number received: " + second);

                    sum = first + second;
                    System.out.println("Sum returned: " + sum);

                    // send the sum to the client
                    outDataStream.writeInt(sum);
                }
            }

            catch (IOException e)
            {
                connected = false;
            }
        }
    }
}
```

For convenience we have hard-coded the port number (8901) into the program and have declared a constant for this purpose; the client will need to be made aware of this port number.

Having declared this constant, we have gone on to declare a number of variables:

```
Socket connection;
ServerSocket listenSocket;

InputStream inStream;
DataInputStream inDataStream;

OutputStream outStream;
DataOutputStream outDataStream;

String client;
int first, second, sum;
boolean connected;
```

The first two variables are, respectively, a `Socket` and a `ServerSocket`. As this is a server application it requires both the general functionality of the `Socket` class and the specialist functionality of the `ServerSocket` class.

Next we declare the objects that we will need to establish an input stream with the client. We have come across the classes `InputStream` and `DataInputStream` before, in Chap. 18. The former allows communication at a low level in the form of bytes; the latter allows the high-level communication in the form of strings, integers, characters and so on with which we are familiar.

After this we declare objects of `OutputStream` and `DataOutputStream` that we will need to establish the output stream. Finally we make some other declarations that we will need later on.

Now we start an infinite loop. The idea is that the server will accept a connection request from a client, and when that client is finished making requests it will be ready to receive connections from other clients; this will continue until the server is terminated. If the server was called from the command line, then closing the console window will terminate it—if you are working in an IDE, you will have to terminate the process via the IDE interface when you no longer require it.

From now everything is placed in a **`try`** block because the constructor of the `ServerSocket` class, and its `accept` method both throw `IOExceptions`.

The first instruction in the **`try`** block looks like this:

```
listenSocket = new ServerSocket(port);
```

We are creating a new `ServerSocket` object and binding it to a particular port.

In order to get the server to listen for a client requesting a connection on that port, we call the `accept` method of the `ServerSocket` class; we also place a message in the console to tell us that the server is listening for a request:

```
System.out.println("Listening on port " + port);
connection = listenSocket.accept();
```

The `accept` method returns an object of the `Socket` class, which we assign to the `connection` variable that we declared earlier.

Once the connection is established we set the **boolean** variable, `connected`, to **true** and display a message:

```
connected = true;
System.out.println("Connection established");
```

The next thing we do is call the `getInputStream` method of the `Socket` object, `connection`. This returns an object of the `InputStream` class, thus providing a stream from client to server. We then wrap this low-level `InputStream` object with a high-level `DataInputStream object`, in the same way as we did when handling files in Chap. 18:

```
inStream = connection.getInputStream();
inDataStream = new DataInputStream(inStream);
```

We then create an output stream in the same way:

```
OutputStream outStream;
DataOutputStream outDataStream;
```

As you will see shortly, we have designed our client to send its IP address to the server once it is connected. So our next instructions to the server are to wait to receive a string on the input stream, and then to display a message on the console.

```
client = inDataStream.readUTF();
System.out.println("Address of client: " + client);
```

Once a connection has been established we want the server to perform the addition calculation for the client as many times as the client requires. Thus we provide a **while** loop that continues until the connection is lost:

```
while(connected)
{
        // read an integer from the client
        first = inDataStream.readInt();
        System.out.println("First number received: " + first);

        // read an integer from the client
        second = inDataStream.readInt();
        System.out.println("Second number received: " + second);

        sum = first + second;
        System.out.println("Sum returned: " + sum);

        // send the sum to the client
        outDataStream.writeInt(sum);
}
```

You can see that we read two integers from the input stream, displaying them each time on the console. We then calculate and display the sum, which we send back to the client on the output stream.

The `accept` method of `ServerSocket` throws an `IOException` when the connection is lost. Therefore we have coded the **`catch`** block so that the connected variable that controls the inner **`while`** loop is set to **`false`**, so that when the client closes the connection the server will no longer expect to receive integers, but will return to the top of the outer **`while`** loop, and wait for another connection request:

In a moment we will show you how to create a client program that requests a

```
catch (IOException e)
{
      connected = false;
}
```

service from our server. But before we do that, take a look at Fig. 23.1, which shows the result of a typical session from the point of view of the server, running in

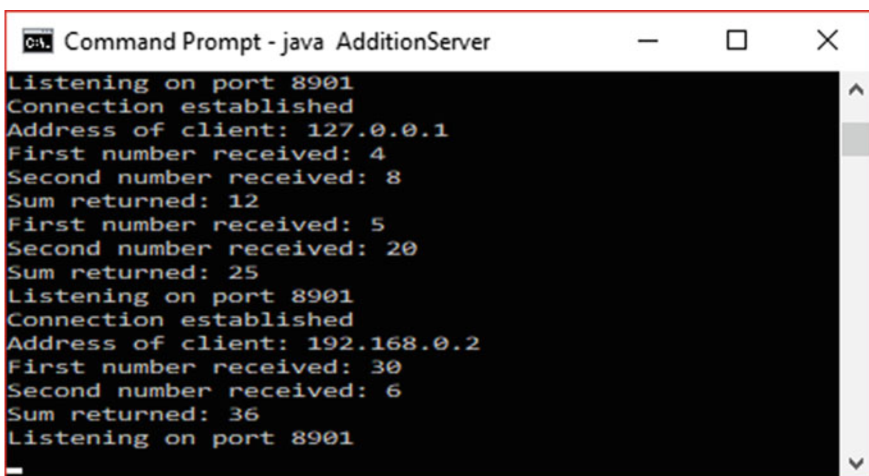


Fig. 23.1 A typical session for the addition server

a console. The server listens on port 8901; a client connects, requests two calculations and ends the session; the server waits for another connection; another client connects, requests one calculation and then ends the session, and the server once again waits for another client to connect.

23.4 A Simple Client Application

The client application will utilize a JavaFX interface as shown in Fig. 23.2.

Figure 23.2 should give you an idea of how the operation of the client will work. Once the address and port number of the remote host are known, the connection is established. Then the user is free to enter numbers and press the button to send these numbers to the server and display the result. The middle text box is used to display messages regarding the connection.

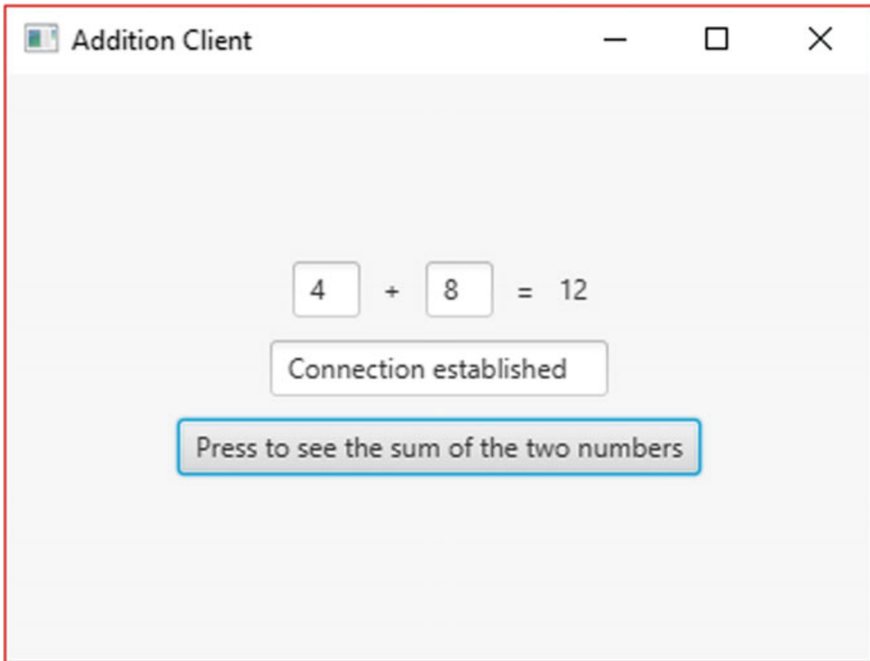


Fig. 23.2 The addition client

Here is the code for the `AdditionClient`:

AdditionClient

```
import java.io.InputStream;
import java.io.DataInputStream;
import java.io.OutputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Optional;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class AdditionClient extends Application
{
       private String remoteHost;
       private  int port;

       // declare low level and high level objects for input
       private InputStream inStream;
       private DataInputStream inDataStream;

       // declare low level and high level objects for output
       private  OutputStream outStream ;
       private DataOutputStream outDataStream;

       // declare a socket
       private Socket connection;

       @Override
       public void start(Stage stage)
       {
            getInfo(); // call the method that gets the information about the server

            // declare visual components
             TextField msg = new TextField();

            TextField firstNumber = new TextField();
            Label plus = new Label("+");
            TextField secondNumber = new TextField();

            Label equals = new Label("=");
            Label sum = new Label();

            Button calculateButton = new Button("Press to see the sum of the two numbers");

            // configure the scene
            msg.setMaxWidth(150);
            firstNumber.setMaxWidth(30);
            secondNumber.setMaxWidth(30);

            HBox hBox = new HBox(10);
            hBox.setAlignment(Pos.CENTER);
            hBox.getChildren().addAll(firstNumber, plus, secondNumber, equals, sum);
            VBox root = new VBox(10);

            root.setAlignment(Pos.CENTER);
            root.getChildren().addAll(hBox, msg,  calculateButton);

            Scene scene = new Scene(root, 400, 300);
            stage.setScene(scene);
            stage.setTitle("Addition Client");

            // show the stage
            stage.show();

            try
            {
```

```
            // attempt to create a connection to the server
            connection = new Socket(remoteHost, port);
            msg.setText("Connection established");

            // create an input stream from the server
            inStream = connection.getInputStream();
            inDataStream = new DataInputStream(inStream);

            // create an output stream to the server
            outStream = connection.getOutputStream();
            outDataStream = new DataOutputStream(outStream);

            // send the host IP to the server
            outDataStream.writeUTF(connection.getLocalAddress().getHostAddress());
        }

        catch (UnknownHostException e)
        {
           msg.setText("Unknown host");
        }

        catch (IOException ie)
        {
            msg.setText("Network Exception");
        }

        // specifiy the behaviour of the calculate button
        calculateButton.setOnAction(e ->
                    {
                       try
                       {
                            // send the two integers to the server
                            outDataStream.writeInt(Integer.parseInt(firstNumber.getText()));
                            outDataStream.writeInt(Integer.parseInt(secondNumber.getText()));

                            // read and display the result sent back from the server
                            int result = inDataStream.readInt();
                            sum.setText("" + result);
                        }
                        catch(IOException ie)
                        {
                        }
                     }
                                    );

    }

    private void getInfo()
    {
          Optional<String> response;

          // use the TextInputDialog class to allow the user to enter the host address
          TextInputDialog addressDialog = new TextInputDialog();
          addressDialog.setHeaderText("Enter remote host");
          addressDialog.setTitle("Addition Client");

          response = addressDialog.showAndWait();
          remoteHost = response.get();

          // use the TextInputDialog class to allow the user to enter port number
          TextInputDialog portDialog = new TextInputDialog();
          portDialog.setHeaderText("Enter port number");
          portDialog.setTitle("Addition Client");

          response = portDialog.showAndWait();
          port = Integer.valueOf(response.get());
    }

    public static void main(String[] args)
    {
              launch(args);
    }
}
```

We have declared a number of attributes, the first of which will hold values for the address of the remote host and the port number, and the remaining attributes are concerned with input and output streams, and the socket:

```
private String remoteHost;
private int port;

private InputStream inStream;
private DataInputStream inDataStream;

private  OutputStream outStream ;
private DataOutputStream outDataStream;

private Socket connection;
```

The first thing that we do inside the `start` method is to call a helper method `getInfo`. This will prompt the user to enter the address of the host machine on which the server is running, and the port number; the user of course must be aware of this information in order for the client to make the connection. So let's begin by briefly looking at the `getInfo` method:

```
private void getInfo()
{
      Optional<String> response;

      // use the TextInputDialog class to allow the user to enter the host address
      TextInputDialog addressDialog = new TextInputDialog();
      addressDialog.setHeaderText("Enter remote host");
      addressDialog.setTitle("Addition Client");

      response = addressDialog.showAndWait();
      remoteHost = response.get();

      // use the TextInputDialog class to allow the user to enter port number
      TextInputDialog portDialog = new TextInputDialog();
      portDialog.setHeaderText("Enter port number");
      portDialog.setTitle("Addition Client");

      response = portDialog.showAndWait();
      port = Integer.valueOf(response.get());
}
```

Here we are using the `TextInput Dialog` class that we introduced in Chap. 17 to get the host address and then to get the port number, as shown in Fig. 23.3.

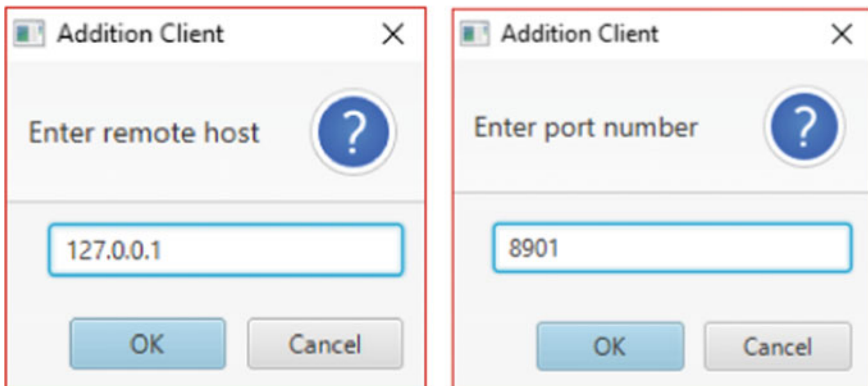


Fig. 23.3 Getting the remote host and port number from the user

Having called `getInfo` we go on to declare and configure the visual elements. Once we have shown the stage, we attempt to make the connection:

```
try
{
         // attempt to create a connection to the server
         connection = new Socket(remoteHost, port);
         msg.setText("Connection established");

         // create an input stream from the server
         inStream = connection.getInputStream();
         inDataStream = new DataInputStream(inStream);

         // create an output stream to the server
         outStream = connection.getOutputStream();
         outDataStream = new DataOutputStream(outStream);

         // send the host IP to the server
         outDataStream.writeUTF(connection.getLocalAddress().getHostAddress());
}

catch(UnknownHostException e)
{
         msg.setText("Unknown host");
}

catch(IOException ie)
{
         msg.setText("Network Exception");
}
```

The code must be placed within a **`try`** block. This is necessary because the constructor of the `Socket` class potentially throws two exceptions. As you can see it is called with two arguments, the name or IP address of the host machine (a `String`) and the port number (an **`int`**).

Creating a new `Socket` in this way transmits a message requesting a response from the remote machine specified, listening on the port in question. If the connection is established, and no exception is therefore thrown, the constructor goes on to display the message "Connection established" in the message area, and then to initialize the input and output streams. It finishes with this instruction:

```
outDataStream.writeUTF(connection.getLocalAddress().getHostAddress());
```

You will recall that we programmed the server so that the first thing it did after the connection was established was to wait for a string from the client. Here you can see how the client sends its address to the server on the output stream. It calls the `getLocalAddress` method of the `Socket` class. This returns an object of the `InetAddress` class. The `InetAddress` class holds a representation of an IP address and enables us to obtain the host name, or the IP address (as a `String`), with the methods `getHostName` and `getHostAddress` respectively.

Now we have to catch the exceptions that can be thrown by the constructor. As you can see there are two **catch** blocks. The first handles an UnknownHostException which will be thrown if the host we are trying to connect to is unknown. As you can see from the code, an appropriate message is placed in the message area.

If there is another network error (perhaps no server is running on the specified host), then an IOException is thrown and the message “Network Exception” is displayed.

Finally we need to provide the code that determines what happens when we press the button that gets the server to perform the addition for us:

```
calculateButton.setOnAction(e ->
                   {
                            try
                            {
                                 // send the two integers to the server
                                 outDataStream.writeInt(Integer.parseInt(firstNumber.getText()));
                                 outDataStream.writeInt(Integer.parseInt(secondNumber.getText()));

                                 // read and display the result sent back from the server
                                 int result = inDataStream.readInt();
                                 sum.setText("" + result);
                             }
                             catch(IOException ie)
                             {
                             }
                    }
                             );
```

This is pretty straightforward: we send the two numbers to the server and read the response. We enclose everything in a **try…catch** block so that the exceptions thrown by the readInt and writeInt methods are handled.

The socket example here is clearly rather elementary. Java provides a very wide range of possibilities for communication via sockets, for example secure sockets and sockets for multicasting. This is beyond the scope of this book, but it is hoped that we have given you a flavour for what is available so that those of you who want to develop your skills in this area are able to move forward. To help you do that we are going to provide two more examples. Firstly we will show you how to extend our addition server so that it accepts multiple clients at the same time. Secondly we are going to develop a rather more complex example, namely a chat application.

23.5 Connections from Multiple Clients

In our previous example the server could accept connections from only one client at a time; it wasn’t able to listen for other connections until the first connection had disconnected. If we want the server to accept multiple connections at the same time, then each connection has to run in its own thread.

We are going to create a class that extends `Thread` which we will call `AdditionServerThread`. But in order to understand the logic, let's first look at the new version of the addition server (`AdditionServerMultiple`) which is going to use this class:

AdditionServerMultiple

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class AdditionServerMultiple
{

    public static void main(String[] args)
    {
        final int port = 8901;
        AdditionServerThread thread;
        Socket socket;

        System.out.println("Listening for connections on port: " + port);
        try
        {
            ServerSocket listenSocket = new ServerSocket(port);

            while(true) // continuously listen for connections
            {
                socket = listenSocket.accept();
                thread = new AdditionServerThread(socket);
                thread.start();
            }
        }

        catch(IOException e)
        {
        }
    }
}
```

The logic is quite straightforward. Once the new `ServerSocket` is created we enter an infinite loop that continuously listens for connections. Once a connection is made, a new thread is created and started; a reference to the socket is sent as an argument. In this way, the server is able to support multiple connections on the same port.

Now we can think about the `AdditionServerThread` class. All the functionality that we previously saw in the `AdditionServer` will be placed in the `run` method of this class. There will also be one extra feature: each client that connects will generate its own id, an integer based on the number of connections that have been made so far. The server will be able to report in the console which client is making a particular request for an addition calculation. To illustrate this, a sample session is shown in Fig. 23.4.

```
Command Prompt - java  AdditionServerMultiple
Listening for connections on port: 8901
Connection established
Address of client: 127.0.0.1
Connection established
Address of client: 192.168.0.2
First number received from connection 1: 5
Second number received from connection 1: 6
Sum returned to connection 1: 11
Connection established
Address of client: 127.0.0.1
First number received from connection 3: 15
Second number received from connection 3: 3
Sum returned to connection 3: 18
First number received from connection 2: 34
Second number received from connection 2: 1
Sum returned to connection 2: 35
First number received from connection 2: 34
Second number received from connection 2: 1
Sum returned to connection 2: 35
```

Fig. 23.4 The addition server making connections with multiple clients and responding to requests

So let's take a look at the `AdditionServerThread` class:

AdditionServerThread

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class AdditionServerThread extends Thread
{
    private int id;
    private static int totalConnections;
    private final int port = 8901;

    // declare a "general" socket
    private final  Socket connection;

    // declare low level and high level objects for input
    private InputStream inStream;
    private DataInputStream inDataStream;

    // declare low level and high level objects for output
    private OutputStream outStream;
    DataOutputStream outDataStream;

    // declare other variables
    private String client;
    private int first, second, sum;
    private boolean connected;

    public AdditionServerThread(Socket socketIn)
    {
         connection = socketIn;
    }

    @Override
    public void run()
    {
        try
        {
            connected = true;
            System.out.println("Connection established");
```

```
            totalConnections++;     // increase the total number of connections
            id = totalConnections;  // assign an id

            // create an input stream from the client
            inStream = connection.getInputStream();
            inDataStream = new DataInputStream(inStream);

            // create an output stream to the client
            outStream = connection.getOutputStream ();
            outDataStream = new DataOutputStream(outStream);

            // wait for a string from the client
            client = inDataStream.readUTF();
            System.out.println("Address of client: " + client);

            while(connected)
            {
                // read an integer from the client
                first = inDataStream.readInt();
                System.out.println("First number received from connection " +  id + ": " + first);

                // read an integer from the client
                second = inDataStream.readInt();
                System.out.println("Second number received from connection " + id + ": " + second);

                sum = first + second;
                System.out.println("Sum returned to connection " + id + ": " + sum);

                // send the sum to the client
                outDataStream.writeInt(sum);
            }
        }

        catch(IOException e)
        {
            connected = false;
        }
    }
}
```

As we have said, the functionality now resides in the `run` method. There is nothing very new here, except for the fact that an id is assigned to the new connection, and this value is reported whenever this thread requests a calculation. You will see that a **`static`** attribute, `totalConnections`, has been declared, and within the `run` method this attribute is incremented each time a new connection is made. The current value is then assigned to the id of this connection.

You should also note that the socket on which the client is connected is received as a parameter and assigned in the constructor.

23.6 A Client–Server Chat Application

The final application that we are going to develop is a chat application. Figure 23.5 shows the sort of thing we are talking about.

The first thing to point out is that the only difference between the client and the server is the fact that initially the server waits for the client to initiate a connection —once the connection is established the behaviour is the same.

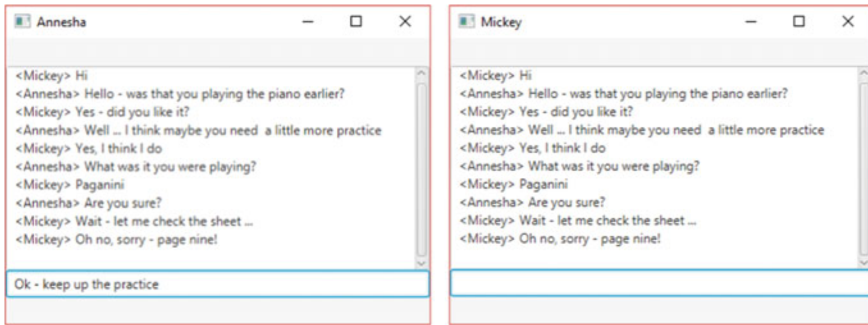


Fig. 23.5 A client–server chat application

Both the client and the server have to be able to listen for connections, and at the same time be capable of sending messages entered by the user. They will therefore need to be multi-threaded. The main thread will allow the user to enter messages which it will send to the remote program. The other thread will listen for messages from the remote application and display them in the text area.

When the thread is created it will need to receive a reference to the text area where the messages are to be displayed, and a reference to the socket connection. It will need to create an input stream which must be associated with this connection. We will be building JavaFX applications, so the threads will require the creation of `Task`s. The `call` method of each task will be written so that the thread continuously waits for messages on the input stream and then displays them in the text area. Both the client and the server classes will need to create an object of this thread and start the thread running.

We have designed our application so that rather than having a button that has to be pressed, the message is sent and echoed in the text area when the <Enter> key is pressed. As you will see in a moment, in order to achieve this the class must provide code for the `setOnKeyReleased` method of `TextField`. This method will check whether the <Enter> key was pressed.

We'll begin by looking at the code for the server:

ChatServer

```
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.OutputStream;
```

```
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Optional;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ChatServer extends Application
{
    // declare and initialise the text display area
    private TextArea textWindow = new TextArea();


    private OutputStream outStream; // for low level output
    private DataOutputStream outDataStream; // for high level output

    private ListenerTask listener; // required for the server thread

    private final int port = 8901;
    private String name;

    @Override
    public void start(Stage stage)
    {
        getInfo(); // call method that gets user name
        startServerThread();  // start the sever thread

        TextField inputWindow = new TextField();

        // configure the behaviour of the input window
        inputWindow.setOnKeyReleased(e ->
                    {
                        String text;

                        if(e.getCode().getName().equals("Enter"))  // if the <Enter> key was pressed
                        {
                            text = "<" + name + "> " +  inputWindow.getText() + "\n";
                            textWindow.appendText(text); // echo the text
                            inputWindow.setText(""); // clear the input window

                            try
                            {
                                outDataStream.writeUTF(text); // transmit the text
                            }

                            catch(IOException ie)
                            {
                            }
                        }
                    }
                                    );

        // configure the visual components
        textWindow.setEditable(false);
        textWindow.setWrapText(true);
        VBox root = new  VBox();
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(textWindow, inputWindow);
        Scene scene = new Scene(root, 500, 300);
        stage.setScene(scene);
        stage.setTitle(name);
        stage.show();
   }

   private void startServerThread()
   {
        Socket connection; // declare a "general" socket
        ServerSocket listenSocket; // declare a server socket

         try
```

```
        {
            // create a server socket
            listenSocket = new ServerSocket(port);

            // listen for a connection from the client
            connection = listenSocket.accept();

            // create an output stream to the connection
            outStream = connection.getOutputStream ();
            outDataStream = new DataOutputStream(outStream );

            // create a thread to listen for messages
            listener = new ListenerTask(textWindow, connection);

            Thread thread = new Thread(listener);
            thread.start(); // start the thread
        }
        catch (IOException e)
        {
            textWindow.setText("An error has occured");
        }
    }

    // method to get information from user
    private void getInfo()
    {
        Optional<String> response;

        // get user name
        TextInputDialog textDialog = new TextInputDialog();
        textDialog.setHeaderText("Enter user name");
        textDialog.setTitle("Chat Server");
        response = textDialog.showAndWait();
        name =  response.get();

        // provide information to the user before starting the server thread
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setTitle("Chat Server");
        alert.setHeaderText
             ("Press OK to start server. The dialogue window will appear when a client connects.");
        alert.showAndWait();
    }

    @Override
    public void stop()
    {
        System.exit(0); // terminate application when the window is closed
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

We start by declaring the attributes:

```
private TextArea textWindow = new TextArea();

private OutputStream outStream;
private DataOutputStream outDataStream;

private ListenerTask listener;

private final int port = 8901;
private String name;
```

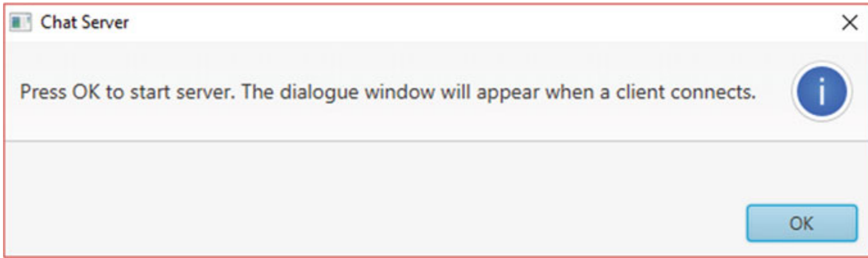


Fig. 23.6 An information alert for the chat server

We have declared and initialised a `TextArea` object, which is where the messages will be displayed. This needs to be an attribute of the class because it will later be passed to the task that listens for and displays the client messages.

Next we have declared a variable of type `ListenerTask`. This is the task that is required for the thread that listens for messages from the remote application, in this case the client; as you will see later the client will also make use of this class in order that it can receive messages from the server. You will see the code for the `ListenerTask` in a moment.

Next we declare a constant for the port number, which, for convenience, we have hard-coded, and finally we have declared a variable to hold the user name.

Now for the `start` method. We begin by calling a helper method, `getInfo`, which will prompt the user to enter a chat name; this uses the `TextInputDialog` class in the same way as you saw in the addition server example. Once the name is entered, we go on to create an information alert, in the way that we explained in Chap. 17:

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Chat Server");
alert.setHeaderText
     ("Press OK to start server. The dialogue window will appear when a client connects.");
```

This causes the following dialogue to appear (Fig. 23.6).

Once the user has acknowledged the message by pressing the OK button, a helper method, `startServerThead` is called. As we shall see in a moment, this method begins by waiting for a connection, and for this reason it is important that we call this method before showing the scene graphic. In a JavaFX application, once the stage is shown, any routine that effectively runs in the background should be placed in a separate thread. To avoid having to create an additional thread for this purpose we have waited for the connection to be established before creating and showing the scene graphic.

```
private void startServerThread()
{
    Socket connection;
    ServerSocket listenSocket;

    try
    {

        listenSocket = new ServerSocket(port);
        connection = listenSocket.accept();

        outStream = connection.getOutputStream ();
        outDataStream = new DataOutputStream(outStream );

        listener = new ListenerTask(textWindow, connection);

        Thread thread = new Thread(listener);
        thread.start();
    }

    catch (IOException e)
    {
        textWindow.setText("An error has occured");
    }
}
```

So let's now take a closer look at the `startServerThread` method.

There is nothing here that is particularly new—you have already seen how we create a server socket and listen for a connection; and you have seen how we associate a data stream with that connection. Notice, however, the last three lines of the **`try`** block. Here we create an instance of `ListenerTask`, which we need in order to create the thread that will listen for remote messages. Notice that we send a reference to the text window and a reference to the connection. We then go on to create and start the thread.

Once the client has connected, the application goes on to deal with declaring and configuring the visual components before finally showing the scene graphic. We should draw your attention to the code for specifying the behaviour of the input window when the user types a message:

```
inputWindow.setOnKeyReleased(e ->
                    {
                        String text;

                        if(e.getCode().getName().equals("Enter"))
                        {
                            text = "<" + name + "> " +  inputWindow.getText() + "\n";
                            textWindow.appendText(text);
                            inputWindow.setText("");

                            try
                            {
                                outDataStream.writeUTF(text);
                            }

                            catch(IOException ie)
                            {
                            }
                        }
                    }
                            );
```

Each time a key is pressed and then released we check whether the key released was the <Enter> key by invoking the `getCode` method of `KeyEvent`. This returns a `KeyCode` object; the name of the key is then retrieved using the

getName method of KeyCode. If the key pressed was the <Enter> key then a string is created from the user name (in angle brackets) plus the text entered, followed by a newline character ('\n'). This string is then appended to the text area, and the input window is blanked, ready for more input. As well as echoing the user's message on the server screen it must, of course be transmitted to the client via the output stream.

We want the program to terminate when the window is closed, and in this case we have done this by implementing the stop method of the JavaFX application:

```
public void stop()
{
       System.exit(0);
}
```

The instruction System.exit(0) will terminate the system normally.

Now let's look at the code for the ListenerTask class, which forms the basis of the thread that handles messages from the remote user:

ListenerTask

```
import java.io.DataInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;
import javafx.concurrent.Task;
import javafx.scene.control.TextArea;

public class ListenerTask extends Task
{
    private InputStream inputStream; // for low level input
    private DataInputStream dataInputStream; // for high level input
    private TextArea window; // a reference to the text area where the message will be displayed
    private Socket connection; // a reference to the connection

    // constructor receives references to the text area and the connection
    public ListenerTask(TextArea windowIn, Socket connectionIn)
    {
         window = windowIn;
         connection = connectionIn;

         try
         {
              // create an input stream from the remote machine
              inputStream = connection.getInputStream();
              dataInputStream = new DataInputStream(inputStream);
         }

        catch(IOException e)
        {
        }
    }

    @Override
    protected Void call()
    {
         String msg;
         while(true)
         {
             try
             {
                 msg = dataInputStream.readUTF(); // read the incoming message
                 window.appendText(msg); // display the message
             }

             catch(IOException e)
             {
             }
         }
     }
}
```

As you can see, the attribute declarations include references to the objects that will be needed for the input stream, as well as a reference to a `TextArea` and a `Socket`. The constructor receives a `TextArea` object and a `Socket` object, and these are assigned to the relevant attributes. A `ListenerTask` object will therefore have access to the text window and the connection associated with the parent object. The constructor then goes on to establish the input stream:

```
public ListenerTask(TextArea windowIn, Socket connectionIn)
{
     window = windowIn;
     connection = connectionIn;

     try
     {
          inputStream = connection.getInputStream();
          dataInputStream = new DataInputStream(inputStream);
     }

     catch(IOException e)
     {
     }
}
```

Now the `call` method:

```
protected Void call()
{
    String msg;
    while(true)
    {
        try
        {
            msg = dataInputStream.readUTF();
             window.appendText(msg);
        }

        catch(IOException e)
        {
        }
    }
}
```

You can see that once the corresponding thread is started, an infinite loop is implemented so that it continuously reads messages from the data stream, and then displays the message in the text area associated with the server or client program that created the thread.

Now we can look at the client application:

ChatClient

```
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Optional;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ChatClient extends Application
{
    // declare and initialize the text display area
    private TextArea textWindow = new TextArea();

    private OutputStream outStream; // for low level output
    private DataOutputStream outDataStream; // for high level output

    private ListenerTask listener; // required for the cleint thread

    private int port; // to hold the port number of the server
    private  String remoteMachine; // to hold the name chosen by the user

    private String name;


    @Override
    public void start(Stage stage)
    {
        getInfo(); // call method that gets user name and server details
        startClientThread(); // start the client thread

        TextField inputWindow = new TextField();

        // configure the behaviour of the input window
        inputWindow.setOnKeyReleased(e ->
                    {
                        String text;

                        if(e.getCode().getName().equals("Enter")) // if the <Enter> key was pressed
                        {
                            text = "<" + name + "> " +  inputWindow.getText() + "\n";
                            textWindow.appendText(text); // echo the text
                            inputWindow.setText(""); // clear the input window

                            try
                            {
                                outDataStream.writeUTF(text); // transmit the text
                            }
                            catch(IOException ie)
                            {
                            }
                        }
                    }
                                );

        // configure the visual components
        textWindow.setWrapText(true);
        textWindow.setEditable(false);
        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(textWindow, inputWindow);
        Scene scene = new Scene(root, 500, 300);
        stage.setScene(scene);
        stage.setTitle(name);
        stage.show();
```

```
    }

    private void startClientThread()
    {
        Socket connection;  // declare a "general" socket

        try
        {
            // create a connection to the server
            connection = new Socket(remoteMachine, port);

            // create output stream to the connection
            outStream = connection.getOutputStream();
            outDataStream = new DataOutputStream (outStream);

            // create a thread to listen for messages
            listener = new ListenerTask(textWindow, connection);
            Thread thread = new Thread(listener);
            thread.start(); // start the thread
        }

        catch(UnknownHostException e)
        {
            textWindow.setText("Unknown host");
        }

        catch (IOException e)
        {
            textWindow.setText("An error has occured");
        }
    }

    // method to get information from user
    private void getInfo()
    {
        Optional<String> response;

        // get address of host
        TextInputDialog textDialog1 = new TextInputDialog();
        textDialog1.setHeaderText("Enter remote host");
        textDialog1.setTitle("Chat Client");
        response = textDialog1.showAndWait();
        remoteMachine = response.get();

        // get port number
        TextInputDialog textDialog2 = new TextInputDialog();
        textDialog2.setHeaderText("Enter port number");
        textDialog2.setTitle("Chat Client");
        response = textDialog2.showAndWait();
        port = Integer.valueOf(response.get());

        // get user name
        TextInputDialog textDialog3 = new TextInputDialog();
        textDialog3.setHeaderText("Enter user name");
        textDialog3.setTitle("Chat Client");
        response = textDialog3.showAndWait();
        name =  response.get();

    }

    @Override
    public void stop()
    {
        System.exit(0); // terminate application when the window is closed
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

As you can see there is not a great deal of difference between the client and the server. The only significant differences are:

- The client needs to know the address of the host that is running the server, so there is an additional attribute, a string, to hold this address; the `port` attribute is not assigned a value when it is declared, but instead is given a value by the

user. Thus, in addition to asking the user to choose a name, the `getInfo` method now also requests information about the host machine and the port number on which the server is listening.
- In the `startClientThread` method there is no need for a `ServerSocket`; instead the socket is created by establishing the connection with the remote machine:

```
connection = new Socket(remoteMachine, port);
```

You are now in a position to test out our chat application—you will need to know the name or local IP address of the machine running the server. If you don't have access to two machines, then you can run both programs on the same machine—although this rather takes away the point! In the end of chapter exercises you are given some help with how to do this.

23.7 Self-test Questions

1. Explain what is meant by each of the following terms:

   (a) client;
   (b) server;
   (c) host;
   (d) port;
   (e) socket.

2. Explain the principles of *client–server* architecture, and describe how this is implemented in Java.
3. Which functions are provided by the Java `Socket` class?
4. Which additional functions are provided by the Java `ServerSocket` class?

23.8 Programming Exercises

1. Implement the `AdditionServer` and `AdditionClient` programs from this chapter. If you have more than one computer running on the same network you can run these programs on different machines. The client will need to be supplied with either the name or the local IP address of the host machine.

If both client and server are running on the same machine, you can use “localhost” as the name, or you can use the IP address 127.0.0.1, which references the local machine.

2. Implement the version of the addition server that accepts multiple clients, and test this out by connecting a number of clients. These can be on the same machine as the server, on remote machines, or a combination.

3. Implement and test out the chat application from this chapter. Again you can run both client and server on the same machine, but it is, of course, more fun to run them from different computers. Just a note, that if you are running them on the same machine, the applications will appear on top of one another, so you will need to move one out of the way to see the other one.

4. Write a server application that tells jokes to the client, and lets the client respond. A good example would be a classic “Knock Knock” joke. The client would receive the message “Knock Knock” from the server, and would be expected to reply “Who’s there?” and so on.
   You might be able to think of variations to this program. You could adapt it, for example, so that the a different joke is told each time a client connects (that is, if you actually know that many “Knock Knock” jokes!). Or perhaps a series of jokes could be told. You might also want to try allowing multiple clients to connect.

5 Try to devise a two- (or even more) player game that could be played across a network. An example might be noughts-and-crosses. The best approach would be to create a server that can deal with multiple connections—take a look at Sect. 23.5 to help with ideas for implementation.

Check for updates

Java in Context

24

Objectives:

By the end of this chapter you should be able to:

- *provide a brief history of the development of the Java language*;
- *identify the potential problems with* ***pointers****,* ***multiple inheritance*** *and* ***aliases***;
- *develop* `clone` *methods and* ***copy constructors*** *to avoid the problem of aliases*;
- *identify* ***immutable objects***;
- *explain the benefits of Java's* ***garbage collector***.

24.1 Introduction

Originally named *Oak*, Java was developed in 1991 by Sun Microsystems. The Java technology was later acquired by Oracle™. Originally, the intention was to use Java to program consumer devices such as video recorders, mobile phones and televisions. The expectation was that these devices would soon need to communicate with each other. As it turned out, however, this concept didn't take off until later. Instead, it was the growth of the Internet through the World Wide Web that was to be the real launch pad for the language.

The Java technology was acquired by Oracle™ in 2010, but the original motivation behind its development explains many of its characteristics. In particular, the **size** and **reliability** of the language became very important.

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_24

24.2 Language Size

Generally, the processor power of a system controlling a consumer device is very small compared with that of a PC; so the language used to develop such systems should be fairly compact. Consequently, the Java language is relatively small and compact when compared with other traditional languages. At the time Java was being developed, C++ was a very popular programming language. For this reason the developers of Java decided to stick to conventional C++ syntax as much as possible. Consequently Java syntax is very similar to C++ syntax.

Just because the Java language is relatively small, however, does not mean that it is not as powerful as some other languages. Instead, the Java developers were careful to remove certain language features that they felt led to common program errors. These include the ability for a programmer to create **pointers** and the ability for a programmer to develop **multiple inheritance** hierarchies.

24.2.1 Pointers

A pointer, in programming terms, is a variable containing an address in memory. Of course Java programmers can do something very similar to this—they can create *references*. Figure 24.1 repeats an example we showed you in Chap. 7.

In Fig. 24.1, the variable `myOblong` contains a reference (address in memory) of an `Oblong` object. The difference between a *reference* and a *pointer* is that the *programmer* does not have control over which address in memory is used—the *system* takes care of this. Of course, internally, the system creates a pointer and controls its location. In a language like C++ the programmer can directly

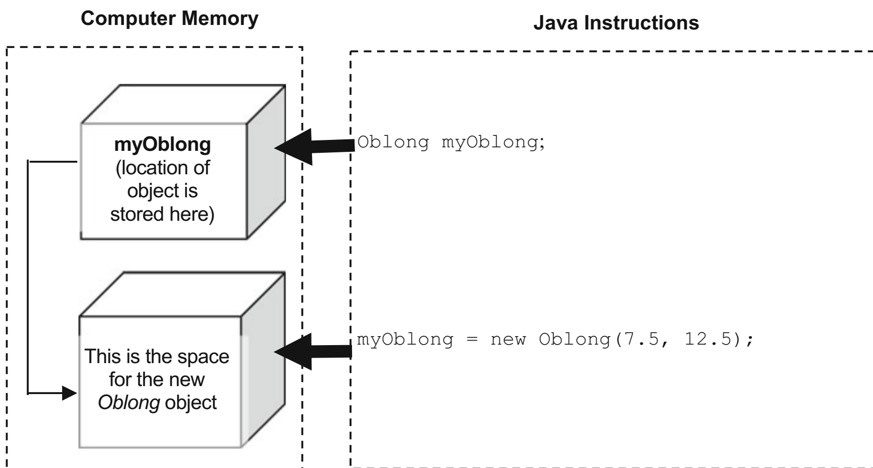


Fig. 24.1 An object variable in Java contains a reference to the object data

manipulate this pointer (move it along and back in memory). This was seen as giving the programmer greater control. However, if this ability is abused, critical areas of memory can easily be corrupted. For this reason the Java language developers did not allow users to manipulate pointers directly.

24.2.2 Multiple Inheritance

Inheritance is an important feature of object-oriented languages. Many object-oriented languages, such as C++ and Eiffel, allow an extended form of inheritance known as **multiple inheritance**. When programming in Java, a class can only ever inherit from at most one superclass. Multiple inheritance allows a class to inherit from more than one superclass (see Figs. 24.2 and 24.3).

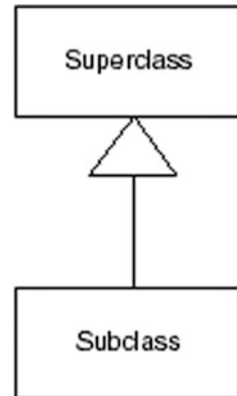


Fig. 24.2 Single inheritance

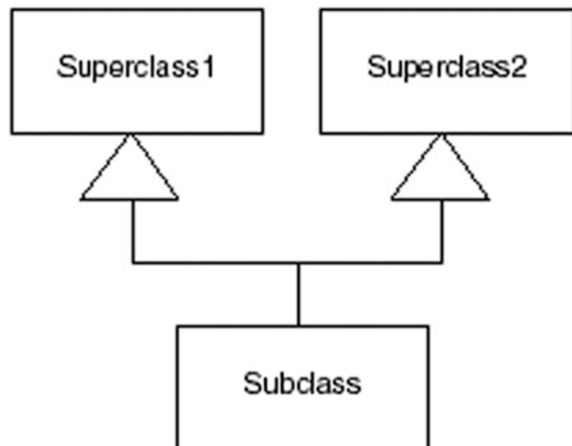


Fig. 24.3 Multiple inheritance

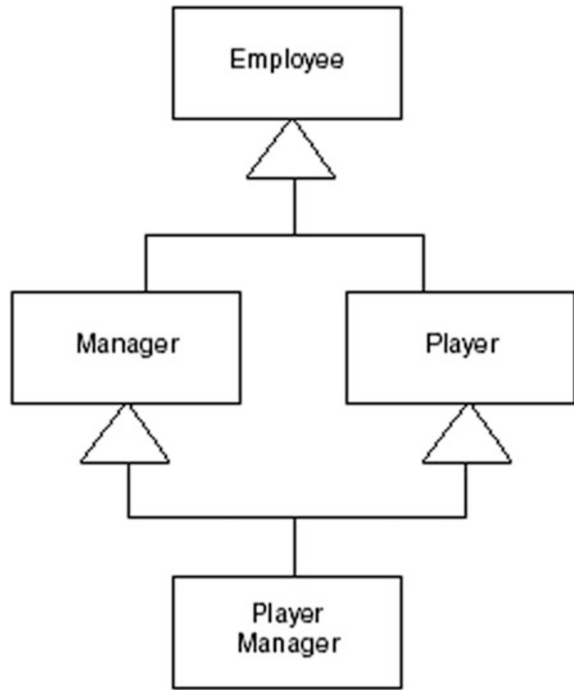


Fig. 24.4 A combination of single and multiple inheritance

As discussed in Chap. 13, multiple inheritance can lead to a variety of problems. The Java developers decided not to allow multiple inheritance for two reasons:

- it is very rarely required;
- it can lead to very complicated inheritance trees, which in turn lead to programming errors.

As an example of multiple inheritance, consider a football club with various employees. Figure 24.4 illustrates an inheritance structure that might be arrived at.

Here, a `PlayerManager` inherits from both `Player` and `Manager`, both of which in turn inherit from `Employee`! As you can see this is starting to get a little messy. Things become even more complicated when we consider method overriding. If both `Player` and `Manager` have a method called `payBonus`, which method should be called for `PlayerManager`—or should it be overridden? This is sometimes referred to as the **diamond problem** given the diamond like shape of the problematic design (as illustrated in Fig. 24.4).

Although Java disallows multiple inheritance it does offer a type of multiple inheritance—interfaces. As we have seen in previous chapters, a class can inherit from only one base class in Java but can implement many interfaces. Up to Java 8 this meant that the diamond problem could not arise as interfaces could contain **`abstract`** methods only.

Since Java 8, however, interfaces can contain **default** methods. As discussed in Chap. 13, **default** methods are regular methods that contain an implementation and reside in interfaces. These methods are automatically inherited by classes that implement these interfaces.

You might think that this could potentially lead to the diamond problem once again if we implement two or more interfaces that contain **default** methods with the same name. For example, let us look at the outline of a `PlayerInterface` that contains a **default** `payBonus` method:

```
public interface PlayerInterface
{
    // other regular abstract methods can be included here

    // a default method has an implementation
    default double payBonus()
    {
        return 1000;
    }
}
```

You can see how we add a **default** method into an interface. We use the keyword **default** and provide an implementation. In our implementation we have given a player a bonus of 1000. It is assumed all **default** methods are **public**, so we do not need to add this scope. Now consider a `ManagerInterface` that also contains a **default** `payBonus` method:

```
public interface ManagerInterface
{
    // other regular abstract methods can be included here

    // this default method has the same name as the default method in the PlayerInterface
    default double payBonus()
    {
        return 2000;
    }
}
```

You can see a manager has been given a bonus of 2000. Now, consider the following `PlayerManager` class that attempts to implement both of these interfaces:

```
public class PlayerManager implements PlayerInterface, ManagerInterface
{
   // implement abstract methods of PlayerInterface and ManagerInterface
}
```

If all we include in this `PlayerManager` class are implementations for the **abstract** methods contained in both the given interfaces this class will **not compile**. The reason for this is to avoid the diamond problem, as it would not be clear which version of `payBonus` to inherit. To resolve this Java insists that we override the `payBonus` method with an implementation of our own. Here is one possible solution:

```
public class PlayerManager implements PlayerInterface, ManagerInterface
{
   // implement abstract methods of PlayerInterface and ManagerInterface

   @Override
    public double payBonus()
    {
        return 3000;
    }
}
```

In this case we have given a player manager a bonus of 3000. Note when overriding the `payBonus` method we must mark this as a **`public`** (unlike **`default`** methods in interfaces which are always assumed to be **`public`** but not necessarily marked as **`public`**).

Now we have overridden the `payBonus` method there is no conflict to resolve and the given class will compile. We can use either (or both) of the inherited `payBonus` implementations when overriding these methods. We do so by making using the **`super`** keyword along with the interface name. For example, we might generate a player manager bonus by adding together the player bonus and the manager bonus as follows:

```
public class PlayerManager implements PlayerInterface, ManagerInterface
{
   // implement abstract methods of PlayerInterface and ManagerInterface

   @Override
    public double payBonus()
    {
       // we can access the inherited payBonus methods when overriding these methods
       return  PlayerInterface.super.payBonus()+ ManagerInterface.super.payBonus();
    }
}
```

24.3 Language Reliability

The Java language developers placed a lot of emphasis on ensuring that programs developed in Java would be reliable. One way in which they did this was to provide the extensive exception handling techniques that we covered in Chap. 14. Another way reliability was improved was to remove the ability for programmers to directly manipulate pointers as we discussed earlier in this chapter. Errors arising from pointer manipulation in other languages are very common. A related problem, however, is still prevalent in Java but can be avoided to a large extent. This is the problem of **aliasing**.

24.3.1 Aliasing

Aliasing occurs when the *same* memory location is accessed by variables with *different* names. As an example, we could create an object, `obj1`, of the `Oblong` class as follows:

```
Oblong obj1 = new Oblong (10, 20);
```

We could then declare a new variable, `obj2`, which could reference the same object:

```
Oblong obj2 = obj1;
```

Here `obj2` is simply a different name for `obj1`—in other words an **alias**. The effect of creating an alias is illustrated in Fig. 24.5.

In practice a programmer would normally create an alias only with good reason. For example, let us assume we have an array of `BankAccount` objects called `accountList` and we wish to overwrite one `BankAccount` in the list with the adjacent `BankAccount`. We are able to make good use of aliasing by assigning an object reference to a different object with a statement like:

```
accountList[i] = accountList[i+1];
```

After this instruction, `accountList[i]` is pointing to the same object as `accountList[i+1]`. In this case that was the intention. However, a potential problem with a language that allows aliasing is that it could lead to errors arising inadvertently. Consider for example a `Customer` class that keeps track of just two bank accounts. Here is the outline of that class:

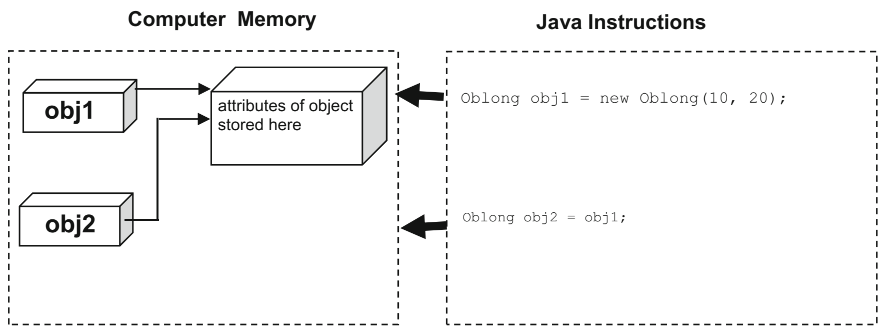


Fig. 24.5 Copying an object reference creates an alias

```
public class Customer
{
    // two private attributes to hold bank account details
    private BankAccount account1;
    private BankAccount account2;

    // more code here

    // two access methods
    public BankAccount getFirstAccount()
    {
        return account1;
    }

    public BankAccount getSecondAccount()
    {
        return account2;
    }
}
```

Consider the methods `getFirstAccount` and `getSecondAccount`. In each case we have sent back a reference to a **`private`** attribute, which is itself an object. We did this to allow users of this class to interrogate details about the two bank accounts, with statements such as:

```
BankAccount tempAccount = someCustomer.getFirstAccount();
System.out.println("balance of first account = "+tempAccount.getBalance());
```

Let us assume that this produced the following output:

balance of first account = 250.0

This is fine, but the `tempAccount` object, that we have just created, is now an alias for the **`private`** `BankAccount` object in the `Customer` class. It can be used to manipulate this **`private`** `BankAccount` object without going through any `Customer` methods. To demonstrate this let us withdraw money from the alias:

```
tempAccount.withdraw(100); // withdraw 100 from alias
```

Now let us go back and examine the bank account in the `Customer` class:

```
double balance = someCustomer.getFirstAccount().getBalance();
System.out.println("balance of first account = " + balance);
```

In this case we have retrieved the first bank account, and its balance in one instruction. We then display this balance, giving the following output:

balance of first account = 150.0

The balance of this internal account has been reduced by 100 without the Customer class having any control over this! From this example you can see how dangerous aliases can be.

There are a few examples in this book where we have returned references to **private** objects, but we have been careful not to take advantage of this by manipulating **private** attributes in this way. However, the important point is that they *could* be manipulated in that way. In order to make classes extra secure (for example, in the development of critical systems), aliasing should be avoided.

The problem of aliases arises when a copy of an object's *data* is required but instead a copy of the object's *reference* is returned. These two types of copies are sometime referred to as *deep copy* (for a copy of an object's data) and *shallow copy* (for a copy of an object's reference). By sending back a shallow copy, the original object can be manipulated, whereas a deep copy would not cause any harm to the original object.

In order to provide such a deep copy, a class should define a method that returns an exact copy of the object data. Such a method exists in the Object class, but this should be overridden in any user-defined class. The method is called clone. We want to send back copies of BankAccount objects, so we need to include a clone method in the original BankAccount class.

24.3.2 Overriding the *clone* Method

You have seen examples of overriding Object methods before. In Chap. 15, for instance, we overrode the toString and hashCode methods in the Object class. There is one important difference, however, between the clone method and other Object methods such as hashCode and toString. The clone method is declared as protected in the Object class, whereas methods such as hashCode and toString are declared as public.

Methods which are **protected** can only be called from within the same package (Object is in the java.lang package), or *within* subclasses. Methods which are **protected** are *not* part of the external interface of a class.

So if we wish to provide a clone method for any class, we are *forced* to override the clone method from Object. When we override this method we must make it **public** and not **protected**. When overriding methods you are able to give them wider access modifiers but not less—so a **protected** method can be overridden to be **public**, but not vice versa. The return type of the clone method is always Object:

```
@Override

// clone methods you write must have this interface
public Object clone() // must be a public method
{
    // code goes here
}
```

There were sound security reasons for the Java developers forcing you to override the `clone` method if you wish objects of your classes to be cloned, rather than allow objects of *all* classes to use the `clone` method in the `Object` class; because you might be developing a class in which you did not want objects of that class to be cloned.

However, we *do* want to provide the original `BankAccount` class with a `clone` method. Such a method would allow the `Customer` class to send back clones of `BankAccount` objects, rather than aliases as it is currently doing. Here is the outline of the `BankAccount` class with one possible implementation of such a method:

```
public class BankAccount
{
    // private attributes as before
    private String accountNumber;
    private String accountName;
    private double balance;

    // previous methods go here

    // now provide a clone method
    public Object clone()
    {
        // call contsructor to create a new object identical to this object
        BankAccount copyOfThisAccount = new BankAccount (accountNumber, accountName);
        /* after this the balance of the two bank accounts might not be the same,
           so copy the balance as well */
        copyOfThisAccount.balance = balance;
        // finally, send back this copy
        return copyOfThisAccount;
    }
}
```

Notice that in order to set the balance of the copied bank account we have directly accessed the **`private`** `balance` attribute of the copy:

```
copyOfThisAccount.balance = balance;
```

This is perfectly legal as we are in a `BankAccount` class, so all `BankAccount` objects created within this class can access their **`private`** attributes. Now, whenever we need to copy a `BankAccount` object we just call the `clone` method. For example:

```
// create the original object
BankAccount ourAccount = new BankAccount ("98765432", "Charatan and Kans");
// now make a copy using the clone method, notice a type cast is required
BankAccount tempAccount = (BankAccount) ourAccount.clone();
// other instructions here
```

The `clone` method sends back an exact copy of the original account, not a copy of the reference (see Fig. 24.6).

Now, whatever we do to the copied object will leave the original object unaffected, and vice versa.

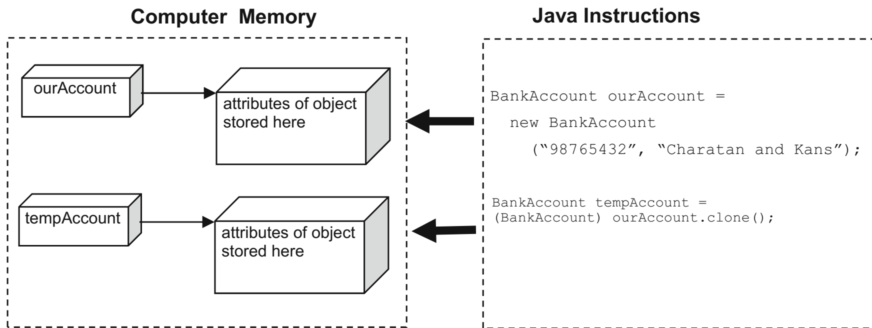


Fig. 24.6 The *clone* method creates a copy of an object

In a similar way, we can ensure that classes that contain `BankAccount` objects do not inadvertently send back references (and hence aliases) to these objects:

```
public class Customer
{
   // as before here

   // next two methods now send back clones, not aliases
   public BankAccount getFirstAccount()
   {
      return (BankAccount)account1.clone();
   }

   public BankAccount getSecondAccount()
   {
      return (BankAccount)account2.clone();
   }
}
```

Now, referrring to our earlier example, the problem is removed because of the use of the `clone` method in the `Customer` class, as illustrated in the fragment below:

```
Customer someCustomer = new Customer();

// some code to update someCutomer here


/* now a temporary variable is created to read details of first account but this is not an alias
it is a clone */
BankAccount tempAccount = someCustomer.getFirstaccount();
System.out.println("balance of first account = " + tempAccount.getBalance();
// assume the balance is displayed as 500
temp.withdraw(100); /*  because temp is a clone the private BankAccount attribute
                        account1 is unaffected */
System.out.println("balance of first account = " +  someCustomer.getFirstAccount().getBalance());
// the balance of the customer's first account will be still be 500
```

24.3.3 Immutable Objects

We said that methods that return references to objects actually create aliases and that this can be dangerous. However, these aliases are not *always* dangerous. Consider the following features of the original `BankAccount` class:

```
public class BankAccount
{
   private String accountNumber;

   // other attributes and methods here

   public String getAccountNumber()
   {
      return accountNumber;
   }
}
```

In this case the `getAccountNumber` method returns a reference to a **`private`** `String` object (`accountNumber`). This is an alias for the **`private`** `String` attribute. However, this alias causes no harm as there are no `String` methods that allow a `String` object to be altered. So, this alias cannot be used to alter the **`private`** `String` object.

Objects which have no methods to alter their state are known as **immutable objects**. `String` objects are immutable objects. Objects of classes that you develop may also be immutable depending on the methods you have provided. If such objects are immutable, you do not have to worry about creating aliases of these objects and do not need to provide them with `clone` methods. For example, let's go back to the `Library` application (consisting of a collection of `Book` objects) that we developed in Chap. 15. Rather than showing you the code, Fig. 24.7 shows you the UML design for the `Library` and `Book` classes.

As you can see, the `Library` class contains a collection of `Book` objects. These `Book` objects are part of the **`private`** `books` attribute in the `Library` class. However the `getBook` method returns a reference to one of these `Book` objects and so sends back an alias. This is not a problem, however, because if you look at the design of the `Book` class the only methods provided are `get` methods.

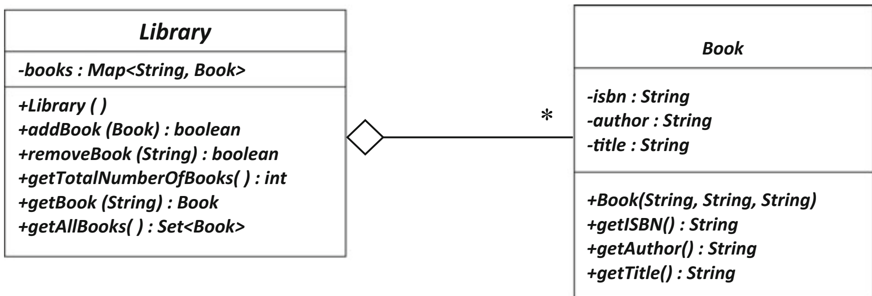


Fig. 24.7 Design for the *Library* application

In other words, there are no Book methods that can alter the attributes of the Book object once the Book object has been created. A Book object is an immutable object.

24.3.4 Using the *clone* Method of the *Object* Class

Although the clone method in the Object class is not made available as part of your class's external interface, it can be used *within* classes that you develop. In particular, you might wish to use it within a clone method that you write yourself, as it does carry out the task of copying an object for you—albeit with some restrictions.

The clone method from the Object class copies the *memory contents* allocated to the object attributes. This is sometimes referred to as a *bit-wise copy*. This means that it makes exact copies of attributes that are of primitive type, and it makes copies of references for attributes that are objects. Of course, a copy of a reference gives you an alias—but if the object in question is immutable, this is not a problem. This means that:

- if a class's attributes are all of primitive type, then make your clone method just call the clone method of Object;
- if a class's attributes include objects, and these objects are *all* immutable, then again make your clone method just call the clone method of Object;
- if a class's attributes include any objects which are *not* immutable, then you *cannot* rely upon the clone method of Object to make a sensible copy and so you must write your own instructions for providing a clone.

Bearing these points in mind, let us revisit the clone method for our BankAccount class.

```
public class BankAccount
{
   // attributes
   private String accountNumber;
   private String accountName;
   private double balance;

   public Object clone()
   {
      // code goes here
   }

   // other code here
}
```

To make a copy of a BankAccount object we need a new bank account with an identical account number, name and balance. The balance attribute is of type **double**, so a bit-wise copy would be fine here. The account name and number are both String objects; since strings are immutable a bit-wise copy is fine here also. This means the *entire* object can be safely copied using the clone method of

Object. Here is a first attempt at using this method within our own clone method—it will not compile!:

```
// this attempt to clone a BankAccount will not compile
public Object clone()
{
    // call 'clone' method of superclass Object
    return super.clone();
}
```

This will not compile at the moment because the clone method of Object checks whether developers of this class really want to allow cloning to go ahead.

To indicate that developers do want cloning to go ahead, they have to mark their class as implementing the Cloneable interface. This interface, much like Serializable, contains no methods. It is just used to mark a class with some extra information. So, in order to call the clone method of Object, we need to mark the BankAccount class as follows:

```
// marking this class Cloneable allows us to call clone method of Object
public class BankAccount implements Cloneable
{
    // code here can use 'super.clone()'
}
```

There is one last thing we need to do in order to use the clone method of the Object class. This method throws a checked CloneNotSupportedException if the calling class does not implement the Cloneable interface. Of course we know our class does implement this interface, but as this is a checked exception, we still need to provide a **try**…**catch** around the call to super.clone() to keep the compiler happy. Here is the modified BankAccount class:

```
// mark that objects of this class can be cloned
public class BankAccount implements Cloneable
{
    // attributes as before
    private String accountNumber;
    private String accountName;
    private double balance;


    // this method allows BankAccount objects to be cloned
    public Object clone()
    {
        try
        {
            return super.clone(); // call 'clone' from Object
        }
        catch (CloneNotSupportedException e) // will never be thrown!
        {
            return null;
        }
    }

    // other code here
}
```

Whether or not you use super.clone() in your implementation of the clone method, it is always a good idea to mark your class Cloneable, so it is clear that objects from your class can be cloned.

24.3.5 Copy Constructors

The previous sections demonstrated how `clone` methods can be used to avoid aliases by providing exact (deep copies) of an object. But, as you could see, implementing `clone` methods can be a little tricky and using `clone` methods requires type-casting. A popular alternative to this approach is to provide copy constructors instead. Copy constructors provide a way of creating an exact copy of an object from an object sent as a parameter to a constructor. Languages like C++ automatically provide copy constructors, but in Java we have to implement them ourselves. Doing so is fairly straightforward. Let's return to the original `BankAccount` class and assume we have not included a `clone` method. Instead we will provide an additional copy constructor that receives a `BankAccount` object as a parameter and copies this parameter's attributes to make a new exact copy. Here is the outline of the class:

```
public class BankAccount
{
    // original attributes here

    // the original constructor
    public BankAccount(String numberIn, String nameIn)
    {
        accountNumber = numberIn;
        accountName = nameIn;
        balance = 0;
    }

    // the copy constructor
    public BankAccount(BankAccount accIn)
    {
        accountNumber = accIn.accountNumber;
        accountName = accIn.accountName;
        balance = accIn.balance;
    }

   // original methods here plus a toString method

}
```

You can see that, as well as the original constructor, we have provided a copy constructor that receives a `BankAccount` object and makes an exact copy by copying across every attribute value of the parameter object:

```
// the copy constructor
public BankAccount(BankAccount accIn)
{
    accountNumber = accIn.accountNumber; // copy account number
    accountName = accIn.accountName; // copy account name
    balance = accIn.balance; // copy account balance
}
```

The `CopyConstructorDemo` program below demonstrates how easy to is to use a copy constructor to create copies of objects. Note we are assuming a `toString` method has been included in the `BankAccount` class for ease of testing:

CopyConstructorDemo

```
public class CopyConstructorDemo
{
    public static void main(String[] args)
    {
        BankAccount b1 = new BankAccount ("001", "Justin Thyme"); // balance zero
        b1.deposit(100); // balance 100
        System.out.println("first object "+b1);
        BankAccount b2 = new BankAccount(b1); // create copy via copy constructor
        System.out.println("second object "+b2);// display copy
        b1.withdraw(50);// modify original object
        System.out.println("first object "+b1);
        System.out.println("second object "+b2);// second object untouched
    }
}
```

We have created a `BankAccount` object, `b1`, and deposited some funds into this object via the `deposit` method before displaying it (using its `toString` method). The next line is the key instruction where we use the copy constructor to create a new `BankAccount` object, `b2`, that is an exact copy of the first object:

```
BankAccount b2 = new BankAccount(b1); // create copy via copy constructor
```

You can see how simple this is. There is no need to type-cast as with a `clone` method. We then display the copy, before withdrawing money from the first object and displaying both objects again. Here is the program output:

first object (001, Justin Thyme, 100.0)
second object (001, Justin Thyme, 100.0)
first object (001, Justin Thyme, 50.0)
second object (001, Justin Thyme, 100.0)

As expected, the second object is an exact copy of the first object. Once the copy has been created we can modify the first object without modifying the copy.

Which technique you use for creating object copies (`clone` methods or copy constructors) is really up to you. You might find using the `Object clone` method simpler if the object in question has many attributes whereas a copy constructor may be easier for objects that have fewer attributes or attributes that are not able to be cloned simply (such as non-immutable objects).

24.3.6 Garbage Collection

When an object is created using the **`new`** operator, a request is being made to grab an area of free computer memory to store the object's attributes. Because this memory is requested during the running of a program, not during compilation, the compiler cannot guarantee that enough memory exists to meet this request. Memory could become exhausted for two related reasons:

- continual requests to grab memory are made when no more free memory exists;
- memory that is no longer needed is not released back to the system.

These problems are common to all programming languages and the danger of memory exhaustion is a real one for large programs, or programs running in a small memory space. Java allows both of the reasons listed above to be dealt with effectively and thus ensures that programs do not crash unexpectedly.

First, exception-handling techniques can be used to monitor for memory exhaustion and code can be written to ensure the program terminates gracefully. More importantly, Java has a built-in garbage collection facility to release unused memory. This is a facility that regularly trawls through memory looking for locations used by the program, freeing any locations that are no longer in use.

For example consider the program below.

Tester

```
import java.util.Scanner;

public class Tester
{
   public static void main(String[] args)
   {
      char ans;
      Scanner keyboard = new Scanner (System.in);
      Oblong object; // reference to object created here
      do
      {
         System.out.print("Enter length: ");
         double length = keyboard.nextDouble();
         System.out.print("Enter height: ");
         double height = keyboard.nextDouble();
         // new object created each time we go around the loop
         object = new Oblong(length, height);
         System.out.println("area = "+ object.calculateArea());
         System.out.println("perimeter = "+ object.calculatePerimeter());
         System.out.print("Do you want another go? ");
         ans = keyboard.next().charAt(0);
      }  while (ans == 'y' || ans == 'Y');
   }
}
```

Here, a new object is created each time we go around the loop. The memory used for the previous object is no longer required. In a language like C++ the memory occupied by old objects would not be destroyed unless the programmer added instructions to do so. So if the programmer forgot to do this, and this happened on a large scale in your C++ program, the available memory space could easily be exhausted. The Java system, however, regularly checks for such unused objects in memory and destroys them.

Although automatic garbage collection does make extra demands on the system (slowing it down while it takes place), this extra demand is considered by many to be worthwhile by removing a heavy burden on programmers. Nowadays many programming languages, such as C# and Python, also include a garbage collection facility.

Table 24.1 TIOBE programming community index

Position Aug 2018	Position Aug 2017	Programming language
1	1	Java
2	2	C
3	3	C++
4	5	Python
5	6	Visual Basic.NET
6	4	C#
7	7	PHP
8	8	JavaScript
9	–	SQL
10	14	Assembly Language

24.4 The Role of Java

While Java began life as a language aimed primarily at programming consumer devices, it has evolved into a sophisticated application programming language; competing with languages such as C++, Python and C#, to develop a wide range of applications. The security and reliability offered by the language has allowed the use of Java to be spread from desktop applications to network systems, web-based applications, set-top boxes, smart cards, computer games, smart phones and many more. To see an example of the enormous range of applications powered by Java visit the Oracle™ site at: http://go.java.

Table 24.1 gives the TIOBE programming community index of the ten most popular programming languages for August 2018.[1]

You can see that Java is at the top of this table, as it was last year. In fact it has been top of this index for many years.

24.5 What Next?

This chapter marks the end of our Java coverage for your second semester in programming. Although you have covered a lot of material, there is still more that you can explore. For example, we looked at how packages can be used to organise and distribute our Java applications in Chap. 19. The Java Programming Module System (JPMS) was introduced with the release of Java 9 and provides an even higher level of organisation to group together a collection of packages. We looked at Java applications that run over a local network in Chap. 23, but Java is also used

[1]The TIOBE index is a respected measure of the popularity of a programming language. For details of the table itself and of how it was compiled go to https://www.tiobe.com/tiobe-index/.

for the development of large distributed enterprise systems over wide area networks as well as cloud-based systems. We have also had a thorough look at JavaFX throughout this text but there is still much more you can find out about, including FXML—a Java FX tailored XML language developed by Oracle™. The good news is that you are now well placed to explore all these areas as well as many more. In the meantime, don't forget you can get further information on the Java language at the Oracle™ website https://www.oracle.com/java/.

Now that you have completed two semesters of programming we are pretty certain that you will have come to realize what an exciting and rewarding an activity it can be. So whether you are going on to a career in software engineering, or some other field in computing—or even if you are just going to enjoy programming for its own sake, we wish you the very best of luck for the future.

24.6 Self-test Questions

1. Distinguish between a *pointer* and a *reference*.
2. What does the term *multiple inheritance* mean and why does Java disallow it?
3. How do you implement a class that inherits two interfaces, both with a **`default`** method with the same name?
4. Consider the following class:

```
public class Critical
{
       private int value;

       public Critical (int valueIn)
       {
          value = valueIn;
       }
       public void setValue(int valueIn)
       {
          value = valueIn;
       }
       public int getValue ()
       {
          return value;
       }
}
```

   (a) Explain why `Critical` objects are not *immutable*.
   (b) Write fragments of code to create `Critical` objects and demonstrate the problem of *aliases*.
   (c) Develop a `clone` method in the `Critical` class (make use of the `clone` method of `Object` here).
   (d) Write fragments of code to demonstrate the use of this `clone` method.
   (e) What is the purpose of a *copy constructor*?
   (f) Develop a copy constructor for the `Critical` class.
   (g) Write fragments of code to demonstrate the use of this copy constructor.

5. Look back at the classes from the two case studies of Chaps. 11, 12 and 21.

   (a) Which methods in these classes return aliases?
   (b) Which aliases could be dangerous?
   (c) How can these aliases be avoided?

6. What are the advantages and disadvantages of a *garbage collection* facility in a programming language?

24.7 Programming Exercises

1. Implement the `Critical` class of self-test question 4 and then write a tester program to demonstrate the problem of aliases.
2. Amend the `Critical` class by adding a `clone` method as discussed in self-test question 4(c) and then amend the tester program you developed in the previous programming exercise to demonstrate the use of this `clone` method.
3. Amend the `Critical` class further by adding the copy constructor discussed in self-test question 4(e) and then amend the tester program you developed in the previous programming exercise to demonstrate the use of this copy constructor.
4. Implement the changes you identified in self-test question 5, in order to remove the aliases that might have been present in the classes from the two case studies.
5. Review all the classes that you have developed so far and identify any problems with aliases. Use the techniques discussed in this chapter to avoid these aliases.

Index

0–9
2D shapes, 281

A
abstract
  class, 250, 254, 255, 261, 359, 587
  method, 253, 254, 268, 359, 360, 361, 369, 370, 371, 374, 378, 383, 385, 412, 413, 436, 447, 458, 460, 482, 588, 649, 652, 656, 692, 693, 694
Abstract Windows Toolkit (AWT), 266
actors - use case model, 604
actual parameters, 95, 101, 110, 116
aggregation, 211, 310
airport case study, *see* case study, airport
`Alert` class, 518, 520
algorithms
  scheduling, 582
aliasing, 694, 695, 697
animations, 591, 593, 596, 597, 600
anonymous class, 357, 364–368, 378, 391, 587, 591, 597, 598
API, *see* Application Programming Interface
append mode - files, 535
Application Programming Interface (API), 369, 453, 641
applications
  deploying, 563
  JavaFX, 267–269, 302, 310, 336, 337, 413–415, 496, 541, 553, 554, 577, 587, 599, 677, 680, 682
  running from command line, 268, 415, 553, 560, 561, 575
application software, 3, 4, 16
arithmetic operators, 25, 26
array(s)
  accessing elements, 119, 124, 126, 187, 394
  creating, 120, 148, 149, 186
  elements, 120–122, 124–126, 137, 139, 142, 185, 210, 232
  index, 125–127, 137, 139, 143, 150, 154, 186, 395
  `length` attribute, 151, 156, 198, 200–202
  maximum, 122, 139, 140
  of objects, 163, 185, 186, 191
  returning from a method, 134
  membership, 141
  passing as parameters, 101, 129, 131
  ragged, 119, 155, 156, 160, 162
  search, 142, 143, 451
  sort, 457, 458, 463
  summation, 141
  two-dimensional, 119, 148–151, 153, 155, 156, 160, 232
  varargs, 131–134, 138, 159, 161
`ArrayIndexOutOfBoundsException`, 394, 395, 405, 422
`ArrayList` class, 188, 193, 211, 315, 317, 553, 376, 427–430, 450, 465
`Arrays` class, 456, 465, 647
assignment, 23–28, 34, 46, 103, 124–126, 200
asynchronous threads, 577, 599
attributes
  class, 109, 165, 224, 227, 236, 237, 328, 376, 477, 570, 608, 701
  `length`, 151, 156, 198, 199, 201, 202, 241
  `private`, 195, 262, 697, 698
  `protected`, 238, 262
  `public`, 195, 238, 262, 315
  `static`, 206, 676
autoboxing, 259
AWT, *see* Abstract Windows Toolkit

B
base class, 237, 394, 692
behaviour specifications, 603–605, 624

© Springer Nature Switzerland AG 2019

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8

BiConsumer interface, 447
BiFunction interface, 378
binary encoding, 530
binary files – reading and writing, 539, 540
BinaryOperator interface, 378
BiPredicate interface, 378
boolean type, 20, 122
BorderPane class, 292, 293, 303
BorderStroke class, 289, 294, 344, 346, 629, 631
BorderStrokeStyle class, 289, 294, 344, 346, 629, 631
BorderWidths class, 289, 294, 344, 346, 629, 631
bounded type parameters, 379
break statement, 56–59, 61, 65, 84–86
BufferedReader class, 529, 537, 538, 542
bugs, *see* errors
busy waiting, 577, 585, 599
byte type, 20

C
C ++, 4, 164, 208, 690, 691, 703, 705, 706
C# (C Sharp), 705, 706
calling method, 100, 105, 191, 312, 401, 413
card menu, 499, 513
Cascading Style Sheets (CSS), 469, 491, 496, 639
case statement, 55–58, 113, 115, 615
case study
  airport, 605
  student hostel, 308, 334–336, 350, 354
catch, 393, 401–406, 408, 421, 423, 540, 573, 667, 673
catching an exception, 401, 422
char type, 22, 23
Character class, 362
check boxes, 267, 282, 499, 509, 511, 512
checked exceptions, 395, 422
ChoiceDialog class, 518, 520, 523, 639
claiming an exception, 397, 398, 422
class(es)
  anonymous, 357, 364–368, 378, 391, 587, 591, 597, 598
  attributes, 109, 165, 200, 203, 236–238, 312, 339, 469, 529, 550, 570, 593, 701
  abstract, 235, 250, 251, 254, 255, 261, 359
  collection, *see* collection classes
  final, 257
  inheritance extending with, 236, 246, 251, 260, 357, 359, 691
  inner, 364, 391
  methods, 116, 117, 170, 205, 208, 229, 238, 374, 396, 399, 402, 411, 412, 456, 457
class method, 205, 208, 229, 238, 396, 411, 412, 456, 457
classpath, 562, 564, 569, 574
client-server model, 661, 676
clone method, 689, 697–704, 707, 708
Cloneable interface, 702
code layout, 319
collection classes
  generic, 188, 378, 383
Collection interface, 428, 435, 436, 645
combo boxes, 499, 507
context menu, 499, 503–506, 524
Color class, 272, 296
colours
  creating, 646
command line running applications from, 268, 415, 553, 560–563, 574, 575
comments
  Javadoc, 14, 307, 310, 313, 317, 318, 330, 332, 334, 421, 424, 609, 621, 635
Comparable interface, 458–460, 463, 465, 466
Comparator interface, 460, 461, 463, 465
comparison operators, 46, 446
compiling programs, 4
composition, 211, 603, 607, 638
compound containers, 301
concatenation operator, 15, 30
concatenating streams, 656
concurrent processes, 578
constants, creating, 25
constructor
  copy, 689, 703, 704, 707, 708
  default, 200, 201
  user defined, 200, 237
containment, 606, 607, 638
context menus, 499, 503, 505, 524
continue (key word), 86, 401
convenience method, 279, 384, 470, 472, 475, 476, 482, 497, 503
Consumer interface, 413, 435, 436, 442, 652
CornerRadii class, 344, 515, 516, 635
creating streams, 646
critical sections, 584
CSS, *see* Cascading Stye Sheets

D
data types, 19–21, 163, 223, 565
database, 353, 553, 564, 565, 567–575, 578, 642, 663

DataInputStream class, 540, 664–666, 669, 676, 682
DataOutputStream class, 539, 545, 664, 677, 685
DecimalFormat class, 297, 301, 344, 555
declaring variables, 21, 35
decrement operator, 29
default methods, 360, 361, 693, 694
deploying applications, 563
derived class, 237
design
  case study airport, 607, 625, 639
  case study student hostel, 309, 310, 335, 336
  program, 35
Dialog class, 348, 499, 518, 520, 523, 671, 680
diamond problem, 692, 693
Driver class, 564
DriverManager class, 567
double colon operator, 374
documentation, 317, 318, 330, 425, 453, 482, 562, 564, 567, 575, 604
dot operator, 169, 172, 186
double type, 21, 23, 24, 179
Double class, 287
do...while loop, 79–84, 90
driver(s)
  JDBC, 564
drop-down menus, 499, 500

E
embedding
  images, 483
  videos, 486
  webpages, 469
embedded software, 4, 5, 16
encapsulation, 169, 195–197, 224, 238
encoding files, 527, 530
enum, 609, 610
enumerated types
  implementing in Java, 603, 609, 622
  using with switch statements, 610, 615
enhanced **for** loop, 119, 138, 139, 141, 143, 158, 187, 190, 316, 331, 427, 434–436, 438–440, 446, 447, 464, 623, 639
environment variable, 562, 574
equals method
  defining, 450
  String class, 433, 445
  Object class, 450
errors
  compile-time errors, 564
  runtime errors, 564, 569
event-handling
  key events, 477
  mouse events, 470
exception(s)
  catching, 401, 403, 422
  checked, 393, 395, 398, 422, 587, 622, 702
  claiming, 397, 398, 422
  documenting, 421, 422
  exception classes creating, 419
  exception class hierarchy, 395
  in JavaFX applications, 415
  handling, 393, 395, 414, 544, 694, 705
  IOException, 394
  NullPointerException, 408, 445
  RuntimeException, 398, 400, 417–419
  throwing, 394, 401, 417, 422, 424
  unchecked, 393, 395, 422–424
executable JAR files, 563
expressions, 26, 27, 29, 30, 105, 106, 279, 280, 341, 357, 358, 368–371, 374, 387, 652, 655
extending classes with inheritance, 236, 357

F
fields data, 570
files
  access, 527, 545, 550
  append mode, 535
  binary, 530, 539, 540, 550
  closing, 403, 534
  encoding, 530, 549
  executable JAR, 563, 575, 639
  file-handling, 527, 530
  file pointers, 531, 545, 548
  object serialization, 542
  random access, 527, 531, 544, 550
  serial access, 527, 531, 550
  streams, 530
  text, 7, 530–532, 535, 536, 539, 541, 549, 551
final
  class, 257
  method, 257, 654
  variable, 25
final operations-streams, 656, 657
finally, 403–408, 423, 534, 573
File class, 488
file pointers, 531, 545, 548
File Transfer Protocol (FTP), 662, 663
FileInputStream class, 540
FileReader class, 537
FileWriter class, 534
float type, 21
Float class, 380, 389

FlowPane class, 292, 303, 485
fonts
  creating, 274, 295
for loop
  enhanced, 119, 137–139, 141, 143, 158, 187, 190, 316, 331, 427, 434–436, 438, 439, 446, 447, 464, 623, 639
forEach loop/method, 427, 435, 436, 438–440, 446, 447, 455, 464, 465, 623, 639, 645, 646, 649, 652
formatting
  numbers, 297
Function interface, 648
functional interfaces, 383, 387
FXML, 707

G
garbage collection, 704, 705, 708
generics
  out of the box interfaces, 378
  upper bound, 357, 379, 389
  wild card, 357, 382, 389, 482
generic collection classes, 428
Graphical User Interfaces (GUIs)
  design, 304, 334, 336
  exceptions, 525
GridPane class, 290, 291, 303
GUIs, *see* Graphical User Interfaces

H
hashCode method, 450–453, 464, 606, 614, 697
HashMap class, 427, 443, 446, 451, 452
HashSet class, 427, 436, 437
Hbox class, 267, 278, 279, 288, 289, 292, 303
heavyweight components - user interface, 266
host - network programming, 353
hostel case study, *see* case study student hostel
HTML, *see* Hypertext Markup Language
Hypertext Markup Language (HTML), 317, 318, 491
Hypertext Transfer Protocol (HTTP), 662

I
IDE, *see* integrated development environment
if statement, 41, 43–45, 47–49, 51, 55, 59, 60, 68, 72, 73, 87, 126, 140, 176, 216, 248, 393, 410, 441
if…else statement
  nested, 41, 53, 55
Image class, 484
ImageView class, 484
immutable objects, 689, 700, 701, 704
import statement, 32, 411, 429, 430, 555, 556, 574
increment operator, 29
indentation, 319
infinite streams, 656
InetAddress class, 672
information hiding, 196
inheritance
  defining, 236
  extending classes with, 236, 357
  implementing, 237, 361, 363
init method, 587
initializing
  attributes, 208
  variables, 24, 103, 208
inner class, 357, 364, 391
input
  devices, 527, 528
  from keyboard, 32, 33
  stream, 527, 529, 544, 664–667, 669, 676, 677, 682, 683
  validation, 65, 77–79, 92, 154, 287, 350, 531
input events
  key events, 469, 473, 477
  mouse events, 469, 470
InputStream class, 396, 666
InputStreamReader class, 529
instantiation, 167
int type, 20, 21, 23, 24
Integer class, 259, 288, 344, 397
integrated development environment (IDE), 6, 16
integration testing, 307, 308, 320, 354, 624
io package, 399, 553
interface
  abstract methods, 359–361, 692
  default methods, 360, 361, 693, 694
  functional, 369, 371, 374, 378, 383, 384, 387, 436, 458, 460, 582, 649, 652, 656
  out of the box, 378, 648
  static methods, 360, 378, 461
intermediate operations - streams, 642, 648, 658
Internet Protocol (IP) address, 567, 663
is-a-kind-of relationship, 237, 257
iteration
  do...while loop, 65, 66, 79–84, 90, 92
  enhanced **for** loop, 119, 138, 139, 141, 143, 158, 187, 190, 316, 331, 427, 434–436, 438–440, 446, 447, 464, 623, 639

for loop, 65–73, 76, 77, 80, 84, 85, 90–92, 119, 120, 126–128, 137, 138, 141, 143, 150, 155, 158, 435, 537
forEach loop, 427, 435, 436, 438, 440, 446, 447, 455, 464, 465, 623, 639, 646, 649, 652
vs streams, 643
while loop, 65, 66, 77–80, 83, 84, 90, 92, 115, 154, 192, 441, 540, 666, 667
Iterator interface, 440
Iterator objects, 427, 440, 441

J
Java Archive (JAR) files
executable, 563, 575, 639
Java API, *see* Application Programming Interface
Java byte code, 5, 8, 16, 555
Java Collections Framework (JCF), 428, 463
Java Database Connectivity (JDBC), 553, 564
Java Development Kit (JDK), 6, 165, 317
Javadoc
@author tag, 317, 318
@param tag, 318, 330
@return tag, 318
@throws tag, 421, 422, 611, 614, 620, 635, 639
@version tag, 317, 318, 344, 609–611, 614, 620, 635
JavaFX
containers, 265
init method, 587
Scene class, 274
Stage class, 271, 507
start method, 268, 271, 274, 301, 336–340, 517, 567, 582, 585, 587, 671, 680
stop method, 268, 587, 682
Java Runtime Environment (JRE), 6, 578
Java Virtual Machine (JVM), 5, 16
JVM, *see* Java Virtual Machine

L
lambda expressions, 265, 279, 280, 284, 287, 341, 358, 368–371, 373–375, 378, 379, 384, 387, 389, 391, 412, 413, 435, 436, 442, 443, 461, 473, 475, 477, 482, 582, 646, 649, 652–655, 657
layout policies
BorderLayout, 292
FlowLayout, 292
lazy evaluation, 641, 642, 659
length attribute–arrays, 151, 156, 198, 200–202
lightweight components, 266
lightweight process, 578
List interface, 383, 428–431, 456, 460
logical operators, 51, 52
loops
do...while loop, 65, 66, 79–84, 90, 92
enhanced **for** loop, 119, 138, 139, 141, 143, 158, 187, 190, 316, 331, 427, 434–436, 438–440, 446, 447, 464, 623, 639
for loop, 65–73, 76, 77, 80, 84, 85, 90–92, 119, 120, 126–128, 137, 138, 141, 143, 150, 155, 158, 435, 439, 446, 535
picking the right loop, 83
while loop, 65, 66, 77–80, 83, 84, 90, 92, 115, 154, 192, 441, 540, 666, 667

M
main method, 12, 29, 35, 36, 41, 95, 97–99, 103, 107, 112, 116, 117, 129, 133, 136, 161, 162, 165, 172, 185, 197, 207, 208, 215, 253, 268–270, 274, 311, 337, 374, 399, 402, 561, 562
Map interface, 383, 428, 443
math package, 380, 575
MediaPlayer class, 488
menu-driven programs, 112, 117, 219, 258, 264, 531
menus
popup, 505
pull-down, 499, 500, 503
method
abstract, 253, 254, 268, 359–361, 369–371, 374, 378, 383–385, 412, 413, 482, 588, 649, 692, 693
actual parameters, 95, 101, 110, 116
calling, 95, 98, 100, 105, 132, 165, 191, 227, 278, 312, 401, 413, 416, 621, 680
class, 205, 208, 229, 238, 396, 411, 412, 456, 457
constructor, 167
declaring and defining, 96
formal parameters, 95, 100, 101, 103, 107, 108, 202
header, 13, 97, 134, 224, 371, 397, 398, 400, 402, 403, 430, 584
helper, 212, 268, 348, 382, 383, 517, 518, 520, 532, 542, 595, 598, 621, 671, 680
overloading, 109, 110, 116, 235, 245, 249, 261, 385, 390
overriding, 235, 245, 246, 248, 249, 261, 385, 390, 692, 694, 697
return value, 102, 103, 129, 186, 318
method references, 374, 461, 462

modal dialogues, 499, 505
modifier, 205, 235, 257, 261, 263, 560, 584, 697
modulus operator, 26, 27, 72, 105
multiple inheritance
  diamond problem, 692, 693
multitasking, 577, 578, 641, 658
mutual exclusion threads, 584
MySQL™ database, 564, 569

N
Nested statements, 53, 55
Netbeans™, 7, 8, 564, 569
network programming
  client-server architecture, 686
  remote databases, 642
non-modal dialogues, 524
null value, 171, 213, 216, 232, 316, 379, 408–410, 412, 413, 416, 445, 446, 660
number formatting, 297

O
object encoding, 527, 549, 551
Object class, 235, 257, 258, 262, 449–451, 585, 697, 698, 701, 702
ObjectInputStream class, 542
object-orientation
  benefits of, 223, 235
object-oriented programming languages, 236, 358
ObjectOutputStream class, 542–544
object serialization, 542
operators
  arithmetic, 25
  concatenation, 3, 15, 30
  increment/decrement, 29
  logical, 51, 52
  overloading, 385, 390
Optional class, 393, 410–413, 423
Oracle™, 707
out of the box interfaces, *see* generics
output
  devices, 527, 528, 549
  stream, 529, 530, 549, 642, 665–667, 670, 672, 682
  to file, 337, 534, 563
  to screen, 9, 14, 15, 528, 535
overloading method, 109, 110, 116, 235, 245, 249, 261, 385, 390
overriding method, 697

P
package(s)
  accessing, 555
  Java API, 369
  deploying, 563
  developing, 558, 559, 609
  hierarchy, 554, 555
  scope, 559, 560, 574
parallelism - streams, 658
parameters
  actual, 95, 101, 110, 116
  arrays, 129, 134, 189
  formal, 95, 100–103, 107, 108, 110, 116, 202
  objects, 195, 209, 482, 562
  varargs, 131–134, 138
parameterized types, 376
platform independence, 5
pointers, 293, 531, 545, 548, 689–691, 694
polymorphic types, 357, 385
polymorphism, 95, 110, 116, 245, 246, 357, 385, 386, 390
popup menus, 505
Predicate interface, 378, 648
PrintWriter class, 535
primitive types, 19–21, 30, 176, 185, 190, 191, 259, 376, 452, 545
private, 195, 197, 200, 204, 205, 212, 215, 238, 239, 246, 247, 262, 337, 338, 341, 606, 620, 637, 696–698, 700
processes concurrent, 578, 582
producer–consumer relationship, 584
program design, 35, 311
programming languages, 3–5, 20, 110, 163, 223, 236, 358, 705, 706
protected, 238, 239, 246, 262, 697
pseudocode, 19, 35, 37, 38, 43, 78, 95, 104, 140, 141, 143, 308, 316, 331, 335, 341–344, 354, 538
public, 11–13, 195, 197, 200, 201, 205, 238, 262, 315, 328, 559, 560, 606, 612, 614, 621, 693, 694, 697
pull-down menus, 499, 500

Q
quantum, 582

R
radio buttons, 499, 509, 511–513, 525
random access files, 527, 544, 545, 550
record data, 567
reference, 121, 122, 130, 138, 167, 168, 171, 185–187, 205, 210, 211, 255, 256, 336, 374, 375, 379, 408, 430, 488, 556, 558, 574, 588, 593, 595, 674, 677, 681, 683, 687, 690, 695–701, 707
repetition, *see* iteration

ResultSet class, 567, 568
Runnable interface, 383, 580, 582
run-time error, 564

S

scalar types, *see* primitive types
Scanner class, 19, 31, 32, 165, 168, 169, 175, 179, 209, 395, 396, 424, 529, 556–558
scenario testing, 625
scheduling algorithms, 582
scheduling thread, 582
scope, *see* variable scope
ScrollPane class, 485
selection, 41–43, 49, 54, 59, 65, 67, 106, 338, 642
sequence, 12, 41, 42, 50, 59, 65, 173, 269, 428, 529, 600, 658
serial access files, 527, 531, 550
Serializable interface, 543, 544, 550, 551
serialization object, 542
ServerSocket class, 661, 663, 665
Service class, 577, 590
Set interface, 428, 436–438
short type, 20
sleep method, 585–587
sliders, 479, 481, 482, 497
sockets, 661–663, 673
Socket class, 663, 665, 666, 672, 686
software
  application software, 3, 4, 16
  embedded software, 4, 16
  systems software, 3, 4, 16, 528
sort methods
  in the Arrays class, 456
  in the Collections class, 456
source code, 3–5, 7, 16, 230, 550, 561
specification, 227, 307, 349, 603, 604, 606, 624
SQL, *see* Structured Query Language
sql package, 567
StackPane, 291, 303, 513
standard error stream, 529
standard input stream, 529
standard output stream, 529
stateful operations - streams, 641, 657, 659
stateless operations - streams, 658
state transition diagram threads, 577, 585
static attribute, 206, 207, 676
static, 11, 13, 97, 100, 195, 205–209, 269, 328, 336, 360, 361, 378, 461, 463, 488, 575, 586, 656, 676
static method, 360, 461
streams
  collecting results, 654
  concatenating, 654, 656
  creating, 645, 646
  infinite, 656
  intermediate operations, 642, 645, 648, 658
  parallelism, 658
  terminating operations, 652
  vs iteration, 643
Stream API, 641
stream - files, 530
string(s)
  comparing, 176, 445
  methods, 163, 174, 176
  using with switch statements, 610
String class, 163, 172–174, 176, 177, 385, 433, 445, 458, 544
Structured Query Language (SQL), 565, 568, 572, 573, 642, 706
student hostel case study, *see* case study
subclass, 237–240, 246, 249, 254, 256, 257, 359, 361, 385, 386, 395, 398, 400
Sun™, 6
super, 239, 244, 694
supplier interface, 378
superclass, 237–239, 244, 248, 249, 251, 254, 257, 359, 380, 386, 420, 449, 691
system software, 3, 4, 16, 528
Swing, 266, 292
switch statement
  with enumerated types, 610, 615, 622
  with strings, 174
synchronizing threads, 584
syntax, 5, 17, 32, 35, 77, 79, 131, 132, 161, 274, 357, 369, 372, 401, 403, 412, 417, 491, 690

T

TabPane class, 603, 626
Task class, 577, 587
termination operations - streams, 645
testing, 217, 223, 253, 258, 308, 310–312, 317, 320, 321, 328, 330, 335, 350, 354, 357, 421, 449, 603, 604, 624, 625, 639
test log, 307, 321, 323, 324, 328, 334, 350, 354
text encoding, 530, 551
text files, 531
text formatting, 539
TextInputDialog class, 348, 669, 671, 680
this, 135, 136, 170
Thread class, 580, 582
threads

asynchronous, 577, 599
execution, 577, 582, 584, 585, 599
lightweight process, 578
mutual exclusion, 577, 584
scheduling, 582
states, 585
state transition diagram, 577, 585
synchronizing, 584
throwing exceptions, 401, 417, 424
throw command, 393, 417
throws clause, 393, 398
time-slicing, 577, 578, 582, 599
tool tips GUI, 635
`toString` method, 258, 311–313, 316, 317, 321, 324, 328, 334, 403, 431, 446, 447, 449, 453, 457, 464, 465, 610, 622, 643, 703, 704
try…catch block, 401, 567
try-with-resources construct, 393, 423, 549, 623, 624
two-dimensional arrays
creating, 148
initializing, 149
ragged, 155
`txt` package, 297, 344, 555
type casting, 242, 244, 583
type inference, 189, 370, 377, 429, 482

U
UML, *see* Unified Modeling Language
`Unaryoperator` interface, 656
unboxing, 259, 463
unchecked exceptions, 395, 422
Unicode, 20, 122, 178, 208, 396, 452, 477, 530, 539, 542, 583
Unified Modeling Language (UML), 195–197, 199–202, 206, 211, 212, 223, 224, 228, 230, 231, 235, 237, 239, 246, 247, 260, 261, 307, 308, 310, 328, 330, 331, 334, 336, 357, 360, 386, 448, 454, 455, 464, 466, 467, 603, 604, 606–608, 638, 700
unit testing, 307, 308, 311, 320, 624
upper bound, *see* generics
use-case model, 603, 604
user-defined constructor, 237
user interface, 9, 285, 304, 335, 353, 525, 587, 603, 626
`util` package, 31, 47, 429, 430, 456, 553, 556, 557, 575, 606

V
validation
input, 65, 77, 78, 92, 153, 154, 287, 350, 531
varargs, 131–134, 138, 159, 161
variable(s)
declaring, 21, 23, 33, 35, 100, 119, 167, 205
environment, 562, 574
local, 107, 108, 114, 200, 208, 244, 364, 371
scope, 107
`Vbox` class, 267, 278, 279, 284, 288, 289, 292, 295, 299, 340, 503, 513, 568, 627, 628

W
`WebView` class, 482, 489–491
wild card, *see* generics
while loop, 79–83, 90
wrapper classes, 190, 259, 287, 376, 389, 463

X
XML, 570, 572, 707