5-3-2019

Introduction to Computer Science with Java Programming

Seth D. Bergmann
Rowan University, bergmann@rowan.edu

Introduction to Computer Science with Java Programming

Seth D. Bergmann

May 1, 2019

Preface

This book is intended to be used for a first course in computer programming. No prior experience with programming should be necessary in order to use this book. But this book is intended to be used with a course that teaches more than computer programming; it is intended to be used with a course that teaches Computer Science. The distinction is subtle, but important.

The author(s) believe that a breadth-first approach is the best way to introduce the concepts of Computer Science to students. Rather than isolate topics in courses (bits and bytes in a computer organization course; formal grammars and languages in a theory course; lists, sets, and maps in a data structurs course; etc) we believe that topics should be introduced in a brief and simple manner at the starting level. Elaboration on these topics should occur in subsequent courses. This breadth-first approach allows the student to build on existing knowledge and retain a greater proportion of the material.

Our colleagues in the physical sciences have done this for over a century: Physics I, Physics II, Physics III; Chemistry I, Chemistry II, Chemistry III.

Some examples of this breadth-first approach to Computer Science:

- We teach the rudiments of binary numbers. Is this necessary to learn to program the solution to a simple problem? Probably not, but it will become necessary at some later time, when studying hardware, or the limitations of software.

- Every introductory programming book teaches the concept of an arithmetic expression. We give a formal definition, and show the structure of an expression by placing boxes around sub-expressions. The student thinks we are teaching how to write a correct expression; we are actually teaching recursion, formal grammars, and derivation trees.

- We advise the student, "Program to an interface whenever possible". The student thinks we are teaching the correct usage of Java interfaces, but we are actually teaching object-oriented design and software engineering.

The student thinks we are teaching programming, but we are actually teaching Computer Science. Knowledge is not separated into compartments, and our curriculum should not attempt to do so.

Several topics were recently addes to this book, in order that it be used as the primary textbook for the Educational Testing Service's Advanced Placement course in Computer Science (CSA):

- More extensive discussion of arrays
- sorting
- binary search

The AP course includes a *Marine Biology* simulation case study which is not included in this version of the book.[1] The reader will notice an empty section for this case study in several chapters of this book. The author invites current instructors to participate in this project by providing these sections; those who do so would be listed as a *secondary author* or *contributor* on the title page and/or the preface, depending on the level of contribution. Those who are interested in contributing should contact the primary author at the email address shown below.

This book is an open source book. That means that not only is the pdf version available (to potential students and teachers) for free download, but that the original (LaTeX) source files are also available (to potential authors and contributors). Based on the model of open source software, open source for textbooks is a relatively new paradigm in which many authors and contributors can cooperate to produce a high quality product, for no compensation. For details on the rationale of this new paradigm, and citations for other open source textbooks, see the journal *Publishing Research Quarterly*, Vol. 30, No. 1, March 2014. The source materials and pdf files of this book are licensed with the Creative Commons NonCommercial license, which means that they may be freely used, copied, or modified, but not for financial gain.

This book is available in pdf at `rdw.rowan.edu /oer`.

The source files are available at `cs.rowan.edu/~bergmann/books`.

The author may be reached at `bergmann@rowan.edu`

Secondary Authors

Contributors

[1]The Marine Biology case study is no longer included on the AP exam. Thus, students using this book are not penalized on the exam for not having seen the case study.

Contents

Chapter 0

Computers and Computer Programs

The first digital computers were developed in the early 1950's. Since then the capabilities of computers (and computer science) have increased at an amazing rate. It has been said that if the automotive industry had seen similar progress, that a Rolls Royce would get 3,000 miles per gallon, would have a top speed approaching the speed of light, and would cost less than one dollar. Various monikers for this amazing development include "The Digital Revolution" and the "Information Age". When we use the word *digital* we are talking about information which is composed of discrete atomic (i.e. having no sub-cmponents) values, generally described as 0 or 1. Anything which is said to be digital, at its innermost level, is nothing but lots of 0's and 1's. This includes not only computers but telephones, cameras, music boxes, televisions, and the list goes on. Today there are digital components in automobiles, household appliances, roads, bridges, medical devices, medicines, and even people.

This book is an introduction to some of the things we have learned about developing computer software. Software is the driving force in a computer system; without software, computer hardware is not capable of carrying out the simplest of tasks. In the process of learning to program a computer in a popular language known as Java, we hope to expose many of the concepts and principles which apply to the development of software in any language.

Computer Science has been defined as "the design, analysis, and implementation of algorithms", and in this book we introduce the notion of *algorithm* in the hope that this will spark the student's interest for further study of Computer Science.

0.1 The CPU and machine language

A computer system is made up of hardware and software. The hardware consists of the physical components:

- Semiconductor chips (CPU, memory, communications, etc)
- Wires and other conductors connecting the components
- Storage devices such as flash memory and disks
- Peripheral devices such as keyboards, mice, displays

The most fundamental hardware component is the *Central Processing Unit* or CPU. This is the component which is capable of performing calculations and making decisions. The CPU is capable of executing only instructions which are coded in a binary format (consisting of only 0's and 1's). These binary instructions, when stored in the computer's memory, constitue a *program*. The language of these binary instructions is called *machine language*

Consequently, if we write a 'program' in Java, it is not a program in the strict sense of the word, because it is not written in machine language.

0.2 High level languages and compilers

If we were to write our programs in machine language, the rate of software development would be slow; the binary coded instructions of machine language make it exceedingly difficult for use by humans. For this reason we have developed *high-level languages*, also known as *programming languages* which are much easier to use for programming than machine language. Some examples of high-level languages are:

- Java
- C++
- C
- Visual Basic
- Python
- Ruby

However, the CPU is not capable of executing the statements of a high-level language directly, so it must first be translated into machine language. This is done by a program known as a *compiler*. The compiler will examine the statements in a program, check for syntax errors, and produce output consisting of binary machine language instructions which the CPU is capable of executing.

In later chapters we speak of 'compile time' versus 'run time'. An error which is detected by the compiler is a *compile-time* error, whereas an error which occurs when the machine language program is executing is a *run-time* error.

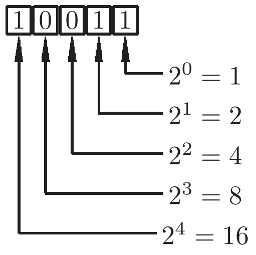


Figure 1: The binary representation of 19 (19 = 16 + 2 + 1)

0.3 Data representation: bits and bytes

As noted above all digital devices, including computers, store information in binary (0's and 1's). Each such binary digit is called a *bit*. This means that in order for us to represent information, everything must be encoded in binary; this includes not only numbers, but characters from the keyboard, sound, images, colors, video clips, ... everything. In this section we offer some insight as to how this is done. Further details on these data representation schemes are given in later chapters.

0.3.1 Whole numbers

To represent a whole number we use base two. Whereas in a base ten (decimal) number each digit represents a power of 10, in base two each digit represents a power of 2, as shown in Fig 1 which shows the base two representation of 19.

Whole numbers may also be negative; to represent a negative whole number we use *two's complement representation*. In this scheme numbers, positive or negative, can be easily added, always producing the correct result. We describe two's complement in more detail in chapter 2

0.3.2 Other numbers

Numbers which are not whole numbers (or are too big to be stored as simple binary values) are stored in floating point format. This means that associated with each number is an exponent to magnify (or diminish) the value. This kind of number is similar to scientific notation: $6.02x10^{23}$

0.3.3 Characters

Any character from the keyboard (and others) can be represented with a binary code. This includes the letters (a..z, A..Z) the numbers (0..9) and other characters such as `$!@#%^&`. A binary code is assigned to each character. A 16-bit code with characters from international alphabets is called Unicode. An older code, using an 8-bit (byte) code called ASCII is a subcode of Unicode.

0.3.4 Images

A digital image is like a newspaper photograph, which consists of a matrix of discrete dots. Each such dot is called a *pixel*, or picture element. For color images each pixel is simply a whole number representing the red, green, and blue components of the desired color.

0.3.5 Sound

Sound is made up of air pressure waves; when these waves enter our ears, our brain receives a signal from the auditory nerves. To represent sound, all we need do is store a digital version of the pressure waves. This is diagrammed in Fig 2, in which the horizontal axis is time, and the vertical axis is the amplitude of the pressure waves. The wave at top left of Fig 2 represents just one cycle of a sound wave. The wave at top right represents a louder sound at the same pitch because the amplitude is greater, but the frequency of the cycles is the same. The wave at bottom left represents a higher pitch, because there are twice as many cycles in the same time period; the loudness is the same as the sound at top left. The quality of the sound is determined by the number of values sampled per unit time. To represent sound with very high quality (high fidelity) requires many numbers. Consequently a sound clip is merely a sequence of whole numbers representing varying air pressure. Most sound clips are actually a compressed format of these numbers. Some examples of compression formats are .wav and .mp3.

0.3.6 Exercises

1. Show the following decimal numbers in base two (binary):
   (a) 7
   (b) 23
   (c) 123
   (d) 127
   (e) 255
   (f) 256
2. Read parts (a) and (b) below aloud so that they make sense.
   (a) There are 10 kinds of people in the world: those who know binary and those who do not know binary.
   (b) There are 10 kinds of people in the world: those who know base three, those who do not know base three, and those who have no idea what I am talking about.
   (c) Make up a similar statement for some number base greater than three but less than 10.

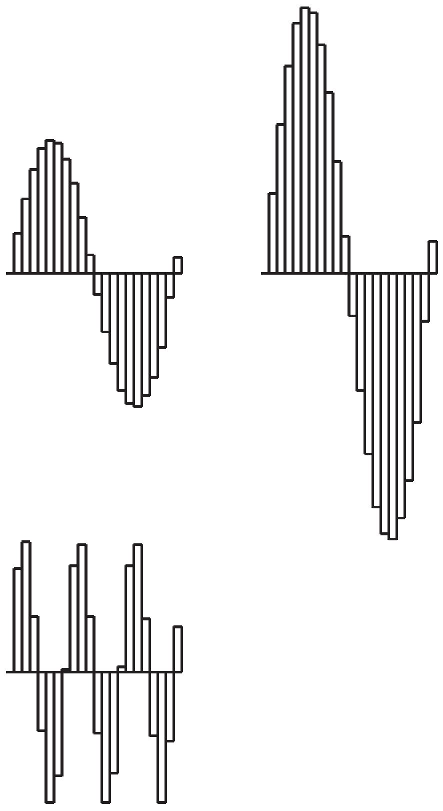


Figure 2: Representing sound (i.e. air pressure waves) as discrete numbers. Top right: a louder sound. Bottom left: a higher pitch.

3. Show your friends how to count from 0 to 31 using only the fingers on one hand (tell them not to be offended when you get to 4).

4. Do an internet search to find the ASCII code for each of the following characters:

   (a) 'a'

   (b) 'A'

   (c) '8'

   (d) '('

5. How many pixels are there in a 8x8 square inch display which has 256 pixels per inch? Hint: Use powers of 2. $256 = 2^8$.

6. Do an internet search to find sound data compression formats other than .wav and .mp3.

Chapter 1

Java classes, objects, object diagrams and methods

1.1 Classes and objects

Computer programs usually deal with concepts and problems selected from our common environment and/or experiences. In order to represent these concepts in software, the Java language introduces the concept of *class*. A class is like a template, or an archtect's blueprint; it allows one to specify the attributes and behavior of objects. A class is the plan from which objects can be created. Hence, an *object* is merely an instance of a class.

For example, we could have a class named `Student` which specifies all the attributes of a student. These attributes could be things such as the student's name, social security number, and gpa. A diagram for a particular Student object is shown in Figure 1.1.

The values of the object's attributes are collectively known as the *state* of the object.

Classes can also specify behavior of the objects. For example:

- We may ask a Student object to provide us with his/her name, gpa, or social security number.

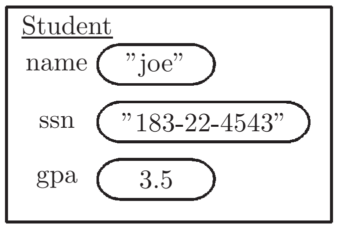


Figure 1.1: An object diagram showing an instance of the class `Student`

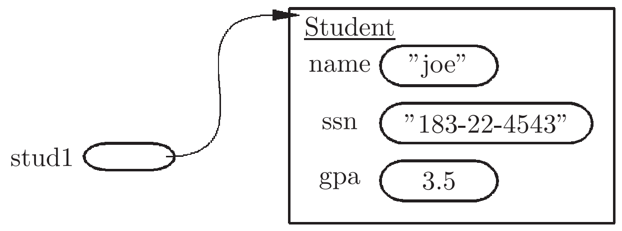


Figure 1.2: An object diagram showing the value of the variable `stud1` as a reference to an object

- We may wish to change a Student's gpa.
- We may allow a Student object to register for courses (which could entail including a list of those courses as another attribute for Students), and do any number of things that students in the real world normally do.

This behavior is specified with *methods* (more on this later). The compiler will permit a class name to begin with a lowercase letter, however we will follow the convention that all class names must begin with an *uppercase* letter.

1.1.1 Exercises

1. Show an object diagram for a Student whose name is "mary", and whose ssn is "999-99-9999" and whose gpa is 3.8.
2. Assume there is a class named `University` which stores a name and a size (number of students enrolled). Draw an object diagram showing an instance of this class; make up values for the state of this object.

1.2 Variables and references

A Java program may have many variables, and as we'll see later, there are various kinds of variables. For now, we'll define a *variable* as a symbol which can store a value. An example would be the symbol `gpa` in the Student class. This kind of variable is called an *instance variable*, or *field*, because it is part of an object which is an instance of a class. Figure 1.1 shows that the value of the variable `gpa` is 3.5.

Variables can also store references to objects. A *reference* is merely an indication of where an object can be found in the computer's memory. Figure 1.2 shows a variable named `stud1` which stores a reference to a Student object. The compiler will permit a variable name to begin with an uppercase letter; however we will follow the convention that all variable names begin with a *lowercase* letter.

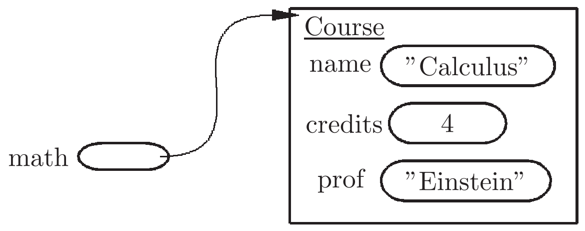


Figure 1.3: An object diagram

1.2.1 Exercises

1. Refer to Figure 1.3.
   (a) What is the name of the object's *class*?
   (b) What are the names of the *fields* in the object?
   (c) A *reference* to the object is stored in which variable?

2. Show an object diagram for a variable named `univ` which stores a reference to a University object whose name is "Southern State" and whose size is 12000 students.

1.3 Defining a Java class

Figure 1.2 is an example of an *object diagram*. It shows that a variable named `stud1` stores a reference to an object which is an instance of the class `Student`. It also shows all the fields (i.e. instance variables) and their values. Note that the value of a variable is always shown in a rounded rectangle, whereas objects are shown in ordinary rectangles.

The value of the variable `stud1` in Figure 1.2 is a reference to a Student object. This value is depicted with an arrow in the diagram. More accurately, the reference is a memory address, or location of the object in memory.

In a Java program, we can specify the name of a class, and its fields, as shown in Figure 1.4. This class definition begins with the key words `public class`, and contains the field definitions within a set of curly braces. We note that:

- The word `public` is used to indicate that the class being defined can be accessed from any other class. The word `class` is used to indicate that what follows is the definition of a class. The words `public` and `class` are *key words*, which means that they can be used only for these purposes (i.e. they cannot be used as variable names).
- After the name of the class there is an open brace.
- This is followed by a definition of each field in the class.

```
public class Student
{  // fields
   private String name;
   private String ssn;
   private double gpa;
}
```

Figure 1.4: Java code defining a class named `Student` which has three fields

- Each field has a type (such as `String` or `double` - more on this later).
- The class is *public*, but the fields are *private*. This is called *visibility* (more on this later).
- The class definition ends with a closed brace.

We also note that:

- Java is *free format*. This means that we are free to include spaces and new lines anywhere in a java program. We will generally try to include spaces so as to make the program easy to read. Conceivably, Figure 1.4 could have been written entirely on one or two lines:

  ```
  public class Student {private String name;private
  String ssn;private double gpa;}
  ```

  Though the Java compiler would allow this, it is not considered good style, and we will avoid it.
- Java is *case sensitive*. This means that you must pay attention to upper case versus lower case letters. The word `class` is different from the word `Class`.

1.3.1 Exercises

1. Show the Java code to define a class named `University`; it should have two fields: `name`, which is a String, and `size`, which is an int (i.e. integer).
2. Refer to Fig 1.4.
   (a) How many fields are in the class defined as Student?
   (b) What is the *type* of the field `ssn`?
   (c) What is the *type* of the field `gpa`?
3. Each class definition shown below contains, at most, one error; find the error and correct it if there is one.

(a)
```
public Class Car
{  private double cost;
   private String make;
   private String model;
}
```

(b)
```
public class Vehicle
{  private double
   cost; private
   String make; private String model; }
```

(c)
```
public class Vehicle
   private double cost;
   private String make;
   private String model;
   private int wheels;
```

4. Show the Java code to define a class named `Ticket` with three fields:

   - A section (type is String)
   - A row number (type is int)
   - A seat number (type is int)

1.4 Object diagrams

This book will make extensive use of object diagrams, similar to the one shown in Figure 1.2. The concept of an object diagram is one that is vital to a good understanding of Java programs and more advanced concepts in computer science, such as data structures.

In an object diagram a reference (i.e. an arrow) will *always* refer to an object, and *never* to another variable. As we will see later, the value of an object's field may be a reference to another object.

1.5 Methods

The fields of a class specify the attributes, or collectively, the state of an object of that class. Objects can also have behavior; the behavior of an object is specified in the class with *methods*, or more properly *instance methods*[1].

A method consists of a signature and a body. The *signature* defines how the method is to be invoked, and the body defines exactly what it does. An example of a method signature in the Student class could be

[1]C++ programmers would call these *member functions*.

```
public String getName()
```

- The visibility of this method is `public`. Methods may also be `private` - more on this later.
- This method will *return* a String.
- The name of this method is `getName`. The compiler will permit a method name to begin with an uppercase letter; however, we will follow the convetion that every method name begins with a *lowercase* letter.
- This method has no *parameters*, though the parentheses are always required in the signature.

The method *body* consists of one or more Java statements enclosed in curly braces. In the `getName` method, the body could be:

```
{  return name;  }
```

The *return* statement defines what the method produces as a result when it is invoked. It also terminates the method.

One more important feature (but not required by the compiler) is a group of *comments* which help to describe the purpose and correct usage of the method. These comments are ignored by the compiler, but can be very useful to us humans as we attempt to define and use methods. These comments begin with /** and end with */ . Comments of this form are used by a utility program, `javadoc` to generate nicely formatted documentation for a class and its methods. We call this documentation an Application Program Interface, or API. The complete method definition, with comments, is shown below:

```
/** This method returns the name of this Student. */
public String getName()
{
   return name;
}
```

Methods do not have to return any value (these are called *void* methods. Also a method may have one or more *parameters*, specified inside the parentheses in the signature. These are used to pass information into the method when it is invoked.

```
/** This method changes the name of this Student. */
public void setName(String newName)
{
   name = newName;
}
```

The `setName` method has one parameter, whose value is the new name for a Student. It is a `void` method because it produces no explicit result.

We can now show the class definition, with fields and methods, in Figure 1.5

```
public class Student
{  // fields
   private String name;
   private String ssn;
   private double gpa;

   /** This method returns the name of this Student. */
   public String getName()
   {  return name;  }

   /** This method changes the name of this Student. */
   public void setName (String newName)
   {  name = newName;  }

}
```

Figure 1.5: Java code defining a class named `Student` which has three fields and two methods

1.5.1 Exercises

1. Refer to the following class defining a `University`

   ```
   public class Unviersity
   {  private String name;
      private int size;
   }
   ```

   (a) Include a method definition in this class to return the University's name.

   (b) Include a method definition in this class to change the size of the University to a given nuumber of students.

2. Refer to Figure 1.4. Each of the following method definitions contains, at most, one error. Find and correct the error if there is one.

   (a)
   ```
   public getGPA()
   {  return gpa;  }
   ```

   (b)
   ```
   public void clearGPA()
   {  gpa = 0.0;  }
   ```

   (c)
   ```
   public void setGPA(newGPA)
   {  gpa = newGPA;  }
   ```

1.6 Constructors and object creation

In this section we discuss how objects can be created. Java has an operator called *new* whose sole job is to create new objects. When the `new` operator is invoked, in the process of creating the object, a *constructor* is called to initialize the fields in the object. The constructor is a peculiar kind of method with the following properties:

- The name of the constructor is the same as the name of its class.
- The constructor has no return type *not even void*.

The student object shown in Figure 1.1 can be created as shown below:

```
new Student ("joe", "183-22-4543");
```

The values ”joe” and ”183-22-4543” are *actual parameters* for the constructor. They are the initial values of two of the fields in the Student object being created.

The constructor, defined as one of the methods in the Student class, is shown below:

```
/** Constructor.
    Initialize this Student's name, ssn, and gpa
 */
public Student (String initialName, String initialSSN)
{  name = initialName;
   ssn = initialSSN;
   gpa = 0.0;
}
```

Note that when creating a new Student, no initial gpa is provided. Instead the constructor initializes gpa to 0.0 for all new students.

The parameters in the constructor could have been the same as the fields which they are initializing. In this case we need to distinguish between the field and the parameter using the key word *this* as shown below:

```
/** Constructor.
    Initialize this Student's name, ssn, and gpa
 */
public Student (String name, String ssn)
{  this.name = name;     // assign parameter value to field
   this.ssn =  ssn;      // assign parameter value to field
   gpa = 0.0;
}
```

We now have three kinds of entities in a class definition:

1. private field(s)

```
public class Student
{  // fields
   private String name;
   private String ssn;
   private double gpa;

   /** Constructor.
       Initialize this Student's name, ssn, and gpa
    */
   public Student (String initialName, String initialSSN)
   {  name = initialName;
      ssn  = initialSSN;
      gpa  = 0.0;
   }

   /** This method returns the name of this Student. */
   public String getName()
   {  return name;  }

   /** This method changes the name of this Student. */
   public void setName (String newName)
   {  name = newName;  }

}
```

Figure 1.6: Java code defining a class named `Student` which has three fields, one constructor, and two methods

2. public constructor(s)
3. public and/or private method(s)

These entities can be placed in the class definition in any order, but we usually conform to the order shown above. Our updated class definition is now shown in Figure 1.6.

1.6.1 Exercises

1. Show a *constructor* for the `University` class which will initialize the `name` and `size` fields from constructor parameters (see Exercises above).
2. Each of the following constructor definitions for the Student class contains, at most, one error. Correct each error, if there is one.

   (a)
   ```
   public void Student ()
   {  name = "joe";
   ```

```
   ssn = "222";
}
```

(b)
```
public StudentConstructor ()
{  name = "joe";
   ssn = "222";
}
```

(c)
```
public Student (String name, String ssn)
{  name = name ;
   ssn = ssn  ;
}
```

3. Show a constructor for the Student class with three parameters. The parameter names should be `name`, `ssn`, and `gpa`. Each parameter should be used to initialize the corresponding field.

1.7 Getting started: IDE or command line

Having seen some of the rudiments of defining a class, we can now see how to work with these concepts on the computer. There are two basic ways to create and test Java programs:

- Use an *Integrated Development Environment* or IDE.
- Run the java compiler and its runtime environment from the command line prompt. This is covered in chapter 9.

Most beginning users will prefer to use a simple IDE, such as BlueJ, but other IDEs are available for free download on the internet.

1.7.1 Using an IDE

Some examples of IDEs which can be used to develop Java software are:

- BlueJ - A fairly simple, yet powerful, IDE which can be used to edit and execute Java source files, and inspect objects. It also allows the user to test code snippets to find out what effect they have. BlueJ is available for free download at `www.BlueJ.org`.
- NetBeans - This IDE began as a student project in Czechoslavakia and was later acquired by Sun Microsystems (now Oracle). Like BlueJ it is designed specifically for Java development. It differs from BlueJ in that it has tools specifically for the automatic creation of graphical user interfaces.
- Eclipse - This IDE has so many features that it is rather difficult to learn. The main advantage of Eclipse is that it can be used for many different programming languages, not just Java. Once you learn Eclipse, you can use it on many different projects. Another interesting feature of Eclipse is that the debugger can *step backwards*.

All three of these IDEs are free for download on the Internet; in addition, they are all open source and have many optional plug-in features.

1.7.2 The BlueJ IDE

Since BlueJ is the easiest IDE for novices to learn, it is the one we choose to look at here. After learning BlueJ, many users will opt for a more powerful IDE at a later time.

1.7.2.1 Getting started with BlueJ

The basic development unit in BlueJ is the *project*. A project may consist of several related classes (a project is really a folder, or directory, containing a source file and related files for each class in the project). When starting up BlueJ, use the menu to open a new project, or to open an existing project. Be sure to save the project to a disk or storage device for which you have write access. As you make changes to your source files, BlueJ will automatically save the project.

To create a new class for your project, click the button `New Class...`. This will bring up a dialog box allowing you to enter the name of the class (the name should begin with an uppercase letter) such as *Student*. There will now be a file with suffix `.java` in your project folder, such as `Student.java`.

An example of a BlueJ project window is shown in Figure 1.7 in which the project has three classes, and one Student object has been created.

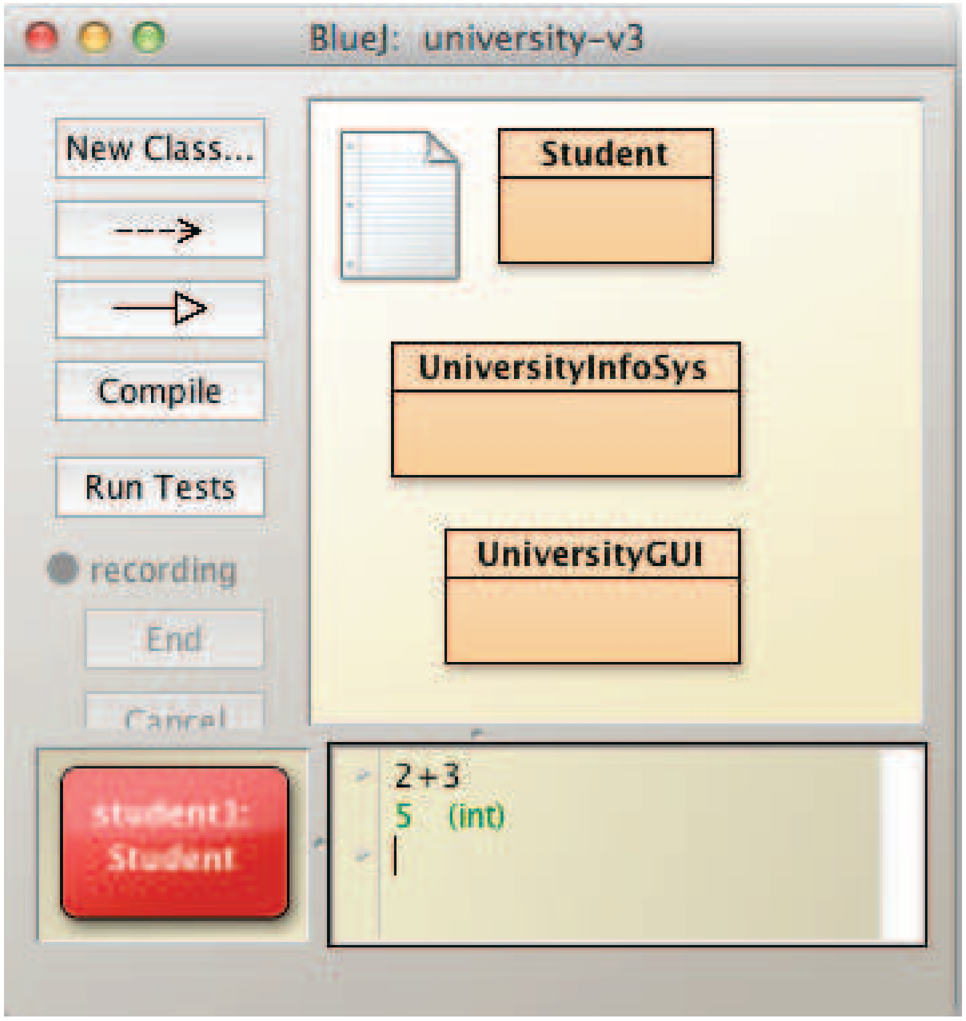
To edit a class, double click on the (tan) icon for the class. This brings up the BlueJ editor window, showing the lines of Java code in the class. BlueJ will provide a template of sorts for you to get started; you will want to modify or delete most of what BlueJ has provided. Note that Java source files are plain text documents (like Notepad documents), and the BlueJ editor is nothing more than a text editor (like Notepad for Windows or TextEdit for MacOS).

To test your work two steps are needed:

1. Compile - Translate your source code to 'byte code', a language that can be understood by the Java runtime environment. If there are syntactic errors in your class, the compiler will advise you of these. You must correct these errors before going further. BlueJ will draw hashmarks on classes which need to be compiled. If there are no syntactic errors, the compiler will create the byte code file, with a `.class` suffix, such as `Student.class`. Do not try to examine a `.class` file; it will be unintelligible and is strictly for Java's use.

2. Test - Execute the byte code by invoking a method in one of your classes. Right-click on a class:

    - If the class has a static method, select it to run it directly from the selected class. If that method has parameter(s), a dialog box will be opened allowing you to enter the parameter value(s).



```
2+3
5   (int)
```

Figure 1.7: An example of a BlueJ project with three classes. One object of class Student has been instantiated. The CodePad shows that the value of the expression 2+3 is 5

- Otherwise, instantiate a class (i.e. create an object of the class) by selecting `new ....` If the class' constructor has parameters, a dialog box will allow you to enter the parameter value(s) at this point (caution: remember the double quote-marks if the parameter is a String). The object should appear as a red box in the object window at the lower left. Right-click on the red object to inspect it (look at the values of its fields) or to invoke an instance method. If the method has parameters, a dialog box will allow you to enter values for the parameters (caution: remember the double quote-marks if the parameter is a String).

1.7.2.2 The BlueJ Terminal Window

If an executing method produces output, it will be sent to the BlueJ Terminal Window. Some options available for the terminal window include:

- Unlimited Buffering. The window acquires a scroll bar on the right, and will retain an unlimited amount of output. Use the scroll bar to view any output.

- Clear. Clear the output window so that you do not confuse the output of multiple executions.

- Clear Screen at Method Call. Clear the screen automatically each time BlueJ starts up a method. This is often the preferred option.

- Record Method Calls. Show all methods which have been invoked from BlueJ.

- Save to file... Open a dialog box to allow all output to be saved in a text file.

1.7.2.3 The BlueJ Debugger

A *debugger* is a tool designed to help the programmer locate the source of a logic error in the program. A debugger will not tell you where to find a bug, nor will it correct the mistake for you; this is the programmer's job. With a debugger, the programmer can step throught the statements of a program, one statement at a time, while watching the values of variable change. This kind of tool is often essential in diagnosing an error. Debuggers are used at runtime, not at compile time.

To start up the debugger, open a BlueJ Editor window, and click the mouse in the left margin (on a line number). You should see a red stop sign appear; this is known as a break point. When execution reaches this point, it will pause and wait for you to direct the debugger to continue execution, either single-step, or at full speed.

More details on the use of the BlueJ debugger are provided in chapter 8.

1.7.2.4 The BlueJ Codepad and inspections

The BlueJ Codepad is in the lower right corner of the project window. The Codepad can be used to execute small code snippets to see how they work. BlueJ will invoke the compiler and runtime environment to evaluate expressions as you enter them in the Codepad. Figure 1.7 shows the expression `2 + 3` in the Codepad, along with its result, `5`. It is possible to instantiate classes in the Codepad and inspect the objects which are created.

Any object on the object bench (lower left corner) can be inspected by right-clicking on the object and selecting `inspect`. You will see the values of all fields; if any are references to objects, you can select the reference and inspect the object to which it refers.

1.7.3 Exercises

1. What is the full name of the source file for the University class?
2. What will be the name of the output file when compiling the University class?
3. True or False: If the compiler produces no error messages, your program must be correct.
4. Which of the IDEs mentioned in this section have a debugger?

1.8 Constructors and objects in the GridWorld case study

1.9 Projects

1. Define a class named Course which is to store information on a university course. Each Course object should store:

   - A course title (a String)
   - The number of credits for the course (an int)
   - The name of the the prof teaching the course (a String)
   - Course level – `"Grad"` or `"Undergrad"`

   Your class should have public methods which provide access to each of the fields and public methods which allow changes to each of the fields. Your class should have two constructors, one with three parameters and one with four parameters; the constructor with three parameters should assume the course is an undergrad course. Each constructor should initialize all four fields.

   Test your solution:

(a) Compile the class; it should compile without error messages

(b) Instantiate the Student class (i.e. create a Student object) using the constructor with three parameters.

(c) Instantiate the Student class (i.e. create a Student object) using the constructor with four parameters.

(d) Inspect both objects (if your IDE will permit this) and make sure the values of the fields are correct.

Chapter 2

Program Elements and Methods (revisited)

2.1 Data types

Java permits several kinds of data for use in a program. As described in chapter 1, each data value can be represented by a sequence of bits (0's and 1's). In this section we will explore some of the primitive data types available. In each case we will take a quick peek at its binary representation.

All data types can be classified as either *primitive* or *reference* types (reference types are also known as *object* types). An easy way to distinguish these is that all pirmitive types begin with a lower-case letter, whereas reference types begin with an upper case letter (because they are class names). Examples of data types are shown in Figure 2.1 In this section we explore some of the primitive types, and a few reference types.

Primitive types	Reference types
int	String
float	Student
double	System
char	
boolean	
byte	
short	
long	

Figure 2.1: Examples of data types. All primitive types begin with a lower-case letter. All reference types begin with an upper-case letter.

2.1.1 Whole numbers: int

One of the simplest primitive data types, *int*, is used to represent (positive or negative) whole numbers. Possible values that can be represented by an int are 124, 0, -9833, and 248888. Note that 2.43, 0.05, and even 3.0 are *not* ints. int values do not contain a decimal point.

Figure 2.2 shows how the values -8 through +7 can be represented using only 4 binary digits. (These are the only values that can be represented with 4 bits, because there are $2^4 = 16$ different patterns of four bits.) This is called *twos complement* representation. Note that:

- There are more negative numbers than there are positive numbers.
- All negative values begin with a 1.
- If a value begins with 0, it may or may not be positive.
- Odd numbers end with a 1, and even numbers end with a 0.
- -1 is represented by all ones. This makes sense; consider an automobile odometer of a new car (circa 1970), initially at 000000. If you back up the car one mile, the odometer will read 999999.
- You can add and subtract values, and throw away the bit carried out of the high order digit.

```
   0101 = +5
  +1110 = -2
------------
   0011 = +3
```

In Java, the int data type is actually 32 bits in length, which places an upper and lower bound on the magnitude of an int.

2.1.2 Other numbers: float and double

In programs where we need to work with numbers other than whole numbers, or with numbers that are very large or very close to zero, there are two data types available: *float* and *double*. Examples of floats (actually doubles) are: 2.43, 0.001, -34334.1, and 3.0

With these data types we must include the decimal point. It is also possible to specify very large numbers and numbers that are very close to zero, using a notation similar to the scientific notation used in your science classes. A value can include an exponent of 10, written after an e or E:

2.04e5 = $2.04x10^5$ = 204000

6.02e23 = $6.02x10^{23}$ (a very large number)

int value	binary
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
+1	0001
+2	0010
+3	0011
+4	0100
+5	0101
+6	0110
+7	0111

Figure 2.2: Representation of int values using a 4-bit word.

-1.0e-53 = $-1.0x10^{-53}$ (very close to zero)
24.03E-2 = $24.03x10^{-2}$ = 0.2403
Note that:

- Each value consists of two parts, which we call the *mantissa* and *exponent*.
- These two parts are separated by an e or E.
- The mantissa and exponent can each be positive or negative.
- The exponent implies an exponent of 10.
- If the exponent is positive, slide the decimal point to the right the given number of places.
- If the exponent is negative, slide the decimal point to the left the given number of places.

The data type *double* is the same as float, but it permits more precision. We will generally use double rather than float, simply because the cost (in memory or processing time) is not excessive. Constant values, such as 1.3e-2, are actually stored as doubles.

2.1.3 Logical values: boolean

Perhaps the simplest primitive data type is *boolean*. There are only two boolean values: `true` and `false`.

This data type will be used extensively in chapters 3 and 4.

char value	binary	decimal
'A'	01000001	65
'B'	01000010	66
'C'	01000011	67
'Z'	01011010	90
'a'	01100001	97
'b'	01100010	98
'z'	01111010	122
'0'	00110000	48
'1'	00110001	49
'2'	00110010	50
'9'	00111001	57
'\$'	00100100	36
'\#'	00100011	35

Figure 2.3: ASCII codes for some common characters.

2.1.4 Character values: char

We will need to work with data other than numbers in our programs, specifically data made up of alphabetic characters and other characters (such as $, %, +, ...). Any character which can be typed on the keyboard (and other characters) can be represented by the data type *char*. Examples of chars are 'a', 'T', '8', '*', '!'. Note that each value is enclosed in *single* quote marks. Figure 2.3 shows how these data values are represented in binary using an 8-bit code known as ASCII (American Standard Code for Information Interchange). When it became necessary to accommodate other alphabets, a 16-bit code, known as Unicode was introduced. ASCII is a sub-code of Unicode.

Note in Figure 2.3 that:

- The codes for numbers are smaller than the codes for upper-case letters, which in turn are smaller than the codes for lower-case letters.
- The codes for upper-case and for lower-case letters are contiguous, i.e. 'a' is 'b'-1, and 'b' is 'c'-1, etc.
- If you subtract the code for a '0' from the code of any numeric digit, you get that digit's int value: '7' - '0' = 55 - 48 = 7

2.1.5 Strings of characters: String

Normally when working with character data, we wish to group several characters into a single entity, known as a *String*. Examples of String data are "joe", "dataSet314", and "3a09d&#9aw". Note that the characters of a String are enclosed in *double* quote marks. Each String may contain any number of characters from the keyboard.

```
String          length
"joe j"         5
"jim"           3
"jo"            2
"j"             1
""              0

'j'             This is a char,  not a String
'jo'            This is a mistake, a char must consist of
                exactly one character
```

Figure 2.4: Examples of Strings and chars, showing the length of each String

Character to be included	Escape sequence	Example
double-quote	`\"`	`"Title: \"Pygmalion\", Shaw"`
newline	`\n`	`"line break \n here"`
tab	`\t`	`"\tcol1 \tcol2 \tcol3"`
backslash	`\\`	`"\\root\\directory\\filename"`

Figure 2.5: Escape sequences are used to include special characters in a String

The `String` data type is a reference type, not a primitive type (it begins with an upper-case letter). As shown in Figure 2.4:

- each String has a length
- a space character counts as one of the characters in a String
- the length of a String can be 0

How can we include the double-quote character in a String constant? We use what is called an *escape* sequence, using the backslash character:

```
"She said \"hi\" to me"
```

The escape chracter can also be used to quote a newline character (`'\n'`) or a tab character (`'\t'`) as shown in Fig 2.5

Finally, we should be able to include the backslash in a String.

2.1.6 Other reference types

Because we have defined a class named `Student`, this may also be used as a data type (it is a reference type). Thus there can be any number of data types, corresponding to classes we have defined. Moreover, we have access to thousands of classes that others have defined and can use them as data types. These classes are available in what is known as the *Java class library*, and `String` is just one example of the classes available there.

2.1.7 Exercises

1. Having defined `Student` as a class, can `Student` be used as a type? If so, would it be a primitive type or a reference type?

2. Refer to Figure 2.2.

   (a) Show a similar table for 5-bit words. (Hint: the value of zero is 00000).

   (b) How many different numbers can be represented with 5 bits?

   (c) What is the largest positive number that can be represented with 5 bits?

   (d) What is the smallest negative number that can be represented with 5 bits?

3. Complete the following table:

word size (bits)	number of different values	largest positive value	smallest negative value
4	16	7	-8
5	?	?	?
8	?	?	?
16	?	?	?
32	?	?	?
64	?	?	?
n	?	?	?

   Hint: Sometimes it is easier to show a number as a power of 2.

4. What are the *primitive types* representing whole numbers which correspond to the word sizes 8, 16, 32, 64, respectively? Hint: There are 8 bits in a byte.

5. Write Avogadro's number as a Java constant.

6. Arrange the following constants in order from smallest to largest: 99.0, 1405.3e-12, 1e2, -999.3e45

7. What is the length of each of the following Strings?

   (a) `"elephant"`

   (b) `"my small cat"`

   (c) `"$*&#@!("`

   (d) `"x"`

   (e) `""`

8. What is the value of each of the following?

Operator	Meaning	Example	Result
$+$	Addition	$7+4$	11
$-$	Subtraction	$7-4$	3
$*$	Multiplication	$7*4$	28
$/$	Division (quotient)	$7/4$	1
%	Mod (remainder)	7%4	3

Figure 2.6: Common arithmetic operations

(a) '8' - '3'

(b) '8' - '0'

(c) '7' - '0'

(d) '6' - '0'

(e) 'f' - 'b'

(f) 'a' - 'z'

2.2 Operations and expressions

Java statements make extensive use of operations (such as add and subtract) and expressions (such as $(a + b) - 3$). But there are many more operations available. This section will introduce some of the more common operations.

2.2.1 Arithmetic operations

The most common arithmetic operations are listed in Figure 2.6.

As an example of an operation, $3 + 4$ would compute the value 7. If either, or both, operands of an operation are floating point (i.e. `float` or `double`), then a floating point operation is performed, producing a floating point result, not an int result. In many cases this will appear to have no effect. For example, $3 + 4$ produces the `int` result 7, but $3.0 + 4$ produces the `double` result 7.0.

This behavior is most evident with division. Division of ints must produce an int result, with no decimal places. Therefore, $7/4$ produces the `int` result 1, not 2, nor 1.75. Decimal places are truncated.

The *modulus* (or *mod*) operation, designated, perhaps surprisingly, by the % operator, has nothing to do with percentages. It should be applied to ints only, and the operation $a\%b$ produces as a result the `int` remainder when a is divided by b. Mathematically, integer division has two results: a quotient (obtained by the / operator, and a remainder (obtained by the % operator).

Examples of division and mod operations are shown in Figure 2.7.

2.2.2 String operations

Strings of characters can also be manipulated with operations. The simplest String operation is called *concatenation* and is designated with a + operator.

Example	Result	type of result	remarks
15/8	1	int	decimal places are truncated
15/8.0	1.875	double	at least one operand is double
15.0/8	1.875	double	at least one operand is double
15%8	7	int	remainder after division
0%5	0	int	
1%5	1	int	
2%5	2	int	
3%5	3	int	
4%5	4	int	
5%5	0	int	
6%5	1	int	
7%5	2	int	
10%5	0	int	mod serves as a circular counter

Figure 2.7: Some examples of division and mod operations

The + operator in this case does not mean addition; it means concatenation. When an operator can take on different meanings, depending on its operands, we say that the operator is *overloaded*. The arithmetic operators were already overloaded because the operation could be either an int operation or a floating point operation, depending on the types of the operands.

In the case of strings, concatenation allows us to form a new String from two smaller strings:

- "john" + "son" produces the result "johnson"
- "no" + "thing" produces the result "nothing"
- "no" + " thing" produces the result "no thing"
- "john" + " " produces the result "john "
- "john" + "" produces the result "john"
- "%f*!3" + "321" produces the result "%f*!3321"

Recall that a space character counts as a character in a String, just like any other character that you find on the keyboard. Also the String "" represents a String whose length is 0 (sometimes called a *null* String, though we will not do so to avoid confusion with the null reference)

When one of the operands of the + operator is a String, but the other operand is not a String, the non-String operand is automatically converted to a new String, which is then used as the operand for the concatenation:

- 23 + "skiddoo" produces the result "23skiddoo"
- "skiddoo" + 23 produces the result "skiddoo23"

- "17" + 23 produces the result "1723"
- 17 + "23" produces the result "1723"
- 17 + 23 produces the result 40

The String class is part of the Java Class Library, and it provides us with a multitude of features with which we can manipulate strings. To see the complete documentation for the String class, point your web browser to a current version of the API for the Java class library (`docs.oracle.com/javase/7/docs/api`). We now give a few of the more useful operations on Strings. These are all methods in the String class, and can be called by attaching them to any String (which we call *this* String).

- `int length()` - returns the length of this String as an int.
- `char charAt(int index)` - returns the char at the given position of this String. The first character is at position 0.
- `int indexOf(String str)` - returns the position of the first occurrence of `str` in this String, or -1 if not found.
- `String substring(int begin, int end)` - returns a part of this String, beginning with the character at position `begin`, and ending with the character at position `end` - 1.
- `String substring(int begin)` - returns a part of this String, beginning with the character at position `begin`, and and continuing to the end.
- `String toUpperCase()` - returns a new String in which all lower-case letters of this String have been converted to upper-case.
- `int compareTo(String otherString)` - returns a positive int if this String follows `otherString` alphabetically, returns a negative int if this String precedes `otherString` alphabetically, and returns 0 if the this String is equal to `otherString`.

Examples of these methods are shown in Figure 2.8.

2.2.3 Arithmetic Expressions

Several operations may be combined into a single *expression* using parentheses as needed to indicate the order of operations. Some examples of expressions are shown in Figure 2.9.

The concept of *expression* is so fundamental to java programming, that we give a more formal and precise definition for expressions involving ints here, in which *expression* is abbreviated as *expr*:

An expr may be a:

1. number

Method signature	Example	Result
`int length()`	`"and how".length()`	`7`
`char charAt(int index)`	`" and how".charAt(5)`	`'h'`
`int indexOf(String str)`	`"jonson".indexOf("on")`	`1`
	`"jonson".indexOf("on.")`	`-1`
`String substring(int begin, int end)`	`"and how".substring(4,7)`	`"how"`
`String substring(int begin)`	`"and how".substring(2)`	`"d how"`
`String toUpperCase()`	`"AnD HoW23!!".toUpperCase()`	`"AND HOW23!!"`
`int compareTo`		
`(String otherString)`	`"john".compareTo("sam")`	a positive int
	`"john".compareTo("johnson")`	a negative int
	`"john".compareTo("john")`	`0`
	`"John".compareTo("john")`	a negative int
	`"99".compareTo("100")`	a positive int

Figure 2.8: Examples of some String operations

Expresssion	Value
$4 * 7 + 3$	31
$(4 * 7) + 3$	31
$4 * (7 + 3)$	40
$((7 - 2) * (7 + 3))/4$	12
$((7 - 2.0) * (7 + 3))/4$	12.5

Figure 2.9: Examples of arithmetic expressions

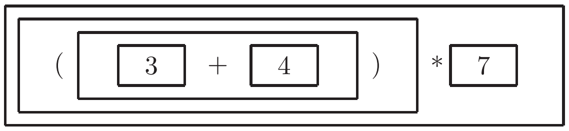


Figure 2.10: Applying the definition of expression to $(3 + 4) * 7$

2. expr + expr
3. expr - expr
4. expr * expr
5. expr / expr
6. expr % expr
7. (expr)

We now suggest an exercise in which we apply the definition given above, to determine whether a piece of code constitues a valid expression. We apply the rules of the definition, one at a time, by putting a box around each expression (or subexpression) noting that nothing is an expression until there is a box around it. Figure 2.10 shows how the definition can be applied to the expression $(3 + 4) * 7$. Note that each box represents a subexpression, and shows which rule of the definition is applied.

Note that our definition of *expr* involves the word *expr*. We call this a *recursive* definition, and this is valid and legitimate as long as:

- At least one rule does not involve the usage of *expr*. This is called the *base case*.
- Rules that use the word *expr* contain other text as well. I.e. a rule should not define an *expr* to be merely an *expr*.

It is interesting to note that any precise definition of expression *must be recursive*. This results from the fact that an expression is inherently recursive. If you continue to study Computer Science, you will learn the importance of recursion.

A better version of Figure 2.10 is shown in Figure 2.11. Here we show the rule number from the definition of expression each time a rule is applied with a box. For example, rule 2 of our definition is:

2. expr + expr

Thus, in Figure 2.11 there is a small number 2 in the upper right corner of the subexpression, $3 + 4$.

Another example is shown in Figure 2.12, in this case for the expression $3+4*7$. Here we note that there is an alternate solution, shown in Figure 2.13.

This is a serious problem. There appear to be two different ways of applying the rules of our definition. The solution shown in Figure 2.12 suggests that $3+4$

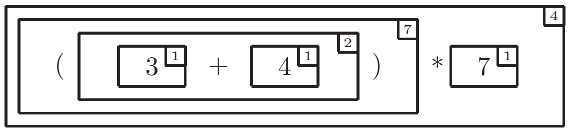


Figure 2.11: Include rule numbers from the definition of expression when applying the definition to $(3 + 4) * 7$

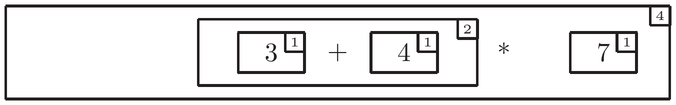


Figure 2.12: Applying the definition of expression to $3 + 4 * 7$

is a subexpression, and that therefore the addition is done before the multiplication. On the other hand, Figure 2.13 suggests that $4 * 7$ is a subexpression, and that therefore the multiplication is done before the addition.

These two solutions produce different results for the same expression, 49 in the first case and 31 in the second case. This is not good; an expression must have a single consistent value. The reason for this problem is that our original definition is *ambiguous*. This means that it allows for different interpretations of the same expression.

One way to resolve this ambiguity is with *precedence rules*:

- Multiplication, division, and mod always take precedence (are performed first) over addition and subtraction. For example $3 + 4 * 7 = 3 + (4 * 7)$, as shown in Figure 2.12. If one wishes to do the addition first, one must use parentheses: $(3+4)*7$. We now see that Figure 2.12 does not suggest a true interpretation of the given expression, but that Figure 2.13 does suggest a true interpretation.

- If there is more than one operation at the same level of precedence, they are executed left-most first. For example, $9 - 4 - 2 = (9 - 4) - 2$ and $12/2/3/2 = ((12/2)/3)/2$

2.2.4 Exercises

1. Show the *value* and *type* of each of the following expressions.

   (a) `2 * 8.0`

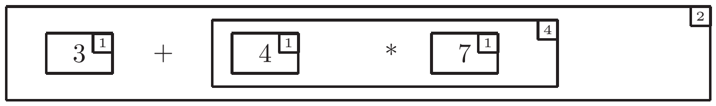


Figure 2.13: An alternate solution when applying the definition of expression to $3 + 4 * 7$

(b) `17 / 5`

(c) `17 % 5`

(d) `17 / 5.0`

(e) `234884173 % 10`

(f) `"some" + "body"`

(g) `("some" + "body").length()`

(h) `"somebody".charAt(2)`

(i) `"SomeBody".toUpperCase()`

(j) `"somebody".indexOf ("me")`

(k) `"somebody".indexOf ("Me")`

2. Find the *value* of each arithmetic expression shown below. They should all be ints.

   (a) `4 + 2 * 3`

   (b) `(4 + 2) * 3`

   (c) `8 / 2 / 3`

   (d) `(9999 / 10000) * (192 - 383)`

   (e) `94528 % 5 + 9 / 10`

3. Draw boxes around each of the *expressions* in the previous problem, as shown in Figure 2.11. Show the rule number applied in each box, and be sure to obey the Java precedence rules.

2.3 Declaration and initialization of variables

In Java a *variable* may be used to store, or remember, a value. A variable may be a single letter or it may consist of many letters, underscore characters, and/or numeric digits, but it must begin with a letter. Some examples of variables are x, total, birthDay2014, sum_total. Note that:

- These are NOT strings (they are not in double quote marks).
- As usual, these are case sensitive, so the variable $sumOfProducts$ is different from the variable $sumofproducts$.
- The compiler will permit a variable to begin with an upper-case letter, but we will not do this (only class names should begin with an upper-case letter).

2.3.1 Declaration of variables

Before a variable can be used, its type must be declared:
    $type\ variable - list$;
For example, the declarations:

```
int x, sum, result;
String name;
```

mean that the three variables `x`, `sum`, and `result` all will be used to store int values, and the variable `name` will be used to store a (reference to a) String.

2.3.2 Iniitialization of variables

Before a variable can be used in an expression it must have a value. We can give a value to a variable in the same statement which declares the variable. This is known as *initialization*, and the format is:
    $type\ \ variable = expression$;
Examples of variable initializations are shown below:

```
int x=0;
double sum=0.0, tolerance=0.0001;
boolean done=false, ok;
String name = "joe";
```

The above gives initial values to the variables `x`, `sum`, `tolerance`, `done`, and `name` but it leaves the variable `ok` uninitialized.

2.3.3 Exercises

1. Which of the following are *not valid names* for variables?
   (a) flummox33foo22
   (b) 3x
   (c) var(ok)
2. Show how the variables `salary`, `tax`, and `fica` can all be declared to be of type `double` and initialized to the values 99,000, 345.53, and 150, respectively, all in one statement.

2.4 Assignment of values to variables

Variables are used to store, or remember, values. A value can be assigned to a variable using the *assignment* operator, =, as shown below:
    $variable = expression$
For example, $result = 3 + 4 * 7$ means that the value 31 will be stored in the variable *result*. The assignment operator can be summarized:

1. The expression on the right side of the operator is evaluated.

2. The value of the expression is stored in the variable.

3. The type of the expression should match the type of the variable. If they are of different types, in some cases the expression can be converted to the appropriate type (as described in the next section).

The = operator does *NOT* mean *equals* and should not be read that way. We suggest reading it as 'is assigned the value of' or 'gets the value of'. Here are a few examples to clarify the requirement that types match:

```
int x = 8;
String city = "Boise";
x = x + city.length();     // x is now 13
city = "New York";
x = city;                  // ERROR - types do not agree
city = 18;                 // ERROR - types do not agree
18 = x;                    // ERROR - left operand must
                           //         be a variable
```

We can now add another rule to our definition of *expression*:

An *expr* may be: 8. *variable*.

This means that we can include $a + b * 28$ as a valid expression. When it is evaluated, the *current* values of a and b are used. (Values should have been assigned to these variables before attempting to use this expression)

We are now able to form an executable Java statement. A Java statement may be:

$variable = expression$;

Java statements are written sequentially, and are executed in the order in which they are written. We will have a more extensive definition of *statement* in chapter 3. Figure 2.14 shows an example of a series of statements, along with the values stored in variables as the statements are executed. Note that:

- The value of a variable can change as the statements are executed.

- When an expression involving variables is evaluated, the *current* value of each variable is used.

- The statement $x = y$; does **NOT** imply a comparison of the value of x with the value of y. It means find the value of y and store that value into the variable x.

Students are often confused by the sequential nature of expressions and statements, but careful attention to these examples should help alleviate the confusion.

Program code	Value of a	Value of b	Value of c
int a,b,c;			
a = 3;	3		
b = 2;	3	2	
c = a+b*2	3	2	7
a = b;	2	2	7
c = (a+b)*5	2	2	20
b = 8;	2	8	20
b = a;	2	2	20
b = b+1;	2	3	20

Figure 2.14: Examples of assignment statements, and their effect on variables

2.4.1 Type conversion in assignments

An int may be assigned to a float or double variable:

```
int total, count;
double average;

total = 10;
count = 3;
average = total / count;
```

In this example, the division produces an `int` result, 3, which is then converted to `double` 3.0 when assigned to the variable average.

The compiler will not allow a float or double to be assigned to an int:
total = 3.0; The compiler will produce an error message: "Possible loss of precision" even though the value being assigned is a whole number.

The programmer can assure the compiler that this possible loss of precision is acceptable by using a *cast*, in which case the decimal places are truncated:
total = (*int*)3.9;
The variable *total* is assigned the `int` value 3

A `char` may also be casted to yield its numeric ASCII code:

```
    char ch = 'x';
    int code = (char) ch;      // code is 120
```

2.4.2 Type conversions and initializations

At this point we caution the student to be careful when initializing variables; consider the following example:

`double x = 5 / 3;`

The intent here is to initialize the variable `x` with the value 1.6666666667, the result of the division. However this is not what happens. The division will be an `int` division because both of its operands are ints, producing an `int`

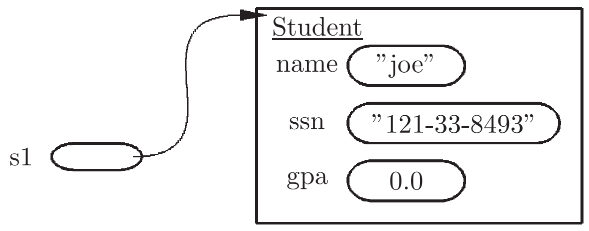


Figure 2.15: An object diagram showing a variable storing a reference to a Student object

result: 1 which is then converted to 1.0 when assigned to the variable `x`. There is an important lesson to be learned here: the components of a statement are executed separately and sequentially; they are not executed 'all at once'.

2.4.3 Assignment of references

The assignment operator will always evaluate the expression on the right side, and assign that value to the variable on the left side. This is straightforward for variables of primitive type. However, for variables of reference type we need to take a careful look at the behavior of the assignment operator.

Recall that a variable of reference type does not store the actual data; rather it stores a *reference* to the data. For example, the declaration

```
int gpa;
```

means that `gpa` stores an int; however the declaration

```
Student s1;
```

does *not* mean that `s1` stores a Student. Rather, it means that `s1` stores a *reference* to a Student (initially a null reference). We can change that reference as shown below: `s1 = new Student ("joe", "121-33-8493");`
Figure 2.15 shows the object diagram which would result. The reference stored in the variable `s1` is actually a memory location for the Student object to which it refers, and this is depicted in the diagram with an arrow.

Suppose we declare another Student variable:

```
Student s2;
```

Now, however, instead of creating a new Student object, we use the assignment operator thus:

```
s2 = s1;
```

This simply means to copy the reference which is in `s1` into `s2`. Consequently the variables `s1` and `s2` will be storing the same reference, and will refer to the same object, as shown in Figure 2.16 .

This becomes interesting if we were to make a change to the data:

```
s1.setName("jim");
```

changing the name of the Student. The result is shown in Figure 2.17 . We then access the name of Student s2:

```
System.out.println ("The name of s2 is " + s2.getName());
```

which will print "jim". Since `s1` and `s2` refer to the same object, a change to

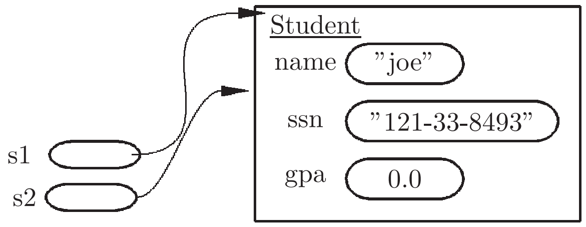


Figure 2.16: An object diagram showing the effect of assignment of a reference to a variable: `s1 = s2;`

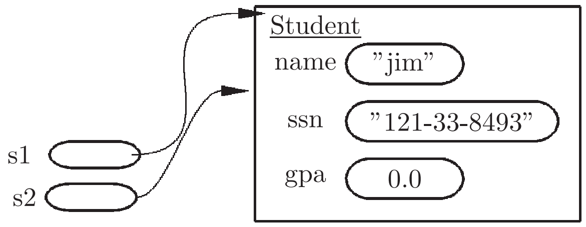


Figure 2.17: An object diagram showing the effect of a change to the variable `s1`. The name of `s2` is now `"jim"`.

the object referred to by `s1` will also change the object referred to by `s2` even though we never made an explicit change to the variable `s2`.

2.4.4 Exercises

1. Show the final *values* of the variables `x,y,z` after the code shown below has executed.

```
int x = 3,y,z;
y = x + 2;
z = y;
x = 14;
y = y + 1;
```

2. Show the final *values* of the variables `x` and `i` after the code shown below has executed.

```
int i;
double x;
i = 4;
x = i;
i = i + 1;
```

3. Which of the statements shown below will cause a *syntax error* from the compiler?

```
int i = 7;
double x = 2.5;
double y = 2.0;
i = x;
i = y;
x = i;
```

4. Show the final *values* of the variables `i,j` and `y` after the code shown below has executed.

```
int i = 7;
double x = 2.99, y = 11 / 12 * 2;
i = (int) x;
j = ((int) ((3.5 / 4) * 4)) % 7
```

5. Refer to the University class introduced in the exercises from chapter 1.

```
University u1, u2, u3;
u1 = new University ("Slippery Rock",1000);
u2 = u1;
u2.setSize(900)
System.out.println ("Size of u1 is " + u1.getSize());
```

   (a) What will be *printed* by the code shown here?
   (b) Draw an *object diagram* showing the values of u1, u2, and u3 after the code shown here is executed.

2.5 Method definitions, signatures, and invocation

We introduced the notion of a Java *method* in chapter 1. Methods, more properly known as *instance methods*, are used to define the behavior of objects in a class.

2.5.1 Method definition

A method definition consists of an *API*, a *signature* and a *body*.

The API (Application Program Interface) is ignored by the compiler, but is very useful to the programmer. It defines the purpose of the method, what it expects from the calling method (preconditions), and what it produces as a result, or what changes it makes to objects (postconditions). Hence the API provides all information needed for someone to write a call to the method. The API begins with $/**$ and ends with $*/$. This API is processed by a utility program known as *javadoc* which produces html for a nice looking description of the method's purpose, preconditions, and postconditions in the form of a web page.

```
/** Calculate this student's gpa.
 *  Precondition: Both parameters are positive.
 *  Postcondition: This student's gpa is set
 *                 as the result of dividing
 *                 the grade points by the credits
 */
public void calculateGPA(int gradePoints, int credits)
{  double creditsAsDouble;
   creditsAsDouble = credits;
   gpa = gradePoints / creditsAsDouble;
}
```

Figure 2.18: Method which calculates, and sets, this Student's gpa, given total number of grade points and total number of credits

2.5.2 Method signature and body

The method signature consists of:

1. Visibility, i.e. `public` or `private`: Private methods can be invoked only from another method in the same class, whereas public methods may be invoked from a method in any class.

2. Return type: This is the type of the explicit result of the method. In this case we think of the method as being similar to a mathematical function which produces a single result. If the method produces no result, the return type is `void`.

3. Method name: The name of the method, as with variables, may consist of 1 or more alphabetic and/or numeric characters, but it must begin with an alphabetic character. Note that the method name is case sensitive.

4. Parameter list: 0 or more variables, with types specified, separated by commas. The list must be enclosed in parentheses, even if there are no parameters. We call these parameters *formal parameters* to distinguish from *actual* parameters, described below. Parameters are used to pass data into a method.

The method body consists of Java statements, such as assignment statements, all enclosed in one set of curly braces.

Figure 2.18 shows an example of a method which could be included in the `Student` class. The purpose of this method is to calculate, and change, the student's gpa, given the total number of grade points, and the total number of credits.

Note that we do not wish to do a floating point division in this method, so we assign the `int` value of `credits` to a `double` variable, `creditsAsDouble`. Then the division is a floating point division, producing a floating point result.

2.5.3 Method invocation

How and when are methods executed? Generally, they are called, or invoked, from another method. In the method call, the name of the method being called is attached to an object on which the method is being invoked. Also, actual values of parameters are included in the parameter list. These *actual parameters* may be constants, variables, or more complex expressions, but their types *must* correspond to the types of the corresponding formal parameters in the method signature.

As an example, we could invoke the calculateGPA method in the Student class from a method in some other class as shown below:

```
Student s1;
int calculus, comp, stats;
calculus = 3;
comp = 4;
stats = 2;

s1 = new Student ("jim", "240-33-4321");

s1.calculateGPA (calculus*4 + comp*3 + stats*2,
                 4 + 3 + 3);
```

When a method is invoked:

1. The actual parameters, which are expressions, are evaluated, and these *values* are copied to the corresponding formal parameters in the method definition.
2. The statements in the method being invoked are executed in sequential order.
3. When the final statement has been executed (or a `return` statement is executed), control returns to the calling method.

Be sure that actual parameters in a method call correspond in number and type with formal parameters in the definition of the method being called. Figure 2.19 illustrates some valid and non-valid method calls.

Also note that the following code is not correct:

```
Student s2;
s2.calculateGPA(20,6);
```

When the variable `s2` is declared, its value is `null`, a reference which refers to nothing. A Student has not been instantiated, and no valid value has been assigned to the variable `s2`. This will produce a run-time error called a `nullPointerException`, [1] meaning that your program will come to a crashing

[1]It is unfortunate that this exception is named *nullPointerException* in the java class library, rather than *nullReferenceException* Other programming languages use pointers which are similar to references, but it is possible to do arithmetic with pointers.

Method signature	Method invocation	Remarks
public void meth(int a, char b)	s1.meth(3);	Incorrect: there are two formal parameters and only two actual parameters
	s1.meth(3, 4);	Incorrect: The type of the second actual parameter must be char, to agree with the second formal parameter
	s1.meth(3,'b');	ok
	x = s1.meth(3,'b');	Incorrect: The method is a void method and has no explicit result for assignment to x.
public int evenOdd (int a)	int result;	
	result = evenOdd(17);	ok
	evenOdd(17);	The compiler will accept this, but it is probably not what is desired; the explicit result is discarded.

Figure 2.19: Examples of correct and incorrect method calls. Actual and formal parameters must have a one-to-one correspondence. Void methods have no explicit result.

halt. This is to be avoided. In general, when your program halts unexpectedly with a `nullPointerException`, check the variable *to the left of the dot*. It should not be null; make sure you have assigned a value to it.

Figure 2.20 shows a method named `getRoundedGPA` which will return the student's gpa, rounded to two decimal places. It may be invoked from another class:

```
double roundedGPA;
Student s1;
s1 = new Student ("jim", "123-33-3222");
...   Statements establishing a GPA for s1, not shown here
roundedGPA = s1.getRoundedGPA();
```

Note that:

- The method `getRoundedGPA` returns an explicit result, which is then stored in the variable `roundedGPA`.
- The method `getRoundedGPA` has no parameters but the parentheses are still needed, both in the method definition and in the method call.
- When the method's `return` statement is executed, the method terminates execution (even if there are more statements after the `return` statement), and the explicit result of the method is available to the calling method.

```
/**  Postcondition: return this Student's gpa,
  *                 rounded to the nearest one
  *                 hundredth.
  */
public double getRoundedGPA()
{  int gpaAsInt;
   gpaAsInt = (int) (gpa * 100 + 0.5);
   return gpaAsInt / 100.0;
}
```

Figure 2.20: Method which returns a Student's gpa, rounded to the nearest hundredth

2.5.4 Methods From the Java Class Library

[2]

There are many methods which are predefined and ready for use, in the Java class library. Here we discuss a few methods in the `Math` class. These methods all happen to be static methods, otherwise known as class methods. To invoke a class method it must be preceded by the name of its class.

To find the absolute value of a number, use the `abs` method. Its parameter may be an `int` or a `double`, in which case it returns an `int` or a `double`, respectively.[3] For example, Math.abs(-3) would return 3, and `Math.abs(12.05)` would return `12.05`.

There is a method in the Math class which raise a number to a given power (i.e. exponent). It's name is `pow` and it is defined for double precision floating point only.[4] For example, `Math.pow(4.0,3)` would return $4.0^3 = 64.0$.

Another method is used to find the (positive) square root of a number. It is abbreviated `sqrt`. For example, `Math.sqrt(16)` would return `4.0` and `Math.sqrt(0.01)` would return `0.1`.

The Math class also has a method which will return a random floating point value. It has no parameters and always returns a random `double` in the range `0.0 .. 1.0`. For example, `Math.random()` might return `0.32803492`. [5]

These methods are summarized in Fig 2.21.

2.5.5 Exercises

1. Describe the *error* in each of the following method signatures:

[2]This section may be omitted without loss of continuity, but note that the methods described here are in the AP subset of the Java language.

[3]Also available are float and long.

[4] If provided with an int or float parameter it will automatically convert the parameter to type `double` and will always return a result of type `double`.

[5]There are other useful random number methods in the `java.util` package (see chapter 5).

Method signature	Example	Result
`int abs(int x)`	`Math.abs(99)`	`99`
`double abs(double x)`	`Math.abs(-9.09)`	`9.09`
`double pow(double base, double exponent)`	`Math.pow(-2.0, 3.0)`	`-8.0`
`double sqrt(double x)`	`Math.sqrt(100.0)`	`10.0`
`double random()`	`Math.random()`	`0.771107`

Figure 2.21: Examples of some class methods from the Math class

(a) `public myMethod ()`

(b) `public int void myMethod()`

(c) `public void myMethod (x, y, z)`

(d) `int myMethod(double x)`

2. Define a *method* for the Student class which will return the student's gpa as a percentage of 100. For example if the student's gpa is 2.5, the result should be `"62.5 %"`. The name of the method should be gpaAsPct, and it should have no parameters (don't forget to include the API).

3. Define a method which could be included in any class to return the *average gpa* of three students. The name of the method should be average3, and there should be three parameters (all students).

4. Consider the following method definitions, which could be included in any class:

```
/** Change the name of the given Student to "jim"
 */
public void meth1(int x, Student s)
{   x = x + 1;
    s.setName ("jim");
}

/** This method is used to expose passing of
    parameters.
 */
public void meth2()
{  int x = 7;
   Student s1 = new Student ("joe", "999-99-9999");
   meth1 (x,s1);
   System.out.println ("x is " + x);
   System.out.println ("name of s1 is " + s1.getName());
}
```

Show the what would be *printed* by a call to `meth2()`

Hints:

- The value of an actual parameter is copied into the corresponding formal parameter. They are separate and distinct variables, even if they have the same name.
- When a parameter is a reference type, the reference, not the object to which it refers, is copied to the actual parameter.

2.6 Recursive methods

Methods can call themselves. These are called *recursive* methods, and the concept is similar to our recursive definition of *expression*. We will take a more careful look at recursive methods in chapter 3.

2.6.1 Exercises

1. In the definition of *expr* given earlier in this chapter, which of the rules do not make use of the word being defined?
2. In the definition of *expr*, which of the rules do make use of the word being defined?

2.7 Printing the output

Up until now we have considered computations that occur when a program executes. At some point, however, we may wish our program to display information for the user. The way this is done can vary considerably. In most modern applications a *graphical user interface* or *GUI* is used to display results in a multitude of different forms. We will introduce GUIs in chapter 10, but for now we will simply display plain text strings for the user to view. Depending on the development environment you are using this could appear in a few different ways:

- If using an IDE such as BlueJ, NetBeans, or Eclipse, the IDE will open a window or pane in which the text is displayed. BlueJ calls this the *terminal window*.
- If running the program from a unix or Windows command line, the text will be displayed in that terminal window.

In any case, the output is produced by a `print` or `println` method in the `System` class. Each of these methods has one parameter - a String. When calling these methods, we must provide a String value. This can be a String constant, a String variable, or any more complex expression which evaluates to a String. Figure 2.22 shows some examples of calls to the print methods.

```
String name = "joe";
int total = 99;
```

Method call	output
`System.out.println ("hello");`	`hello`
`System.out.println (name);`	`joe`
`System.out.println ("hello " + name);`	`hello joe`
`System.out.print ("hello ");`	
`System.out.println (joe);`	`hello joe`
`System.out.println ("The total is " + total);`	`The total is 99`

Figure 2.22: Examples of calls to print methods

Note that the *println* method prints its output on a separate line, whereas the *print* method does not. Students who have programmed in other languages are cautioned that the print methods have just one parameter. Other languages may utilize a sequence of expressions separated by commas, but not Java.

2.7.1 Exercises

1. Show what would be *printed* by the following code segment. Be careful in regard to newlines.

   ```
   int x = 3;
   String str = "foo";
   System.out.print (str + "bar");
   System.out.println ("foobar");
   System.out.println (x + 2 + " is the result.");
   System.out.println ("The result is " + x + 2);
   ```

2.8 Constants and class variables

2.8.1 Constants

There are many cases where we need to use a value in a program repeatedly. For example, in a mathematical application we may wish to use the value of Pi, 3.14159... in many parts of a class or method. It is best to use a named constant in cases like this, as opposed to the actual value. This can be done with the keyword `final`. A variable which is declared to be *final* can be initialized, but can never be changed after its initialization. An example would be:

```
final int PI = 3.14159;
```

whereupon we would then use the variable PI instead of the number 3.14159 throught the scope of that variable. This has a few advantages:

- The program is easier to read and understand; presumably the name of the variable provides a clue as to the meaning or intent of the value being used.

- If the value is incorrect, it needs to be corrected in one place only, the initialization of the constant.

For historic reasons constants are usually written in all upper-case letters, and we will conform to this practice despite the fact that it contradicts our convention that only class names begin with upper-case letters.

2.8.2 Class variables

We've seen that classes can have fields (also known as instance variables). Each object of the class has its own copy of the values for those fields; these make up the *state* of the object. In situations where all objects are to share the same value for a field, we can make use of *class variables.* A class variable can be declared with the `static` keyword. This means that there is only one copy of the variable, shared by all objects of that class. Our Student class could have a class variable to designate the maximum number of credits which can be taken by any Student:

```
public static int maxCredits = 18;
```

This declaration would generally be placed with the other fields. A static field is a class variable, and a non-static field is an instance variable. Since the field maxCredits is public and not final, some other class (such as University) could change its value, in which case all student objects would see the new value of maxCredits.

A public class variable can be accessed from any other class, but the name of its class needs to be specified:

```
System.out.println ("Max credits is " + Student.maxCredits);
```

2.8.3 Class constants

Constants and class variables are most often used together, as shown below:

```
public static final int MAX_CREDITS = 18;
```

The variable `MAX_CREDITS` is both a class variable (static) and a constant (final). This is often referred to as a *class constant.* In this case all Student objects would share it value, which can never be changed during execution of the program.

Many students confuse `static` with `final`, and understandably so, because of the normal usage of 'static' in the English language – "Having no motion; at rest". The Java keyword `static` does NOT mean that the variable cannot be assigned a new value.

There are many class constants in the Java class library, most notably the constant PI in the class Math. We can use it as shown below:

```
double area = Math.PI * radius * radius;
```

2.8.4 Exercises

1. You are given the following class definition of `MyClass` (this class has no constuctors, so the compiler supplies a default constructor which does not initialize any of the fields):

   ```
   public class MyClass
   {  private int x = 3;
      public static int var = 7;
      public final  int VAR = 8;
      public static final int MAX = 99;
   }
   ```

   Which of the following statements in a method of some other class would cause *syntax errors*?

   ```
       MyClass mc = new MyClass();
       mc.x = 4;
       mc.var = 18;
       MyClass.var = 9;
       mc.VAR = 0;
       MyClass.VAR = 9;
       System.out.println (mc.MAX);
       System.out.println (MyClass.MAX);
   ```

2.9 Comments and readability

It will soon become evident that programs can be arbitrarily complex, difficult to read, and difficult to understand completely. Even if a program produces correct output, its value is limited if people, including the original programmer, find it difficult to read, understand, and make modifications. For this reason it is extremely important that the programmer make efforts to explain and clarify all aspects of the program.

2.9.1 Formatting a program

One way of clarifying a program is by formatting it in a readable way. Notice in Figure 2.20 that the statements of the method are placed on separate lines. As far as the compiler is concerned, these statements could be written on 1 line, or any number of lines; Java is free format. However, it is in our best interest to format the program in what we consider to be a readable style. We will discuss this further in chapter 3.

Also notice that the statements in a method are indented. Again, this is not required by the compiler, but we do it anyway to make the program easier to read. In the next few chapters indentation becomes increasingly important.

2.9.2 Comments

Another way of improving the clarity of a program is to provide *comments*. Comments are ignored by the compiler, but are helpful to programmers (including the original programmer) trying to understand the program. Java allows for two kinds of comments: *single-line* comments and *mult-line* comments.

Single-line comments begin with // and end at the end of the line. Some examples of single-line comments are shown below:

```
// Calculate the gpa
gpa = gradePoints / credits;  // This should be a floating point division
```

Note that a single-line comment can stand alone by itself on a line. It can also be tacked on to a statement (or even part of a statement). It ends at the end of the line.

Multi-line comments allow for a single comment to span across several lines. A multi-line comment begins with /* and ends with */ as shown below:

```
/* Calculate the gpa by dividing
   grade points by the total number
   of credits.  Be sure that this
   is a floating point division.
*/
gpa = gradePoints / credits;  /* Do NOT divide by zero! */
```

Be sure to include the */ which terminates the multi-line comment. If this is omitted, the remainder of the program will be viewed as one long comment! It should now be clear that the specifications of pre-conditions and post-conditions (the API) in Figure 2.18 is really a special kind of multi-line comment; it begins with /** instead of just /*.

A good programmer will include a liberal dose of comments throughout a program. Rarely is a software project declared to be finished, complete, and fixed. Rather, there will always be corrections, enhancements, extensions, etc. Whether these modifications are to be made by the original programmer or by a maintenance programmer, the comments will guide the way through the existing code.

2.9.3 Exercises

1. Rewrite the following code segment to be more readable without altering the meaning:

```
int sum = 0; double average; sum
=

first +
second + third; average =
((
double) sum
)/ 3; System
.   out.println (
average)
;
```

2. Show what would be printed by the following code segment:

```
// int x = 3;
double x = 1, y = 2;
/*
y = 4;   //  x = y+3;
System.out.println (x + y);
*/
y = x / y;      // x = 17;
System.out.println (x + y);
```

2.10 Program elements and methods in the GridWorld case study

Chapter 3

Selection Structures

We have seen that the statements in a method are executed sequentially, in the order in which they appear in the method definition. However, it is often that we might wish to alter this *flow of control*. For example, we may wish to execute a statement only if certain conditions are satisfied; or we may wish to skip over a statement if certain other conditions are satisfied. For this purpose, Java provides *selection structures*. We have *one-way* selection structures and *two-way* selection structures, which permit us to execute statement(s) *conditionally*. But first we need to discuss comparison operators and boolean operations.

3.1 Comparison operators

Java numbers may be compared with operators similar to those you've seen in math courses. These operators operate on two numbers, and always produce a boolean result: `true` or `false`. The six comparison operators are described in Figure 3.1.

It is important to note that comparison for equality is a *double* equal sign, ==. Do NOT confuse this operator with the assignment operator which is a *single* equal sign, =. $a = b + c$ can *change* the value of a, but $a == b + c$ can *not* change the value of a.

operation	returns
$x == y$	true only if x is equal to y
$x < y$	true only if x is less than y
$x > y$	true only if x is greater than y
$x <= y$	true only if x is less than or equal to y
$x >= y$	true only if x is greater than or equal to y
$x! = y$	true only if x is not equal to y

Figure 3.1: Definitions of the comparison operators

Also note that Java provides slightly different versions of the following comparison operators that you may have seen in your math courses: $\leq, \geq, \neq$.

These comparison operators have *lower precedence* than the arithmetic operators. Hence, the expression `3 == 4 + 5` is the same as `3 == (4 + 5)`. The addition is done before the comparison.

Comparison for equality (or inequality) may be applied to reference types and to booleans:

```
myStudent == null          // true only if myStudent stores a null reference
myStudent != null          // compare for inequality
(x < 3) == false           // same as x >= 3
```

3.1.1 Exercises

1. What is the *value* of each of the following (assume `x` has been declared as an int variable)?

   (a) `3 != 4`

   (b) `-99 <= 3`

   (c) `7 - 7 == 2 / 3`

   (d) `x = 3`

   (e) `(2 < 5) == (3 > 5)`

2. *Rewrite* the following expression more succinctly without changing its meaning:

   `(x > 3) == (x < 3)`

3.2 Boolean operators

In chapter 1 we exposed the `boolean` type. This type consists of only two values: `true` and `false`. Do not think of these as strings of characters, nor as variables; they are constants, in the same way that 23 and -302 are constant values of type `int`.

3.2.1 AND, OR, NOT

Just as there are operations on numbers, there are operations on booleans. The basic operations are *or*, *and*, and *not*. These operations are represented by the operators ||, `&&`, and !, respectively, and are defined in Figure 3.2. Note that:

- The *or* operation produces a `false` result only when both operands are `false`.
- The *and* operation produces a `true` result only when both operands are `true`.

- The *not* operation is a *unary* operation; it has only one operand, to its right. It produces as a result the logical complement of its operand.

Figure 3.3 shows some examples of logical statements in English to further describe the meanings of these logical operations. These logical operations are fundamental to computer science and will be used extensively in writing Java programs.

Just as we had a formal definition of arithmetic expressions, we can provide a similar definition for boolean expressions:

A boolExpr may be:

1. `false`
2. `true`
3. boolean variable
4. expr comparison expr
5. boolExpr || boolExpr
6. boolExpr && boolExpr
7. ! boolExpr
8. (boolExpr)

in which `comparison` represents any of the six comparison operators shown in Figure 3.1, and `expr` is an arithmetic expression as defined in chapter 2. Again we have a definition in which we use the word we are defining, boolExpr in this case. As with arithmetic expressions a boolean expression is inherently recursive, and consequently there is no other way to define a boolean expression. An example of a boolean expression is:

```
!a || b && c
```

in which `a`, `b` and `c` are assumed to be declared as `boolean`.

Figure 3.4 and Figure 3.5 show how this boolean expression can be diagrammed as we did with arithmetic expressions in chapter 2. Once again we have an ambiguous definition: there are at least two different ways of diagramming the same expression.

This ambiguity is resolved as follows:

- ! *not* is executed before `&&` *and*
- && *and* is executed before || *or*

As usual, parentheses may be used to effect the desired order of operations. When you cannot remember these precedence rules, parentheses can be used even if not needed.

The logical *or* operation is designated by the Java operator `||`

x	y	x `||` y
false	false	false
false	true	true
true	false	true
true	true	true

The logical *and* operation is designated by the Java operator

x	y	x `&&` y
false	false	false
false	true	false
true	false	false
true	true	true

The logical *not* operation is designated by the Java operator ! and is a unary operation

x	! x
false	true
true	false

Figure 3.2: Definitions of the logical operations *or*, *and*, and *not*.

Statement	`true` or `false`
This book is written in French *or* elephants have 6 legs	`false`
This book is written in French *or* elephants have 4 legs	`true`
This book is written in French *and* elephants have 4 legs	`false`
This book is written in French *and* elephants have 4 legs	`false`
This book is written in English *and* elephants have 4 legs	`true`
This book is not written in French	`true`

Figure 3.3: Examples of logical statements in English

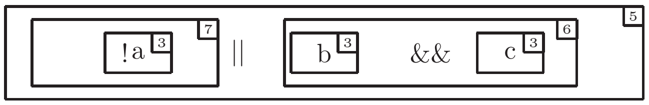


Figure 3.4: Applying the definition of boolean expression to: $!a||b\&\&c$

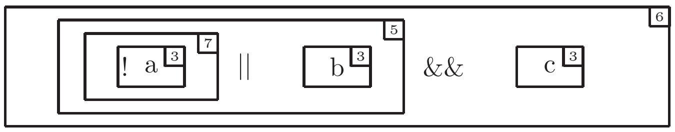


Figure 3.5: An alternative application of the definition of boolean expression to: $!a||b\&\&c$

We now see that Figure 3.4 represents the desired interpretation for the given boolean expression. In other words,

```
!a || b && c
```

is the same as

```
((!a) || (b && c)
```

All three of these boolean operators have lower precedence than the comparison operators:

```
 x < 3 || y == 0
```

is the same as

```
 (x < 3) || (y == 0)
```

3.2.2 Short circuit evaluation

If `b` represents any boolean expression, then we have the following identities:

- `true || b` is always true
- `false && b` is always false

To see this look at Figure 3.2 and substitute `true` (or `false`) for the operand `x`.

Java can make use of these identities to *optimize* the evaluation of boolean expressions (and provide a convenience for the programmer). When evaluating a boolean expression, the *left* operand of a logical operator is *always* evaluated first. If the operator is an OR (`||`) and the left operand is true, the result *must* be true; hence, there is no need to evaluate the right operand, and Java will not attempt to evaluate the right operand. Likewise, if the operator is an AND (&&) and the left operand is false, the result *must* be false; hence, there is no need to evaluate the right operand, and Java will not attempt to evaluate the right operand. This is called *short circuit* evaluation.

Note that this allows for an easy way to check, and avoid, possible run-time errors:

```
student != null && student.getGPA() === 4.0
```

The call to `student.getGPA()` will cause a null pointer exception if student is a null reference. But we avoid that error by checking for a null reference first, and the right operand of the && is not evaluated.

The following would *not* work correctly:

```
student.getGPA() == 4.0 && student != null
```

because the left operand of the && is always evaluated first; in this case it would cause a null pointer exception.

3.2.3 De Morgan's Laws

There are other logical identities which, in some cases, can simplify boolean expressions. Here we examine the two identities known as De Morgan's Laws. They can be expressed concisely, for boolean expressions `x` and `y`:

x	y	`!(x && y)`	`(!x || !y)`	`!(x || y)`	`!x && !y`
false	false	true	true	true	true
false	true	true	true	false	false
true	false	true	true	false	false
true	true	false	false	false	false

Figure 3.6: Proof of De Morgan's Laws

- `!(x && y) = !x || !y`
- `!(x || y) = !x && !y`

For example, the following two boolean expressions shown below are perfectly equivalent:

```
! (salary < 100000 && status==FULL_TIME)
salary >= 100000 || status!=FULL_TIME
```
[1]

The second version is perhaps a little simpler, and therefore preferable, though both versions would behave the same.

A proof of De Morgan's Laws is shown as a *truth table* in Fig 3.6. A truth table shows the evaluation of a boolean expression for all possible values of the variables. In this case there are two variables and therefore four rows in the truth table. The first of De Morgan's Laws is proved by noting that columns 3 and 4 are the same. The second of De Morgan's Laws is proved by noting that columns 5 and 6 are the same.

3.2.4 Exercises

1. Find the *value* of each of the boolean expressions shown below:
   (a) `4 > 3 && 4 < 2`
   (b) `(3 < 2 || 2 - 2 == 0) && 5 > 3`
   (c) `!  true || 3 >= 2`
   (d) `!  ( x < 2 || 3 > 1) || (x == 0 || true)`
2. Draw boxes around each boolean expression from the above problem, as shown in Figure 3.4. Be sure to show the rule numbers, and adhere to the precedence rules for boolean expressions.
3. Simplify the following boolean expressions (assume x and y have been declared as ints):
   (a) `(x == 0 || x != 0) && y < 0`
   (b) `y < 0 || (x == 0 || x != 0)`
   (c) `(x > 0 && x < 0) || y == 3`

[1]Note that the logical complement of $<$ is $>=$

   (d) `y == 3 && (x > 0 && x < 0)`

4. Assume x and y have been declared as ints. Which, if any, of the following expressions will cause a run-time error when the value of x is 0 (division by 0)?

   (a) `y/x > 3 || x == 0`

   (b) `x == 0 || y/x > 3`

   (c) `y/x > 3 && x != 0`

   (d) `x != 0 && y/x > 3`

5. Which of the following boolean expressions is *equivalent* to:
   `!(name.length()>12 && gpa < 1.0)` ?

   (a) `name.length()>12 && gpa < 1.0`

   (b) `name.length()<12 || gpa > 1.0`

   (c) `name.length()<=12 && gpa >= 1.0`

   (d) `name.length()<=12 || gpa >= 1.0`

6. For each of the following boolean expressions use one of De Morgan's Laws to show an *equivalent* boolean expression.

   (a) `!(name.equals("joe") && gpa <= 3.5)`

   (b) `name.equals("susie") && gpa == 3.5`

   (c) `name.equals("susie") || gpa == 3.5`

   (d) `!(name.equals("sue") || gpa > 3.5)`

3.3 One-way selections

One-way selections are used when we want to execute a statement, but only if a certain condition holds. The Java keyword `if` is used for this purpose. The general form is:

```
if (boolean expression) statement
```

The statement is executed only if the boolean expression is true. For example:

```
if (credits > 0)
    gpa = gradePoints / credits;
```

In this example, the value of the variable `gpa` will be changed to the result of the division, but only if the value of the variable `credits` is greater than 0.

A flow diagram for the one-way selection is shown in Figure 3.7 in which *Condition* represents the boolean expression in parentheses.

A few remarks on one-way selections:

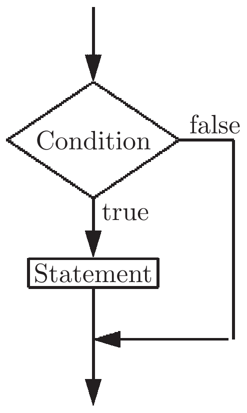


Figure 3.7: Flow diagram for a one-way selection structure

- The parentheses are *always* needed.
- The consequence of the `if` must be a *single* statement. If it is desired that several statements be executed when the condition is true, we will use a compound statement, defined later in this chapter.
- The condition in the parentheses must produce a `boolean` result, i.e. it must evaluate to either `true` or `false`.

The student should be cautious when using boolean expressions; a boolean expression which 'sounds ok' in English is not necessarily correct. For example, suppose we wish to calculate a student's gpa, but only if the number of credits is positive and less than 200. We may be tempted to write it as:

```
if (credits > 0 && < 200) gpa = ...
```

The compiler would reject this as incorrect. To see why, try to box the boolean expression using the rules of our definition; it cannot be done. This statement should be written as:

```
if (credits > 0 && credits < 200) gpa = ...
```

3.3.1 Exercises

1. Which of the following `if` statements contain syntax errors (assume `x` and `y` have been declared as int variables and `b` has been declared as a boolean variable)?

   (a)
   ```
   if x==0
      y = 3;
   ```

   (b)
   ```
   if (b)
      y = 3;
   ```

(c)
```
if (x > 0 && < 100)
   y = 3;
```

(d)
```
if (b = true)
   y = 3;
```

2. Show what would be printed by each of the following `if` statements (assume `x` and `y` are declared as int variables, and `b` is declared as a boolean variable):

   (a)
   ```
   x = 3;
   y = 4;
   if (x == y)
      System.out.println(x);
      System.out.println(y);
   System.out.println ("done");
   ```

   (b)
   ```
   x = 3;
   y = 5
   b = x + y < 8;
   if (b == true)
      System.out.println(x);
   System.out.println ("done");
   ```

   (c)
   ```
   x = 3;
   y = 5
   b = x + y < 8;
   if (b = true)
      System.out.println(x);
   System.out.println ("done");
   ```

3. Show a better way of writing this `if` statement (which would prevent the slip-up exposed in part (d) of the above exercise):

   `if (b == true) ...`

3.4 Two-way selections

Two-way selections are similar to one-way selections. The main difference is that two statements are provided in a two-way selection, exactly one of which must be executed. The general format is :

```
if (boolean expression)
   Statement1
```

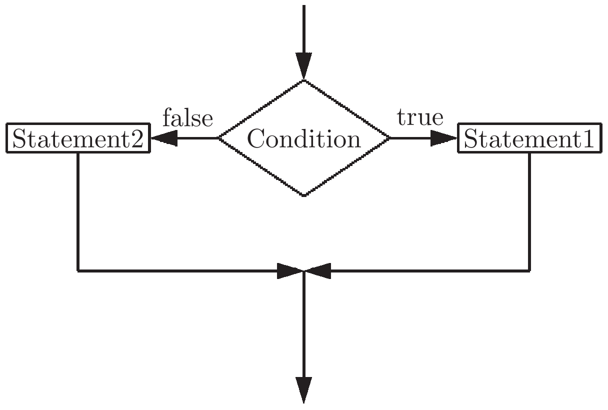


Figure 3.8: Flow diagram for a two-way selection structure

```
else
   Statement2
```

As with one-way selections, the boolean expression is evaluated. If it is true, Statement1 is executed. If it is false, Statement2 is executed.

A diagram of two-way selections is shown in Figure 3.8.

Note that (many of these have been noted previously with respect to one-way selections):

- The parentheses are *always* needed.
- The `true` consequence of the `if` must be a *single* statement. If it is desired that several statements be executed when the condition is true, we will use a compound statement, defined later in this chapter.
- The `false` consequence of the `if` must also be a *single* statement. This is the statement which comes after `else`. If it is desired that several statements be executed when the condition is false, we will use a compound statement, defined later in this chapter.
- The condition in the parentheses must produce a `boolean` result, i.e. it must evaluate to either `true` or `false`.
- Exactly one of the two statements must be executed because the boolean expression must evaluate to either `true` or `false`.

An example of a two-way selection is shown below:

```
if (credits >  0)
    gpa = gradePoints / credits;
else
    gpa = 0.0;
```

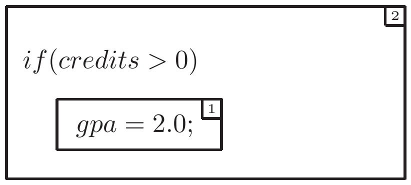


Figure 3.9: Applying the definition of Statement to:
$if(credits > 200)gpa = 2.0;$

In this example the variable `gpa` is set to the result of the division only if the variable `credits` is positive, and the variable `gpa` is set to 0.0 only if the variable `credits` is not positive. Notice that we have carefully *indented* both the true and false consequences of the `if`. This is not required by the compiler, but is done to make the program easier to read and maintain. The indentation of the two assigment statements is supposed to clarify the fact that they are part of the `if` statement.

We can now provide a preliminary definition of a Java statement, abbreviated stmt:

A Java stmt may be:

1. variable = expression ;
2. if (boolean expression) stmt
3. if (boolean expression) stmt else stmt

Once again, we have an inherently recursive construct; there is no way to define *statement* without using the word *statement* in the definition. Fortunately, rule 1 does not use the word *statement*, providing a base case.

Figure 3.9 shows how the definition can be applied to the one-way selection:

```
if (credits > 0)
   gpa = gradePoints / credits;
```

Figure 3.10 shows how the definition can be applied to the two-way selection:

```
if (credits > 0)
   gpa = gradePoints / credits;
else
   gpa = 0.0;
```

The definition of Stmt shown above indicates that both the true and false consequences of an `if` can be any Stmt, which would include `if` statements. In other words `if` statements may contain `if` statements, which in turn may contain other `if` statements, ... to any number of levels in depth. In other words `if` statements may be *nested* as deeply as you may wish. We now look at a more interesting example:

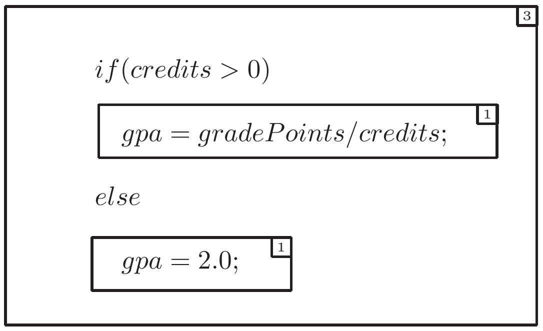


Figure 3.10: Applying the definition of Statement to: if credits > 0) gpa = gradePoints / credits; else gpa = 0.0;

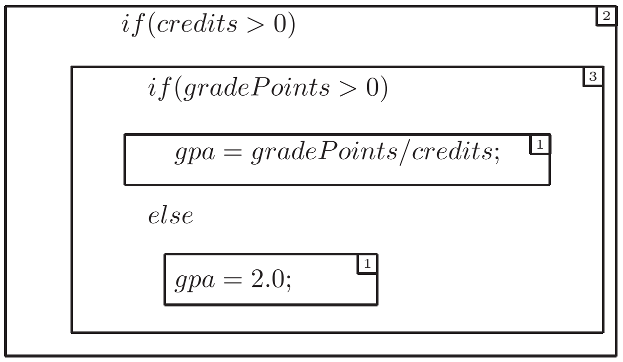


Figure 3.11: Applying the definition of Statement to an `if` statement which contains another `if` statement

```
if (credits > 0)
if (gradePoints > 0)
   gpa = gradePoints / credits;
else
   gpa = 2.0;
```

The indentation in this example is not good, but we will improve it after some discussion. Figure 3.11 shows how we can box all the statements using our definition of statement. However, Figure 3.12 shows a different way to box the statements. In Figure 3.11 we have a two-way selection inside a one-way selection; i.e. the `else` goes with the *second* `if`. This means that gpa will be set to 2.0 only if either credits or gradePoints is not positive.

In Figure 3.12 we have a one-way selection inside a two-way selection; i.e. the `else` goes with the *first* `if`. This means that gpa will be set to 2.0 only if credits *only* is not positive.

The fundamental question here is 'which `if` is matched with the `else`?' Could it be that once again we have an ambiguous definition? Yes, that is the

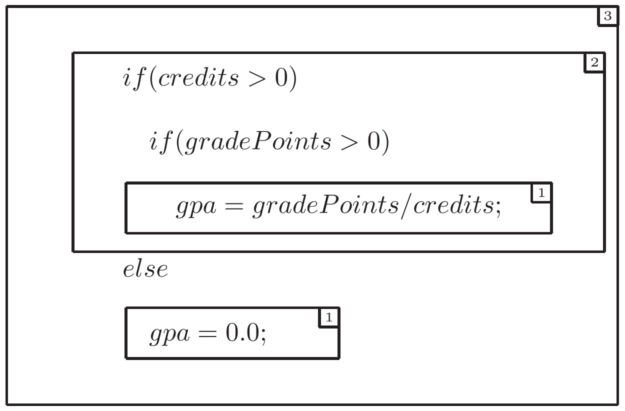


Figure 3.12: An alternate application of the definition of Statement to the same `if` statement containing an `if` statement

case, because there are two different interpretations for the same statement. This is a classic ambiguity problem in computer science, known as the *dangling else*. To resolve the ambiguity we state the following rule:

- Each `else` is matched with the nearest preceding unmatched `if`.

This means that the `else` should be matched with the second `if`, and we have a two-way selection inside a one-way selection. The correct interpretation is shown in Figure 3.11.

3.4.1 Exercises

1. Which of the following `if` statements contain syntax errors (assume `x` and `y` have been declared as int variables and `b` has been declared as a boolean variable)?

   (a)
   ```
   if (x >= 17)
         y = 2;
   else
         y = 3;
   ```

   (b)
   ```
   if (x >= 17)
         y = 2;
   else
         y = 3;
   else
         y = 0;
   ```

(c)
```
if (x >= 17 && < 25)
      y = 2;
else
      y = 3;
```

(d)
```
if (b)
      b = false;
else
      b = true;
```

(e)
```
if (x > 0)
    x = 2;
    y = 3;
else
      b = true;
```

2. Show what would be printed by each of the following `if` statements (assume `x` and `y` have been declared as int variables and `b` has been declared as a boolean variable):

(a)
```
x = 7;
y = 3;
if (x >= y+4)
   System.out.println (x);
else
   System.out.println (y);
   System.out.println ("false case");
```

(b)
```
x = 7;
y = 3;
if (x < 0)
   System.out.println (x);
else
   System.out.println (y);
   System.out.println ("false case");
```

(c)
```
x = 7;
y = 3;
b = x < y;
if (b)
   System.out.println (x);
else
   System.out.println (y);
```

(d)
```
x = 7;
y = 3;
if (x > y)
if (x <= y)
System.out.println (x);
else
System.out.println (y);
else
System.out.println (x + y);
```

(e)
```
x = 7;
y = 3;
if (x < y)
   if (x <= y)
      System.out.println (x);
else
   System.out.println (x + y);
```

3. Draw a box around each *statement* in the previous problem, as shown in Figure 3.11. Show the rule number applied, and be sure to resolve the dangling `else` correctly.
4. You are given a variable declared as:

   `Student stu;`

   Show a single `if` statement which will print stu's gpa if stu is not null, and will print `"null"` if stu is null.

3.5 Compound statements and scope

3.5.1 Compound statements

We noted in the previous section that the consequence(s) of an `if` in a one-way selection or in a two-way selection must be a *single* statement. However, it is often the case that we wish to execute (or not execute) more than one statement, i.e. a whole group of statements. The solution here is to use a *compound statement*. A compound statement is 0 or more statements enclosed in curly braces. This compound statement is treated as one big statement in a selection structure. For example:

```
double creditsAsDouble;
boolean active;          // true only if this Student is active
int tuition;             // current tuition in whole dollars
active = true;

if (credits > 0)
```

Program code	value of a	value of b	value of c
int a,b;			
a = 3;	3		
b = 7;	3	7	
{ // compound stmt	3		
int b;	3		
int c;	3		
b = 5;	3	5	
a = 9;	9	5	
c = 11;	9	5	11
}	9	7	

Figure 3.13: Scope of local variables

```
   {  creditsAsDouble = credits;
      gpa = gradePoints / creditsAsDouble;
   }
else
   {  tuition = 0;
      active = false;
   }
```

In this example, a group of two statements is executed if the condition is true and another group of two statements is executed if the condition is false.

Note that it is possible to have just one statement in a compound statement. Moreover, many instructors recommend this practice, because students often omit the curly braces when they are actually needed.

3.5.2 Scope of variables

Variables declared inside methods are called *local* variables, as opposed to *instance* variables (described in chapter 1). When a variable is declared inside a compound statement, the *scope* of that variable is limited to that compound statement; it is not known outside the compound statement. We say the variable is *local* to that compound statement. The scope of an instance variable is the entire class. It is possible to redefine a local variable in a separate scope as shown in Figure 3.13. When a method terminates, all local variables and parameters declared in that method are disposed from memory; they no longer exist.

Care must be taken to distinguish between local variables (or parameters) named the same as fields. Field names may be prefixed with `this.` to distinguish them from a local variable or parameter with the same name. A very common (and nasty) error is shown below:

```
// This method is in the Student class which has a field
// named gpa.
```

```
public void someMethod ()
{   int gpa = 3.4;
    System.out.println ("This student's gpa has been changed to 3.4");
    ...
}
```

The student's gpa has not been changed to 3.4 because the variable gpa is a *local* variable; it is declared in the body of the method. It is not the *field* gpa. When this method executes there are two, different, variables named gpa; one is a field and the other is a local variable. To refer to the field gpa, use `this.gpa`. To correct the problem, do not declare gpa as an `int`; do not declare it at all, and you will have just one occurrence of the variable gpa. The error described here is rather nasty because the compiler will not produce an error message; instead there will likely be a runtime error, a crash, or incorrect output. It may take several hours of work with the debugger to track down this problem.

3.5.3 Java statements - revisiting a formal definition

We can now expand our definition of statement to include compound statments.

A Java stmt may be:

1. variable = expression ;
2. if (boolean expression) stmt
3. if (boolean expression) stmt else stmt
4. method call, such as `System.out.println()`l
5. { stmt stmt stmt ... }

Figure 3.14 shows a diagram for the statement given above using this definition.

3.5.4 Exercises

1. Which of the following statements contain syntax errors (assume `x` has been declared as an int variable)?

   (a)
   ```
       if (x > 0)
          {  System.out.println ("positive");
       else
    System.out.println ("negative");
          }
   ```

   (b)
   ```
       if (x > 0)
          {  System.out.println ("positive"); }
       else
          {   }
   ```

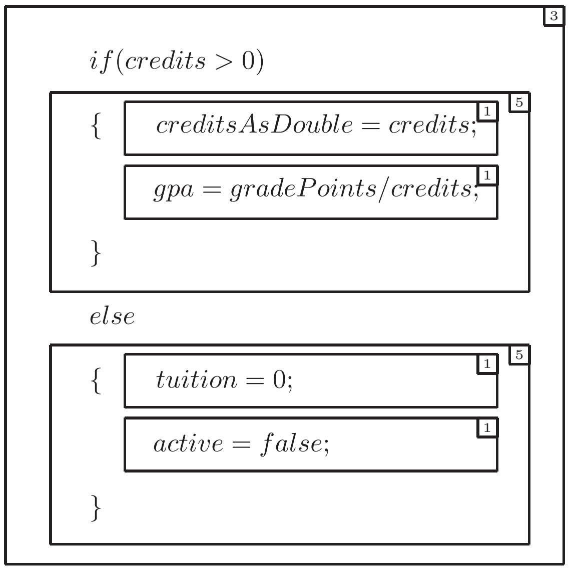


Figure 3.14: Applying the definition of Statement to an `if` statement containing compound statements

(c)
```
     if (x > 0)
          System.out.println ("positive"); }
x = 0;
```

(d)
```
{ int y = 0;
  x = x + 1;
  y = x + 2;
}
```

2. Show what would be printed in each of the following:

(a)
```
{  int a = 7, b = 8;
   {  char b = '$';
      System.out.println ("a is " + a);
      System.out.println ("b is " + b);
   }
   System.out.println ("a is " + a);
   System.out.println ("b is " + b);
}
```

(b)
```
{  int a = 7, b = 8;
   {  char b = '$';
      a = 3;
```

```
          System.out.println ("a is " + a);
          System.out.println ("b is " + b);
       }
       System.out.println ("a is " + a);
       System.out.println ("b is " + b);
    }
```

(c)
```
    int x = 7;
    if (x >0)
       {
          x = x + 1;
          System.out.println (x);
       }
    else
          x = x - 1;
          System.out.printn (x);
```

(d)
```
    x = 12;
    if (x > 20)
       x = x + 1;
       System.out.println (x);
    System.out.println ("done");
```

3. Draw a box around each statement in the previouis problem, as shown in Figure 3.11. Show the rule number for each statement. For purposes of this problem, you may ignore variable declarations such as `int a = 7;`

3.6 Recursive methods revisited

As we mentioned earlier, methods can call themselves. Such a method is a *recursive* method. In order for this to work correctly, there are two criteria:

- There must be a path through the method which does not involve a call to itself. This is called the *base case*.

- The input(s) to the method, i.e. the parameter(s) must be reduced, in some way, when the method calls itself.

The mathematical function *factorial(n)* is defined to be the product of all whole numbers from 1 through n:
$factorial(n) = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$
Sometimes this function is written with an exclamation point: n! For example, $factorial(4) = 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$
Here is another definition of factorial:

$$\begin{aligned} factorial(1) &= 1 \\ factorial(n) &= n \cdot factorial(n-1) \end{aligned}$$

This definition can be used to write a recursive method which will return the value of factorial(n).

```
/** @param n is greater than or equal to 0
 *  @return the value of n!
 */
public int factorial(int n)
{  if (n<2)
      return 1;                  // base case, 1! = 1
   return n * factorial(n-1);    // recursive case
}
```

We should note a few aspects of this method:

- The API at the top has a few *javadoc keywords*:
  - `@param` is used to describe valid value(s) for any parameter to the method. This is a precondition. If the parameter `n` is less than 1, this method is not guaranteed to work.
  - `@return` defines what this method will return as its explicit result. This is an example of a postcondition (there could be others).
- The `if` statement checks for the base case; the case where the parameter's value is 1. In this case this call to the method is terminated immediately, returning the value 1.
- If the `if` condition is false, control falls through to the next statement, which is the recursive call, and the result of the multiplication is returned, terminating this call to the method.
- We could have used an `else` with the `if` but in this case it isn't needed because the `return` terminates the method.

Figure 3.15 depicts what happens when factorial(3) is called. It calls factorial(2), which in turn calls factorial(1), which returns a value, enabling the other calls to factorial to return values. The final result is 6.

This is our first example of a recursive method, and we will see others. It may surprise the student to learn that there are some problems in computer science which *must* be solved with recursive methods.

3.6.1 Exercises

1. Insert a print statement in the `factorial` method to print the value of the parameter, before the `if` statement. What is the output, when the parameter value is 5?

```
fact(3) = 3 * fact(2)
  fact(2) = 2 * fact(1)
   fact(1) = 1
  fact(2) = 2 * fact(1) = 2 * 1 = 2
fact(3) = 3 * fact(2) = 3 * 2 = 6
```

Figure 3.15: Execution of factorial(3)

2. The *fibonacci* sequence is

   1, 1, 2, 3, 5, 8, 13, 21, 34, ..
   Note that each number in this sequence is the sum of the two previous numbers. If `fib(n)` returns the n^{th} value in the sequence, we can say that:

   ```
   fib(1) = 1
   fib(2) = 1
   fib(n) = fib(n-1) + fib(n-2)   for values of n >= 3
   ```

   (a) What are the next two numbers after 34 in the fibonacci sequence?

   (b) Write a recursive method named `fib` which will return the n^{th} value in the fibonacci sequence. Use the definition given above.

   ```
   /** @return The nth value in the fibonnaci sequence.
    *  @param n must be greater than or equal to 1.
    *  This is a recursive method.
    */
   public int fib(int n)
   ```

3. Write a recursive method named `mult` with two int parameters (the second parameter must not be negative). It should return the product of the two parameters. Do *not* use the * or / operators; instead use the following:

   ```
   x * 0 is 0, for any value of x
   x * y is x + x*(y-1), for any positive value of y
   ```

   The API is:

   ```
   \** @return the product x*y, without using * or /
    *  @param y is not negative
    */
   public int mult(int x, int y)
   ```

4.

3.7 Comparing Strings and other reference types

Up to this point all of the comparison operations have involved primitive types. We now wish to discuss the comparison of reference types.

3.7.1 Comparison for equality or inequality

When comparing Strings or other reference types for equality, one should not use the == operator, nor the != operator. Instead one should use the method `.equals` (this method is defined in the String class), as shown below:

```
if (name.equals("joe") && gpa == 4.0)
   System.out.println ("joe is perfect");
```

The compiler will permit you to write the comparison as `name == "joe"` but in this case you are comparing the references, not the objects to which they refer. With Strings the `==` comparison will often work, but we can find cases for which it will not work correctly. A good habit is always to use `.equals()` when comparing reference types, unless you really mean to compare the references:

```
   if (name == null) ...  // check for null reference
```

To compare for inequality, use the logical NOT operator (!):

```
if ((! name.equals("joe")) && gpa != 4.0)
   System.out.println ("Somebody other than joe is not perfect");
```

The usage of `.equals` presumes that the class of the object to which the method is applied defines the meaning of `.equals` (the String class and most other classes in the Java class library take care of this for us). However, our Student class could define two Students to be equal if and only if they have the same name and same ssn.

Figure 3.16 shows object diagrams and the results of the two kinds of comparisons when applied to reference variables. In this object diagram the variables `s1` and `s2` refer to the same object; consequently they store the same value (the same reference), and the comparison `s1 == s2` will be true. However, the variables `s1` and `s3` refer to separate objects; consequently they store different references, and the comparison `s1==s3` will be false. The objects to which s1 and s3 refer are, presumably, equal since the corresponding fields are all equal (this assumes that the Student class provides a `.equals()` method, and the comparison `s1.equals(s3)` is true, as is the comparison `s2.equals(s3)`.

3.7.2 Ordered comparisons

To compare primitives for ordering, we can use one of the comparison operators given earlier in this chapter:

```
int x,y; ...
if (x < y) ...
if (x >= y) ...
```

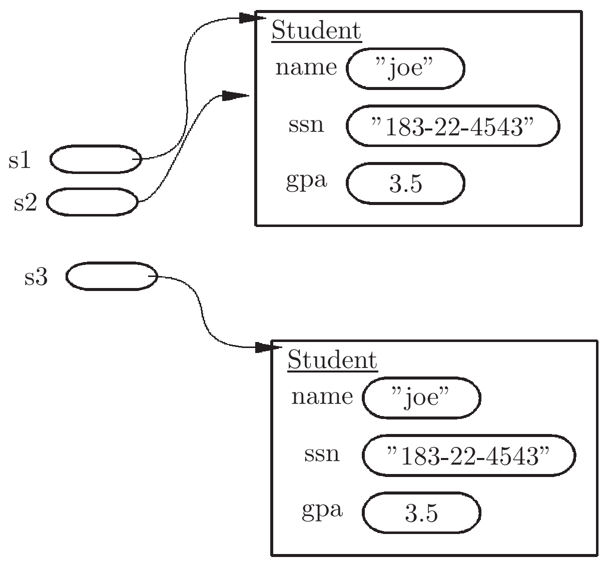


Figure 3.16: An object diagram showing the difference between comparison with == and comparison with .equals(): `s1==s2` is `true` and `s1==s3` is `false` and `s1.equals(s3)` is `true`

This cannot be done with boolean types; The comparison `true < false` will produce a syntax error at compile time.

The standard comparison method for reference types is `compareTo`, which has one parameter. It will compare *this* object with the parameter which should be an instance of the same class. The `compareTo` method will return an int which is:

- negative if *this* object is less than the parameter
- zero if *this* object is equal to the parameter
- positive if *this* object is greater than the parameter

If you have trouble remembering these return values, imagine that the returned value is like the result of the subtraction *this - parameter*.

When comparing Strings, the internal codes of the characters of the two Strings are compared, left to right. As soon as a discrepancy is found, the result is determined by the two characters being compared. In other words, comparison of Strings is essentially an alphabetic comparison: the String `s1` is less than the String `s2` if s1 precedes `s2` alphabetically. Examples of values returned by compareTo are shown in Figure 3.17. We will have more to say about compareTo in chapter 6.

comparison	result
`"abc".compareTo("def")`	negative
`"abc".compareTo("abc")`	zero
`"def".compareTo("abc")`	positive
`"abc".compareTo("aba")`	positive
`"abc".compareTo("abaci")`	negative
`"abc".compareTo("Bbc")`	positive
`"99".compareTo("100")`	positive

Figure 3.17: Returned values for the compareTo method, applied to Strings

3.7.3 Exercises

1. Which of the following contains a syntax error (assume the variables `s1` and `s2` have been declared as Strings)?

(a)
```
if (s1 == s2)
   System.out.println ("equal");
```

(b)
```
if (s1 < s2)
   System.out.println ("smaller");
```

(c)
```
if (s1.compareTo(s2))
   System.out.println ("smaller");
```

(d)
```
if (3.equals(4))
   System.out.println ("smaller");
```

2. What will be printed by each of the following (assume the variables `s1` and `s2` have been declared as Strings)?

3. (a)
```
s1 = "john";
s2 = "John";
if (s1.equals(s2))
   System.out.println ("equal");
else
   System.out.println ("not equal");
```

(b)
```
s1 = "john";
s2 = "johnson";
if (s1.equals(s2))
   System.out.println ("equal");
else
   System.out.println ("not equal");
```

(c)
```
s1 = "john";
s2 = "johnson";
if (s2.equals(s1 + "son"))
   System.out.println ("equal");
else
   System.out.println ("not equal");
```

(d)
```
s1 = "john";
s2 = "johnson";
if (s2 == s1 + "son")
   System.out.println ("equal");
else
   System.out.println ("not equal");
```

(e)
```
s1 = "john";
s2 = "johnson";
if (s2.compareTo(s1) < 0)
   System.out.println ("smaller");
else
   System.out.println ("not smaller");
```

3.8 Selection structures in the GridWorld case study

3.9 Projects

1. The greatest common divisor of two whole numbers, x and y, is the largest divisor which they share. For example, if x=70 and y=30, the divisors of x are 2, 5, 7, 10, 14, 35 and the divisors of y are 3, 5, 10, 15. The greatest common divisor is 10.

   The Euclidean algorithm will find the greatest common divisor of two positive whole numbers efficiently. The algorithm can be summarized as follows:

   (a) Let r be the remainder when x is divided by y.
   (b) If r is 0, the greatest common divisor is y.
   (c) If r is not 0, the greatest common divisor is the greatest common divisor of y and r.

   Use this algorithm to define a method named `gcd` with two parameters, both of which should be positive integers. The method should return the greatest common divisor of its two parameters.

```
/** @return The greatest common divisor of x and  y.
 *  @param x is positive.
 *  @param y is positive.
 */
public int gcd (int x, int y)
```

2. Because there are not exactly 365 days in a solar year, the calendar must be corrected every four years by inserting an extra day. This is called a *leap year*. This is done in years which are divisible by 4 (2008, 2012, 2016, ...). However, this is not a perfect correction. Every hundred years we skip a leap year. 2100, 2200, 2300 will not be leap years even though they are divisible by 4. Yet another correction is necessary; every millenium will be a leap year even though it is divisible by 100 (2000, 3000, 4000, ...).

   Define a method named `leapYear` with one parameter, an int representing a year. The method should return a boolean – true only if the given year is a leap year.

   Hints:

   - Use the mod operator (%) to test for divisibility.
   - Check for least frequently occurring cases first: 1000, then 100, then 4.
   - Terminate the method with a `return` statement as soon as the result has been determined.

3. Roman numerals are often used to indicate the year that a movie was made; they are also sometines used in outlines. The Roman letters and their decimal equivalents are shown below:

Roman symbol	Decimal equivalent
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

   Some examples of roman numerals are shown below:

Roman symbol	Decimal equivalent
III	3
IV	4
V	5
VI	6
VIII	8
IX	9
X	10
LXIV	64
MMDCXXIX	2629

Write a method named `toRoman` with one parameter, an int. The method should return a String representation of the given int as roman numerals.

```
/** @return The Roman numeral representation of n.
 * @param n is in the range 1..3999
 */
public static String toRoman (int n)
```

Hint: Write a private helper method named `roman` with 4 parameters which returns a String:

- an int in the range 1..10
- The Roman symbol for 1, 10, 100, 1000
- The Roman symbol for 5, 50, 500
- The Roman symbol for 10, 100, 1000

If the helper method call is `roman (7, "X", "L", "C")`, its result will be `"LXX"`.

4. A circle in the x-y plane can be represented by three numbers: its radius and the x,y coordinates of its center. Two different circles in a plane can intersect in 0, 1, or 2 points. Define a method with 6 parameters, the specs for two circles, which will return the number of points in which the two circles intersect.

```
/** @return the number of intersection points of the two given circles.
 *  @param r1 is the radius of the first circle.
 *  @param x1 is the x coordinate of the center of the first circle.
 *  @param y1 is the y coordinate of the center of the first circle.
 *  @param r2 is the radius of the second circle.
 *  @param x2 is the x coordinate of the center of the second circle.
 *  @param y2 is the y coordinate of the center of the second circle.
 *  The parameters represent two different circles.
 */
public static int intersection (double r1, double x1, double y1,
                                double r2, double x2, double y2)
```

Because floating point numbers are not exact, you may use a tolerance value when comparing floating point values, e.g. `double epsilon = 1e-12`. Rather than comparing two distances for exact equality, compare for equality within this tolerance.

Hint: Define a private helper method which will calculate the distance between two points: $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

The method `sqrt` in the `Math` class will be helpful here.

Chapter 4

Iteration Structures

The computational power of computers lies in their ability to repeat a group of instructions a number of times. If it were not for this capability, computers would be as useful as old time 4-function calculating machines.

We call the section of code containing statements to be repeated a *loop*, probably because a diagram depicting the sequendce in which statements are executed would form a closed loop. Every loop has a *loop control* section and a *loop body*. The body is the statement to be repeated. The loop control section determines the number of times the body is repeated. There are several kinds of loop control, and we discuss two of them in this chapter - *while* loops and *for* loops.

4.1 Looping with `while`– pre-test loops

The `while` loops are most useful when the number of times a loop repeats cannot be calculated; it may depend on complex conditions arising as the loop body executes.

The format of a `while` loop is shown below:

```
while (boolean expression)
   statement
```

The intent here is that the statement forms the body of the loop; it is the part which is repeated. The body is performed repeatedly but only if the boolean expression evaluates to `true`. If the the boolean expression evaluates to `false`, the loop terminates, and control falls through to the next statement after the loop. A flow diagram depicting the flow of control for a `while` loop is shown in Figure 4.1.

An example of a loop is shown below:

```
credits = getCredits();
while (credits > 0)
```

```
{
 totalCredits = totalCredits credits;
 credits = getCredits();
}
```

In this example we presume there is a method, `getCredits()`, which will return the number of credits for a student (perhaps for one semester). Each time it is called it returns the number of credits for successive semesters. The value returned is added to `totalCredits` and the result is stored back into `totalCredits`. The variable `totalCredits` is used as an *accumulator*; i.e. it accumulates the total number of credits taken be a Student as the loop repeats.

We point out that:

- The loop body consists of *one* statement. If it is desired to have several statements in the body of the loop, a *compound statement* would be needed, as with selection structures; this is the case in the example shown above.
- The loop shown above will continue to execute as long as the variable `credits` is positive; when this variable is not positive, the condition is false, and the loop terminates.
- If the first call to `getCredits()` returns 0, the loop condition will be false, and the loop body will be executed 0 times.
- A `while` loop is often called a *pretest loop* because the test for continuation is done before the first execution of the loop body, as shown in Figure 4.1.

Notice that in the above example we have carefully *indented* the body of the loop so that it is clear what belongs to the loop and what does not belong to the loop. We did the same thing in chapter 3 with selection structures. It is important that the *appearance* of the program properly exposes the meaning, or *semantics*, of the program.

We now present another example of a pre-test loop. This is a mathematical example, taken from Calculus, though this example is accessible to those who have not yet had Calculus. The *exponential* function is often abbreviated as $exp(x)$ or as e^x (where the constant e is approximately 2.71828182846). Using Calculus we can derive an infinite series to calculate this function:

$exp(x) = e^x = 1 + x/1 + x^2/2! + x^3/3! + x^4/4! + ...$

To calculate this function exactly, we would need an infinite number of terms, something we really don't have the patience for. Consequently we will be satisfied with an *approximation* to the correct result by limiting the number of terms. The method is shown below:

```
 /** Calculate exp(x) with a Taylor series
  *  @param x is not negative.
  *  @param epsilon is the tolerance to which the
  *  result should approximate exp(x).
```

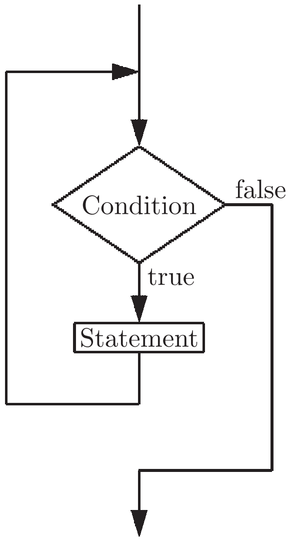


Figure 4.1: Flow diagram for a pretest loop structure

```
      *  @return exp(x), to within tolerance epsilon.
      */
     public double exp(double x, double epsilon)
     {
             double result = 1.0;
             double term = 1.0;
             int ctr = 1;                    // counter for the denominator
             while (term > epsilon)
                 {   term = term * x / ctr;
                     result = result + term;
                     ctr = ctr+1;
                 }
             return result;
      }
}
```

Note that:

- The parameter `epsilon` determines the degree of accuracy desired, since we are forced to return an approximation of the correct result.
- Each term is calculated using the value of the previous term. This is done by multiplying the numerator by `x` and multiplying the denominator by a counter (actually by dividing the term by the counter).
- When the value of `term` is sufficiently small (i.e. less than `epsilon`, it cannot provide a significant change to the result, and the loop terminates.

4.1.1 Infinite loops

The purpose of the loop control is to ensure that the body of the loop repeats the correct number of iterations. Care must be taken to make sure this is correct. It is possible that the loop control, if not correct, will cause the loop to repeat forever, with no termination. In other words, the boolean expression evaluates to `true` and never evaluates to `false`. This situation is called an *infinite loop* and is generally to be avoided.

An example of an infinite loop is shown below:

```
int x = 0;
while (x < 100)
    {  System.out.println ("x is " + x);
    }
```

In the loop shown above the boolean expression `x < 100` is `true`, so the print statement is executed. But the value of `x` is not changed in the body of the loop, so the boolean expression will continue to be `true`, and the loop will continue executing forever (or until the user intercedes by terminating execution with the IDE or by a system interrupt such as ctrl-alt-delete for Windows or ctrl-c for Mac OS). One way to correct this error would be to include a statement such as

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

in the body of the loop.

If your program contains several loops, and one of them is caught in an infinite loop, it will not be clear where the problem lies. In this case rely on a debugger to step through the statements of the program until the guilty loop control is detected.

Because of the complexity of software development, infinite loops can occur even in software which has been well tested. If you have ever noticed that your computer has "frozen", and moving the mouse or typing on the keyboard has no effect, the program (or operating system) is probably caught in an infinite loop, and is failing to accommodate input of any kind.

4.1.2 Exercises

1. Which of the following contain syntax errors (assume the variable `x` has been declard as an int)?

   (a)
   ```
   while (x > 2) x = x - 3;
   ```

   (b)
   ```
   while (x > 2 && < 10)
        { x = x - 3;
          System.out.println (x);
        }
   ```

(c)
```
while (x > 2)
     if (x < 0)
        System.out.println (x);
```

(d)
```
while (x > 0)
   x = x - 1;
   System.out.println (x): }
```

2. Show what would be printed by each of the following (assume the variable `x` has been declard as an int):

(a)
```
x = 4;
while (x >= 0)
   {  System.out.print (x);
      x = x - 1;
   }
```

(b)
```
x = 16;
while (x > 0)
   {  System.out.print (x + " ");
      x = x / 2;
   }
```

(c)
```
x = 1;
while (x < 5)
   {  System.out.print (x);
      x = x + 1;
   }
```

(d)
```
x = 10;
while (x < 5)
   {  System.out.print (x);
      x = x + 1;
   }
```

3. Which of the following is an infinite loop (assume `x` and `y` have been declared as ints)?

(a)
```
x = 12;
while (x > 0)
  {  System.out.println ("x is " + x);
     y = y + 1;
  }
```

(b)
```
x = 12;
y = 99;
while (x > 0 && y < 100)
  {  System.out.println ("x is " + x);
     x = x + 1;
  }
```

(c)
```
x = 12;
while (x == 0)
  {  System.out.println ("x is " + x);
     x = x + 1;
  }
```

(d)
```
x = 12;
while (x != 0)
  {  System.out.println ("x is " + x);
     x = x + 1;
  }
```

4. Define a method named `range` with two parameters which will print, on one line, all the whole numbers in the range from the first parameter to the second parameter. For example, `range(2,5)` should print 2 3 4 5.

```
/** Print, on one line, all whole numbers from
 *  low thru hi, inclusive.
 */
public void range (int low, int hi)
```

4.2 Looping with `for` – counter-controlled loops

In situations where we know, when writing the program, exactly how many times the loop should repeat, a `for` loop, or *counter-controlled loop* can be used.

4.2.1 Autoincrement and autodecrement

Before looking at the `for` statement, we would like to introduce an easy short cut which is frequently used in `for` statements.

A numeric variable can be incremented by 1 very easily by using the `++` notation, which is called *autoincrement*. The expression

```
x++;
```

is equivalent to

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

This autoincrement operator increases the value of x by 1. When the following code has executed, the value of `x` will be 5;

```
int x;
x = 4;
x++;
```

It is possible to use the result of the autoincrement operator as part of a larger expression, but we do not recommend this usage.

Autodecrement is similar to autoincrement, but uses a `--` rather than `++`.

```
x--;
```

is equivalent to

```
x = x-1;
```

This autodecrement operator decreases the value of x by 1. When the following code has executed, the value of `x` will be 3;

```
int x;
x = 4;
x--;
```

4.2.2 The for loop

The format of a `for` loop is shown below:

```
for (declaration ; boolean expression ; expression)
   statement
```

As with `while` loops the statement forms the body of the loop, which is repeated. The control, however, is quite different. The loop control is achieved by the following sequence of events:

1. The declaration is established; it typically is a declaration of an `int` variable used to count the iterations. This variable is normally called the *loop variable*. This step is done just once, at the start, and never again for this execution of the loop.

2. The boolean expression is evaluated. If it is false, the loop terminates and control falls through to the next statement after the `for` loop. This boolean expression typically involves a comparison of the loop variable with some predetermined limiting value.

3. At this point the boolean expression is true, the statement (i.e. the loop body) is executed once.

4. The expression shown after the second semicolon in the `for` statement is evaluated. This expression typically involves an assignment to the loop variable.

5. Control returns to step 2 above to determine whether the loop body should be executed yet another time.

An example of a `for` loop which repeats the body exactly 10 times is shown below:

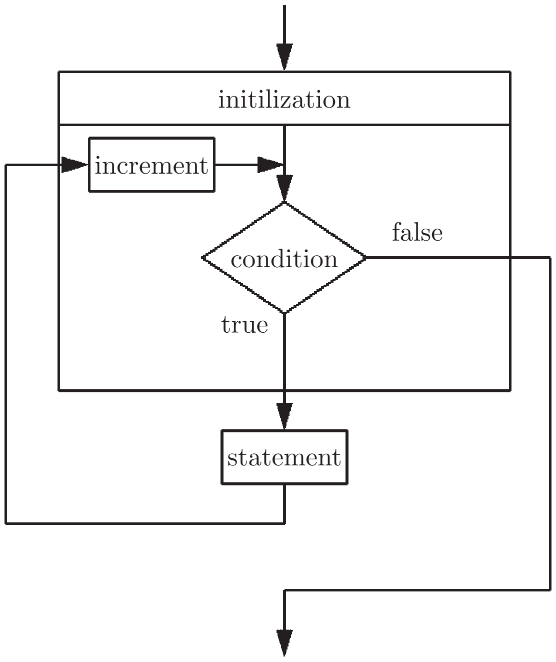


Figure 4.2: Flow diagram for a `for` loop structure

```
int sum = 0;        // declare and initialize an accumulator
for (int i=0; i<10; i++)
    sum = sum + i;
```

In this example when the loop terminates, the variable `sum` will contain the sum of the first 10 whole numbers (zero included). We note that:

- The loop variable is `i`. It is initialized to 0.
- The boolean expression `i<10` determines whether the body of the loop should be executed.
- The expression `i++` adds 1 to the value of `i` and stores the result back into `i`. It effectively increases `i` by one; we say that it *increments* `i`.
- In the loop body the statement `sum = sum+i` adds the value of `i` to the value of `sum` and stores the result back into `sum`.
- Since the loop variable is declared in the `for` statement, it is *local* to the loop. It is distinct from any variable of the same name declared outside the loop, and it will be disposed of when the loop terminates.

Figure 4.2 depicts the flow of control for a `for` loop.
Another example of a `for` loop is shown below.

```
int semesters = 7;
for (int sem=0; sem<7; sem++)
    totalCredits = totalCredits + getCredits(sem);
```

In this example it is known that the student attended for exactly 7 sememsters, so the loop should repeat exactly 7 times. Since the loop variable, `sem` is initially 0, and the loop continues to execute as long as the loop variable is *strictly* less than 7, the loop will repeat 7 times. Each time the loop repeats a method, `getCredits(int)`, is called; presumably that method returns the number of credits for a given semester (beginning with semester 0).

4.2.3 Exercises

1. Which of the following statments contain syntax errors?

   (a)
   ```
   for (int i=0; i<20; i++)
       System.out.println (i);
   ```

   (b)
   ```
   for (int i=0; i<20; i++)
       { System.out.println (i);  }
   ```

   (c)
   ```
   for (int i=0; i<20; i++)
       if (i > 10)
   ```

   (d)
   ```
   for (int i=0; i<20)
       System.out.println (i);
   ```

2. Show what would be printed by each of the following statements:

   (a)
   ```
   for (int i=0; i<5; i++)
       System.out.print (i-1 + " ");
   ```

   (b)
   ```
   for (int i=0; i<10; i++)
       if (i>5)
            System.out.print (i + " ");
   ```

   (c)
   ```
   for (int i=0; i<10; i++)
        System.out.print (i + " ");
        System.out.println ("again");
   ```

   (d)
   ```
   for (int i=0; i<10; i = i + 3)
      {
        System.out.print (i + " ");
        System.out.println ("again");
   ```

```
        }
```

3. Write a method named `showRange`, with two int parameters which will print on one line all the whole numbers in the given range. Use a `for` statement.

   ```
   /** Print all the whole numbers from low to hi, inclusive
    *  @param low The low end of the range.
    *  @param hi The high end of the range.
    */
   public void showRange (int low, int hi)
   ```

4. Write a method named `roots`, with one int parameter. It should print a table of whole numbers from 1 to the given parameter, showing the square root of each of those whole numbers. Use a `for` statement.

   ```
   /** Print a table of square roots for all whole numbers in the
    *  range 1..max, inclusive.
    *  @param max Should be positive
    */
   public void roots (int max)
   ```

5. (a) Write a method named `sumInts`, with one int parameter, max. It should return the sum of all the whole numbers from 1 through max, inclusive. Use a `for` statement.

   ```
   /** @return The sum of the whole numbers 1..max
    *  @param max Is positive
    */
   public int sumInts (int max)
   ```

   (b) Use the internet to find a simple algebraic formula to do the same calculation without using a loop.

4.3 Equivalence of `while` and `for` loops

Every `for` loop can be written as an equivalent `while` loop which does exactly the same thing. The `for` loops are provided as a feature of Java (and most other programming languages) merely as a convenience for the programmer. Figure 4.3 shows how any `for` loop can be rewritten as an equivalent `while` loop.

```
                                                init ;
for (init ; boolean expr ; expr)                while (boolean expr)
    body                                           {  body
                                                      expr ;
                                                   }
```

Figure 4.3: Equivalence of `for` and `while` loops

4.3.1 Exercises

1. Show a `while` loop which is equivalent to each of the following `for` loops:

   (a)
   ```
   for (int i=0; i<10; i++)
       System.out.println (i);
   ```

   (b)
   ```
   for (int i=17; i>=0; i = i - 1)
       System.out.println (i);
   ```

   (c)
   ```
   int i=3, x;
   for (x=10; i>=0; x--)
       System.out.println (i+x);
   ```

   (d)
   ```
   for (;i>=0; i--)
       System.out.println (i);
   ```

2. Show a `for` loop which is equivalent to each of the following `while` loops:

   (a)
   ```
   int i=0;
   while (i < 10)
    {
       System.out.println (i);
       i++;
    }
   ```

   (b)
   ```
   int i=0;
   while (i < 10)
    {
       System.out.println (i);
       i = i * 2;
    }
   ```

   (c)
   ```
   int i=0;
   while (i != 10)
   ```

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

4.4 Nested loops

In our original definition of the `while` and `for` loops we stated that the loop body consisted of a single statement. This statement is not necessarily an assignment statement; it could be an `if` statement or another `for` or `while` statement. The loop body could also be a compound statement containing any number of assignment, `if` `while` , and `for` statements. In other words the body of a loop can itself be a loop; we call this a *nested* loop.

An example of a nested loop is shown below:

```
int sumOfProducts = 0;
for (int i=1; i<10; i++)
   for (int j=1; j<10; j++)
       sumOfProducts = sumOfProducts + i*j;
```

In this example we add up the values

$1 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 + ... 1 \cdot 10 + 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 + ... 2 \cdot 10 + ... 10 \cdot 1 + 10 \cdot 2 + ... 10 \cdot 10$

There is another shortcut called *assignment operators.* For any operator $\circ$, the statement

$x = x \circ y$

is equivalent to

$x \circ = y$

This means that the statement shown above,

```
    sumOfProducts = sumOfProducts + i*j;
```

can be written more succinctly as

```
    sumOfProducts += i*j;
```

4.4.1 Exercises

1. Which of the following contain syntax errors?

   (a)
   ```
   int x = 3;
   while (x < 10)
       for (int i=0; i<5; i++)
           System.out.println (i+x);
   ```

   (b)
   ```
   int x = 3;
   while (x < 10)
     {
       for (int i=0; i<5; i++)
           System.out.println (i+x);
       x += 2;
      }
   ```

(c)
```
{
 for (int i=0; i<10; i++)
     for (int j=0; j<10; j++)
}
```

(d)
```
for (int i=0; i<10; i++)
    for (int i=0; i<10; i+)
      System.out.println ("i is " + i);
```

2. Show what would be printed by each of the following:

(a)
```
for (int row=0; row<2; row++)
  {
    System.out.println ("row " + row);
    for (int col=0; col<3; col++)
        System.out.print (col + " ");
    System.out.println ();
   }
```

(b)
```
int sum = 0;
while (sum < 20)
  {
    for (int i=0; i<4; i++)
        sum = sum + i;
    System.out.print (sum + " ");
   }
```

(c)
```
for (int row=0; row<5; row++)
  {
    for (int col=0; col<5; col++)
        System.out.print ("*");
    System.out.println ();
  }
```

(d)
```
int i=0, j=10
while (i+j < 12)
  while (i < 3)
     { j++;
       i++;
     }
System.out.println ("i is " + i ", j is " + j);
```

3. Define a method named `triangle` with one int parameter, which will print asterisks in the shape of a triangle with the given size. For example, if the size is 5, the output should look like this:

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

```
/** Print a triangular shape of asterisks
 *  @param size Number of rows, and length of base
 */
public void triangle (int size)
```

4. Repeat the previous problem, but make the triangle upside-down. Name the method `triangleInv`

5. Define a method named `mult` which will display a multiplication table of the given size.

```
/** Print a multiplication table
 *  @param size Number of rows and columns, must be positive.
 */
public void mult (int size)
```

4.5 Definition of Statement - updated

Now that we have seen iteration structures, we can update our formal definition of a Java statement:

A Java stmt may be:

1. variable = expression ;
2. if (boolean expression) stmt
3. if (boolean expression) stmt else stmt
4. method call (e.g. System.out.println ())
5. { stmt stmt stmt ... }
6. while (boolean expr) stmt
7. for (declaration ; boolean expr ; expr) stmt

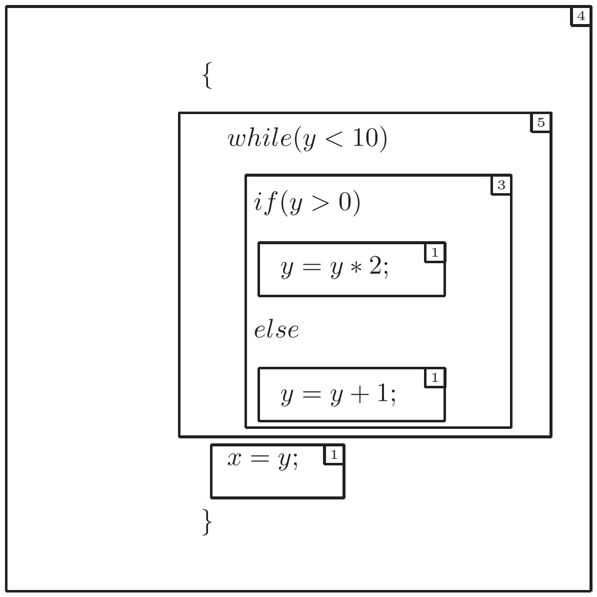


Figure 4.4: Applying the definition of Statement by drawing a box around each statement

As we have already noted our definition is recursive, and that allows statements to contain other statements, which in turn contain other statements, ... to any depth of containment levels.

Figure 4.4 shows how the definition of statement can be applied to the following:

```
{
   while ( y < 10 )
     if (y > 0)
        y = y * 2;
     else
        y = y + 1;
    x = y;
}
```

4.5.1 Exercises

1. Using the definition of a Java statement given in this section, draw a box around each statement shown below. Be sure to include the rule number in each box.

   (a)
   ```
   while ( i < 10)
       i = i+1;
   ```

(b)
```
while ( i < 10)
    if (x > 3)
        i = i+1;
```

(c)
```
if (x > 6)
   while (i < 10)
      System.out.println (i);
else
   { x = x - 1;
     System.out.println (x);
   }
```

4.6 Iterations in the GridWorld case study

4.7 Projects

1. (a) Rewrite the `fib` method from chapter 3 using a loop instead of a recursive method. `fib(n)` should return the n^{th} number in the fibonacci sequence.
   (b) Which appears to run faster, the looping version of fib or the recursive version?
   (c) What is fib(15)? fib(100)?
   (d) How can you explain the result of fib(100)?

2. A *prime* number is a whole number greater than 1, which is divisible only by 1 and itself. Some examples of prime numbers are: 2, 3, 5, 7, 11, 13, ... Define a method named `isPrime`. It should have one parameter, an int, and should return a boolean – true if the parameter is a prime number.

   ```
   /** @return true only if x is prime
    *  @param x is positive
    */
   public boolean isPrime (int x)
   ```

   Hint: Use the mod operator(%) to determine whether the given number is divisible by some other number.

3. The Taylor series expansion of the function *sin(x)* is shown below:

   $sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$

   Define a Java method to return the sin of a given value, to within a given tolerance.

```
/** @return sin(x)
 *  @param epsilon is the tolerance.
 */
public double sin (double x, double epsilon)
```

4. Write a method named `inBinary` which will return the binary representation of a non-negative int as a String of 1's and 0's. For example if the parameter value is 18, it should return the String `"10010"`.

   ```
   /** @return The binary representation of x as a String
    *  of 0's and 1's.
    *  @param x is not negative.
    */
   public String inBinary (int x)
   ```

   Hint: Work from right to left in the binary representation of the given number. The mod operator (%) can tell you whether the low order bit is 0 or 1. The integer divide operator (/) can remove that bit.

5. Write a method to find an approximation to the square root of a double which is at least 1.0, to within a given accuracy, using a *bisection* algorithm:

   (a) Call the parameter x. Establish low and high boundaries for the result; low is 0 and hi is x.

   (b) Repeat the following as long as the result is not sufficiently accurate.

      i. An approximation to the result will be the average of low and hi.

      ii. If this approximation is too high, assign it to hi.

      iii. If this approximation is too low, assign it to low.

   ```
   /** @return an approximation to square root of x
    *  @param x is at least 1.0
    *  @param epsilon is tolerance for correct result
    */
   public double sqrt(double x, double epsilon)
   ```

Chapter 5

Collections, and Iteration Revisited

Thus far we have written programs which deal with small quantities of data. Each variable stores a single value, or a reference to a single object. However, most applications have a need to deal with large quantities of data; we do not wish to define a new variable for each data item that we need to store. Consequently we desire the capability of defining one variable which stores a reference to a *collection* of data items. These collections can be organized in a multitude of different ways, depending on how we wish to optimize the time required to access a particular value. In this chapter we examine a few different ways of organizing, and working with, collections of data items. We also begin a discussion of time efficiency, comparing the relative speeds of various operations.

5.1 Lists

A *list*, in mathematics and computer science, is defined as a collection of items with the following properties:

- Items may be added to, and removed from, a list; its size may be changed.
- The items have a particular order - the order in which they have been added to the list (though it may be possible to insert an item in the middle). The ordering of the items in a list is maintained.
- Each element has an *index* or position number associated with it. The index of the first item in the list is 0 (as with the positions of the characters in a String).
- There may be duplicated items in a list (some other collections do not have this property).

- Lists, as with other kinds of collections, are *homogeneous* – all items in a list are of the same type (we'll see considerable flexibility here, however, after we discuss inheritance).

The efficiency, or time required, for the operations of accessing, adding, removing items can vary depending on the particular implementation of the list; we shall discuss a few different implementations, noting their relative advantages and disadvantages.

The Java class library provides us with several kinds of lists. We'll begin by considering a particular implemenation called *ArrayList*.

5.1.1 Java packages and java.util

There are thousands of classes in the Java class library. Fortunately they are grouped and organized in such a way that we can deal with the ones we need and not worry about the others. A group of classes which share a common purpose or use can be grouped into a *package*. Packages, in turn, with similar purposes may be grouped together, forming a recursive hierarchy of packages. The package which deals with lists is named *java.util*. In order to use any of the classes in this package, those classes must be specified in an *import* statement at the beginning of your program. For convenience, we can have access to all classes in that package as follows:

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

5.1.2 ArrayList

The ArrayList class can be used if it is imported from java.util using the import statement given above.

5.1.2.1 Declaring and instantiating an ArrayList

There are actually a few different kinds of lists in the Java class library, though we are limiting our discussion to ArrayLists. To declare a variable which is capable of storing a reference to a list of Students, we should declare it as follows:

```
List <Student> roster;
```

This means that the variable `roster` can store a reference to any kind of list, including ArrayLists [1]. The value of the variable `roster` at this point is a *null reference*. It is a reference which refers to nothing at all; we have not yet even created the ArrayList. Because of the word `Student`, in angle brackets, this variable is capable of storing a reference to a List which stores references to Students *only*. This satsifies our requirement that all lists must be homogeneous.

In order to *instantiate* (i.e. to create an ArrayList instantiation instance of) the ArrayList we will use the `new` operator (as we did with the `Student` class):

[1] Technically, List is an interface which we cover in chapter 6, but since we believe in establishing good habits early on, we use it here

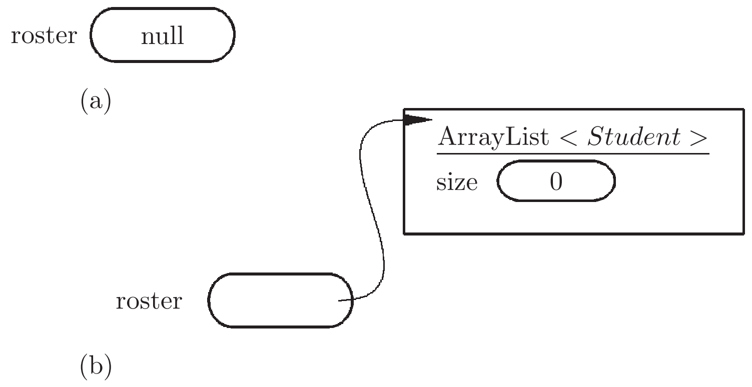


Figure 5.1: An object diagram showing the value of the variable `roster` (a) when it is declared and (b) after it has been assigned a value

```
roster = new ArrayList <Student>();
```

Notice that:

- Once again we specify the kind of objects (Student, in this case) to be stored in the ArrayList being created (recent versions of Java have relaxed this requirement).
- When we instantiate a list, we must specify what kind of list it is, in this case ArrayList.
- At this point the variable `roster` is no longer storing a null reference.

We now have an ArrayList containing zero items; its *size* is 0.

Notice also that Figure 5.1 depicts the process of declaring the variable `roster` and instantiating the ArrayList with an object diagram. In an ArrayList object we always show the size of the ArrayList as an int field, and the size is initially 0.

5.1.2.2 Adding or inserting items to an ArrayList

Once an ArrayList has been instantiated, items can be added using the *add* method. To add an item at the end of the list, supply one parameter, the item to be added:

```
Student s;
s = new Student ("joe", "256", 3.5);
roster.add (s);
```

This will add the new Student to the ArrayList referred to by `roster`. Note that lists, and all collections, do *NOT* store objects; they store *references* to objects as shown in the object diagram in Figure 5.2. Every time we add another

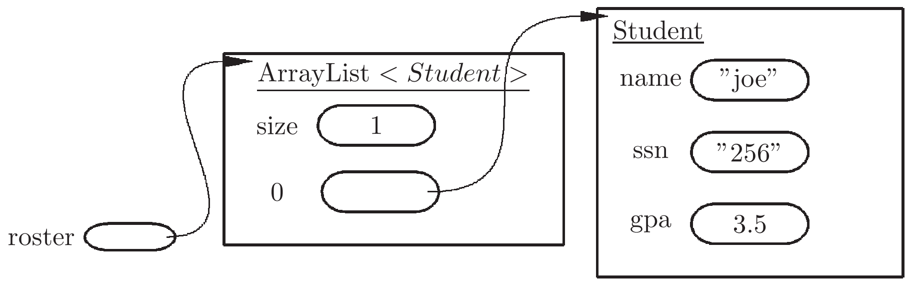


Figure 5.2: An object diagram showing the value of the variable `roster` after a Student has been added

student to the ArrayList, another reference is included in the ArrayList, and its size is automatically updated. Since we haven't looked at the source code for the ArrayList class, we'll simply make up our own field names and draw the diagram accordingly. For each reference added to the ArrayList, we show its position number (0 in Figure 5.2).

Figure 5.3 shows the object diagram for `roster` after a total of three students have been added to it.

Items can also be inserted at any position in an ArrayList, using the same method, but with an additional parameter. The first parameter is an int specifying the index, or position, at which the item is to be added. This form of the `add` method is really an *insertion.* The example below shows how a new Student can be added (inserted) at position 1:

```
roster.add (1,new Student ("joe", "304", 3.5));
```

Since it is not possible to have two items at the same position, all items with larger indices are automatically given higher index numbers. In other words, when adding at position 1 to a list of 3 items, the items formerly at positions 1 and 2 will be now be at positions 2 and 3.

Care needs to be taken when using this form of the `add` method. The index, or position, of the inserted item must be less than or equal to the current size of the list. If the index is negative, or greater than the size, a *run-time error* will occur, causing your program to come to a crashing halt. To insert an item at the beginning of the list, use an index of 0, and to insert at the end you can use an index equal to the current size (but its easier, and less risky, to use the `add` method with only one parameter).

Incidentally, this is our first example of two different methods with the same name. That's ok, as long as they have different parameter lists. They can be distinguished in a method call by the number (and types) of parameters. However, you cannot use differing return types to define two methods with the same name and same parameter types.

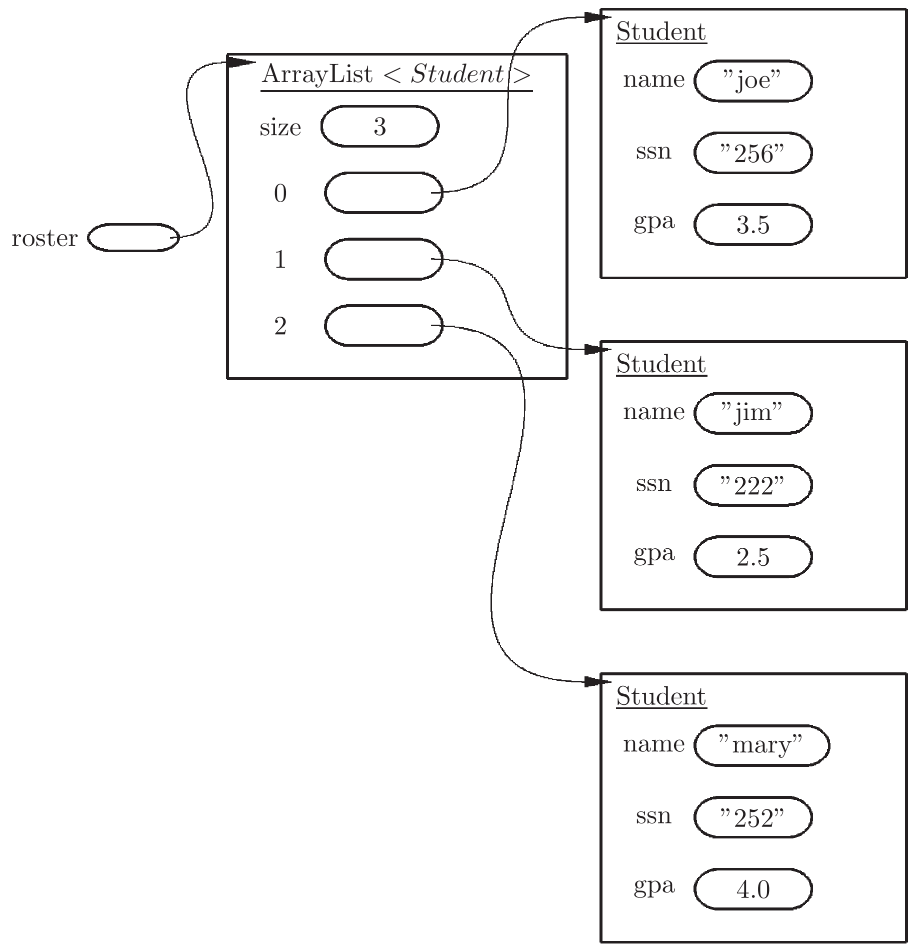


Figure 5.3: An object diagram showing the value of the variable `roster` after three Students have been added

5.1.2.3 Accessing or changing items in a list – get and set

After instantiating an ArrayList and perhaps adding several items to it, we may wish to discover the current value of some item in the list, or we may wisht to change the reference at a particular position. These operations are accomplished with the methods `get` and `set`.

To access the item at position 4 in our list, we could use the `get` method as shown below:

```
Student s;
s = roster.get(4);
```

This presumes that there are at least 5 Students in the list (remember, the first one is at position 0). The `get` method returns the reference which is at position 4, and the assignment stores that reference into the variable `s`. Again, we need to be careful that the given index is in the correct range: 0..size-1, otherwise a run-time error will occur.

The `get` method API is shown below:

```
/**
 *   @param ndx The position of the item to be returned,
 *              must be non-negative and less than size.
 *   @return The reference at the given index.
 */
E  get (int ndx)
```

The notation `E` in the code shown above is supposed to stand for the type of the items in the ArrayList. If it is an ArrayList of Students, E represents `Student`.

The `set` method is used to *store an item* at a given position in a list. For example,

```
Student s = new Student ("jim", "323", 3.2);
roster.set(12,s);
```

This code will replace the reference at position 12 with a reference to the new Student, jim, at that position. The indices of other items in the list will be unchanged. The compiler will insist that the type of the second parameter matches the type specified when creating the ArrayList, `Student` in this case.

The first parameter for both `get` and `set` must be an int in the range 0..size-1, otherwise a run-time error will result.

Note that there is an important difference between `set(int,E)` and `add(int,E)`. The `set` method does not change the size of the list, but the `add` method always increases the size of the list by one.

Primitive type	Wrapper class
int	Integer
float	Float
double	Double
boolean	Boolean
char	Character

Figure 5.4: Wrapper classes for primitive types

5.1.2.4 Collections of primitives

We've seen that collections, such as ArrayLists, store references to objects. But suppose we wish to store a collection of primitives, such as ints, floats, doubles, booleans, or chars? Java provides a 'workaround' for this situation, called *autoboxing* and *autounboxing*, using *wrapper* classes. The wrapper classes for the primitive types are shown in Figure 5.4. Each wrapper class is a simple class whose only purpose is to store a primitive value of the corresponding type. For example, an Integer object has a field with stores an int value, a Float object has a field which stores a float object, etc. When you wish to create a collection of primitives, use the wrapper class instead. For example, the code shown below will create an ArrayList to which int values can be added:

```
List <Integer> grades = new ArrayList <Integer>();
grades.add(100);
grades.add(90);
grades.add(75);
grades.add(100);
int first = grades.get(0);
System.out.println ("The first grade is " + first);
```

Note that when we add a primitive value (an int) to the ArrayList, Java automatically converts it to the corresponding wrapper class (Integer) so that the reference to it can be stored in the ArrayList. This is called *autoboxing*. Also, the `get` method returns a reference to the wrapper class (Integer), but we can store it into an int. This is called *autounboxing*.

A simpler way of restating the above paragraph is that you can have collections of primitives, but use the name of the corresponding wrapper class (shown in Figure 5.4) when declaring the collection.

We can now create a list of the first ten multiples of 10, using a `for` loop quite easily:

```
List<Integer> multiplesOf10;
multiplesOf10 = new ArrayList <Integer> ();
for (int i=0; i<10; i = i + 1)
   multiplesOf10.add (i*10);
```

5.1.2.5 A note on wrapper classes

Wrapper classes also provide other information on the corresponding primitive type. For example the `Integer` class has class variables `MAX_VALUE` and `MIN_VALUE` which store the largest possible int and the smallest possible int, respectively. Since they are class variables, the variable is preceded by the class name:

```
Integer.MAX_VALUE
Integer.MIN_VALUE
```

The wrapper classes are in the `java.lang` package, and do not need to be imported. See the API for more information on these wrapper classes.

5.1.2.6 Other operations on lists

The Java class library provides several other operations on lists; we will discuss a few of them here. For a complete list of operations see the API for `List` or `ArrayList` at `docs.oracle.com/javase/7/docs/api`.

We can *remove* the item at a given position from a list using the method shown below:

```
/** Removes the item at the given index from this list.
  * @param ndx is a valid index for this ArrayList,
  * ndx is not negative, and ndx is less than the size of this ArrayList.
  * @return a reference to the item removed.
  */
E  remove (int ndx)
```

As an example, we could remove the item at position 7 from a list named roster, and store the reference to the removed item in a variable:

```
Student s;
s = roster.remove(7);
// s now stores a reference to the removed student
```

The indices of all subsequent items in the list are decremented by one (no 'gap' is left in the list), and the size is also decremented by one. The `remove` method returns a value, but if we are interested in the removal only, and don't need the item that was removed, we can ignore the value returned:

```
roster.remove(7);
// the size of the list is decreased by 1.
```

In general, the value returned by a method can be used as part of an expression, or it can be ignored. If ignored, we say that we are using the method only for its *side effects* - in this case the removal of an item.

We can obtain the *size* of a list:

```
/**
  * @return the size of this list.
  */
int size()
```

We could print the size of a list as shown below:
`System.out.println ("We have " + roster.size() + " students.");`
We could use the `size()` method to determine whether a list is empty (if its size is 0), or we could use the *isEmpty* method:

```
/**
  * @return true only if this list is empty.
  */
boolean isEmpty()
```

The `isEmpty` method could be used as shown:

```
if (roster.isEmpty())
    System.out.println ("No students enrolled");
```

The `isEmpty` method returns a boolean, and therefore it can be used where a boolean expression is expected, as in an `if`, `while`, or `for` statement.
Many novice programmers would code the example shown above as:

```
if (roster.isEmpty() == true)
   ...
```

This would work, but we advise *against* this form. [2]
We can print out an entire list, simply be giving its name:
`System.out.println (students);`
All items in the list will be printed on one line, separated by commas, and enclosed in square brackets.

5.1.3 Exercises

1. Which of the following have syntax errors, and which will result in run-time errors (assume that the ArrayList class has been imported)?

   (a)
   ```
   List <Student> kids;
   kids = new Student();
   ```

   (b)
   ```
   List <Student> kids;
   kids = new ArrayList<Student>();
   kids.add ("jim");
   ```

[2] if the example had been `boolean b; if (b == true)...` then if the == operator is mistakenly typed as = (meaning assignment instead of comparision), the compiler will accept it, but the program will not work correctly, and many hours of debugging effort may be required to find the error.

(c)
```
List <Student> kids = null;
kids.add (new Student ("jim", "234"));
```

(d)
```
List <Student> kids;
kids = new ArrayList<Student>();
kids.add (new Student ("jim", "234"));
```

2. Show what would be printed by each of the following (assume the ArrayList class has been imported):

   (a)
   ```
   List <Student> kids = null;
   System.out.printlin (kids);
   kids  = new ArrayList <Student> ();
   System.out.printlin (kids);
   ```

   (b)
   ```
   List <Student> kids  = new ArrayList <Student> ();
   kids.add (new Student ("joe", "222"));
   kids.add (new Student ("jim", "333"));
   System.out.printlin (kids.size() + " kids");
   ```

   (c)
   ```
   List <Student> kids  = new ArrayList <Student> ();
   kids.add (new Student ("joe", "222"));
   kids.add (new Student ("jim", "333"));
   System.out.println (kids.get(0).getName());
   kids.remove(0);
   System.out.println (kids.get(0).getName());
   ```

   (d)
   ```
   List <Student> kids, roster;
   kids  = new ArrayList <Student> ();
   roster = kids;
   kids.add (new Student ("joe", "222"));
   kids.add (new Student ("jim", "333"));
   System.out.println ("size of roster is " + roster.size());
   ```

3. If `roster` is storing a reference to a list of at least 10 Students,

   (a) show how to print the name of the student at position 8 in the list.

   (b) show how to insert a new Student whose name is ”alice” and whose ssn is ”234” at the end of the list.

   (c) show how to insert a new Student whose name is ”alice” and whose ssn is ”234” at the beginning of the list.

   (d) show how to insert a new Student whose name is ”alice” and whose ssn is ”234” at position 3 in the list.

(e) show how to change position 7 of the list to refer to a new Student whose name is ”jim” and whose ssn is ”321”.

(f) show how to remove the Student at position 7, and print that student’s name, using just one statement.

4. Draw an object diagram showing the values of the variables num1, num2, num3, and num4 after the code shown below has executed:

```
List <Integer> num1, num2, num3;
num1 = new ArrayList <Integer> ();
num2 = num1;
num3 = new ArrayList <Integer> ();
num1.add (17)
num2.add (3);
```

5.2 Iteration revisited, with lists

Now that we have discussed loops and lists, we will see how to use a loop in conjunction with a list. This is a very common construct in programs: We have a list, or some other kind of collection, and we wish to ’visit’ each item in the list, perhaps to examine it for certain properties, print it in a certain way, or even possibly to change it. Whatever the reason for visiting each item may be, it can be done easily with a loop, and a `for` loop is usually most amenable for this purpose.

For example, if the name of our list is `roster`, its original declaration might be:

```
List <Student> roster;
```

Suppose this list has been instantiated as an ArrayList, and many Student objects have been added to it, and we wish to print the name of each student. The loop can be controlled with a `for` statement as follows:

```
Student s;
for (int i=0; i<roster.size(); i++)
  { s = roster.get(i);
    System.out.println (s.getName());
  }
```

Note that:

- We use a local variable, `s`, to store a reference to a Student obtained from the list with the `get` method.
- The loop variable is initialized to 0, the position of the first item in the list.
- The loop will be terminated when the loop variable is *equal* to the size of the list (the last item in the list is at positon size-1).

- The loop variable is incremented by 1 each time the loop repeats
- The body of the loop is a compound statement in which we obtain a reference to a list item and print the name of the Student object to which it refers.

As a second example, we show a complete method which will determine whether a given list has at least one student with a perfect 4.0 gpa. This method uses a `while` loop, but it could also have been done with a `for` loop.

```
/**
 *  @return true only if there is a perfect student in the
 *  given list, students.
 */
public boolean hasAperfectStudent (List <Student> students)
{
   int i=0;                          // position of first item
   while (i<students.size())
     {  s = students.get(i);
        if (s.getGPA() == 4.0)
           return true;              // terminate the method
     }
   return false;                     // no perfect students found
}
```

In coding this method we are careful to make sure that it always returns the correct result, but we are also careful to make sure that it doesn't waste time needlessly. As soon as one student with a 4.0 gpa is encountered, we know the result should be `true`, and we terminate the method with `return true`. There is no need to continue looping. For small lists, the distinction may not be significant; even if we continue looping after finding a perfect student, it may take a small fraction of a second to complete its work. However, for large lists the time could be significant, and if the call to the method is itself inside a loop, the time could be very significant. We encourage the programmer to think about efficiency even at this early stage in learning to program.

If the loop runs to completion (all students in the list have been visited), we know that there could not have been any perfect students in the list, and the result should be `false`. In this case we terminate the method with `return false`.

5.2.1 Exercises

1. Define a method named `perfect` with one parameter, a list of Students. It should return the number of students in the list who have a perfect gpa of 4.0.

   ```
   /** @return The number of students who have a 4.0 gpa
   ```

```
 *  @param students Is not null
 */
public int perfect (List <Student> students)
```

2. Define a method named `average` with one parameter, a list of Integers. It should return the average of those integers, as a double.

```
/** @return The average of the given integers
 *  @param numbers Is not null and is not empty
 */
public int average (List <Integer> numbers)
```

3. Define a method named `sameName` with two parameters which will return a new list of students consisting of all those in the given list who have the same name as the second parameter.

```
/** @param students Is not null
 *  @param name Is the name to be matched
 *  @return List of all students with the given name
 */
public List <Student> sameName (List <Student> students,
                                String name)
```

5.3 Sets

We now discuss a different kind of collection known as the *set*. Sets are different from lists in two important aspects:

- The items in a set are not maintained in any particular order. The order in which items are obtained from the set may be completely different from the order in which they were added. The items could conceivably be reordered at any time.
- There are no duplicate items in a set. When an item is added to a set, if an item with the same value is already in the set, the size of the set is not changed. The new item is not added.

This means there will be no indices, or position numbers, for the items in a set. As with lists, a set will have a size - the number of items in the set, and sets must be homogeneous - the items in a set must all be of the same type.

The Java class library provides a few implementations of sets; we will limit our discussion to the one called *HashSet*. Like `ArrayList`, this class must be imported from `java.util`.

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

We will declare a variable to be of type `Set`, without specifying what kind of Set it may be. When instantiating it, however, we must specify the kind of Set, such as HashSet. We could create a set of whole numbers, and add several values to it:

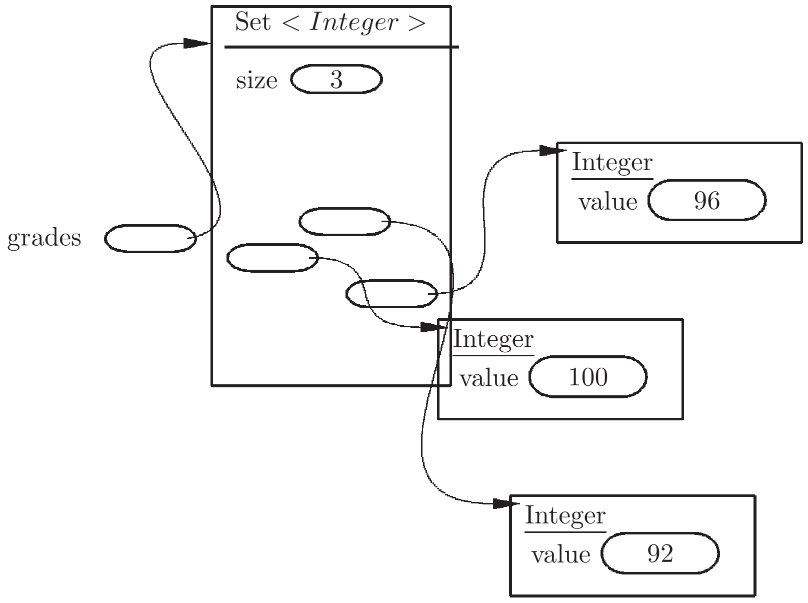


Figure 5.5: An object diagram showing a set of Integers after three items have been added. The items are stored in no particular order.

```
Set <Integer> grades;
grades = new HashSet <Integer> ();
grades.add (92);
grades.add (96);
grades.add (92);
grades.add (100);
grades.add (92);
```

At this point the size of the set would be 3 (the value 92 is added only once). In the next section we will see how we can obtain each item from a set, in a loop, possibly to print the values; in this case we would have no control over the order in which the values are printed, for example - 100,96,92. The implementation of the set controls the order. Figure 5.5 shows an object diagram for the set named `grades` created in the code segment shown above. Since we have not seen the source code for the `HashSet` class, we will simply show the references in a set in a nonlinear fashion, with no position numbers, to imply the lack of order in a set. As with ArrayLists, we will always show the size of the set, even if the size is 0.

Object diagrams can become rather large and complex. To alleviate this we will allow a slight 'shortcut': String objects and objects of wrapper classes (see Figure 5.4) may be treated as primitives; i.e. for a variable whose type is String, Integer, Double, Character, etc. we are permitted to show the value directly in the oval box, rather than as a reference to an object (which it really is). This

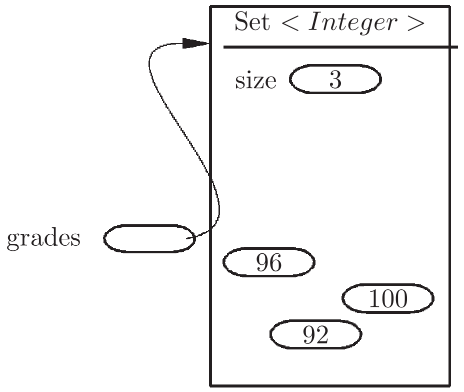


Figure 5.6: A simplified object diagram showing a set after three items have been added. Since the type of the items is Integer (a wrapper for int), they can be treated like primitives.

shortcut is possible only because the String class and the wrapper classes are *immutable*; i.e., it is not possible to change objects of these classes once they have been created. Figure 5.6 shows a simpler version of the object diagram in Figure 5.5.

Since sets do not have position numbers, the question may arise: How can we access individual items from a set – there are no `get` nor `set` methods? We will defer the answer to this quesion until the section on *iterators*. However, we can create a String representation of all the items in a set, and print it if we wish, very easily:

```
System.out.println ("The grades are " + grades);
```

This will convert each item in the set named `grades` to a String, concatenate them all together, separated by commas, into one long String inside square brackets. This is a nice feature of the HashSet class (and most other classes in the Java class library), and we will see how to provide the same feature for classes that we build ourselves.

One final remark with regard to sets is in order. If we were to create a set of Students, or some other class which we have defined ourselves, a few methods are required in that class: `equals(Object)` and (for HashSets) `hashCode()`. We will cover these in a later chapter. These methods are already provided in the String, Integer, and other classes in the Java Class Library, so they work fine.

5.3.1 Exercises

1. Which of the following contain syntax errors and which contain run-time errors (assume all classes from java.util have been imported)?

   (a)
   ```
   Set <Integer> numbers = new Set <Integer> ();
   numbers.add(18);
   ```

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

(b)
```
Set <Integer> numbers = new HashSet <Integer> ();
numbers.add(18);
numbers.add(3);
System.out.println (numbers.get(1));
```

(c)
```
Set <Integer> numbers = new HashSet <Integer> ();
numbers.add(18);
numbers.add(3);
System.out.println (numbers);
```

(d)
```
Set <Integer> numbers = new HashSet <Integer> ();
numbers.add(18);
numbers.add(3);
numbers.remove(0);
System.out.println (numbers);
```

2. Show what would be printed by each of the following (assume all classes from java.util have been imported):

(a)
```
Set <Student> others, kids = new Set<Student> ();
others = kids;
kids.add (new Student ("jim", "321"));
kids.add (new Student ("joe", "333"));
System.out.println ("size is " + others.size());
```

(b)
```
Set <Integer> numbers;
for (int i=0; i<100; i++)
   numbers.add (i/10);
System.out.println ("size is " + numbers.size());
```

(c)
```
Set <Student> others, kids = new Set<Student> ();
kids.add (new Student ("jim", "321"));
kids = others;
System.out.println (kids);
```

3. Define a method named `copyToSet` with one parameter, a List of Students, which will return a Set of all the Students in the given List.

```
/** @return a Set of all Students in the List students.
 *  @param students, May be empty but not null.
 */
public Set <Student> copyToSet (List <Student> students)
```

5.4 Iteration through a collection with for-each, and extrema problems

The need to visit every item in a collection is very common in computer programs. As we have seen in a previous section, this can be done with a `while` or `for` loop when working with a list.

5.4.1 Iteration through a collection with for-each

In this section we introduce an easier, and better, way to visit every item in a collection. It is called a *for-each* loop (though the word *each* does not appear anywhere in this construct). The general format is:

```
for (type variable : collection)
    statement                     // loop body
```

An example is:

```
for (Student s: roster)
    System.out.println ("The student's name is " + s.getName());
```

Here we are assuming that `roster` is a list of Students. The loop control will ensure that the variable, s, will be assigned the next item in the ArrayList roster each time the loop repeats, beginning with the first Student in the list and ending with the last student in the list. If we read the loop control as: "for each Student s, in roster", even though the words 'each' and 'in' appear nowhere in the code, we have a fairly good understanding of the intent.

We now see how to obtain a reference to each item in a set. We can use a for-each loop (with sets there is no other way since sets do not have indices). The following example is a method with one parameter, a set of integers, which will return the average value of those integers, as a double.

```
/** @return the average value of the given numbers
 *  @param numbers A Set of Integers to be averaged.
 */
double average (Set <Integer> numbers)
{  int sum = 0;
   for (int i : numbers)
       sum = sum + i;                       // accumulate the sum
   return sum / (double) numerss.size();    // divide by size to get average
}
```

In the example above, we accumulate the sum of the numbers in an accumulator, `sum`, in the loop body. When the loop terminates, we calculate the average by dividing the sum by the number of numbers. Since we wish the division to be a floating point division, we cast the result of the size() method to a double, ensuring a floating point division, and return the result of the division.

We should take a little more care, however, in this example. It is always a mistake to divide by 0, even if the dividend is 0. In this example the size of the set could be 0 (the API doesn't specify that the set must not be empty). Therefore we need to check for an empty set before doing the division. Also, the API should warn us what the result will be if the parameter is an empty set. Here is an improved version of the above example:

```
/** @return the average value of the given numbers, or 0 if the
 *  the set is empty.
 *  @param numbers A Set of Integers to be averaged.
 */
double average (Set <Integer> numbers)
{  int sum = 0;
   if (numbers.isEmpty())
       return 0;                          // terminate the method
   for (int i : numbers)
       sum = sum + i;                     // accumulate the sum
   return sum / (double) nums.size();    // divide by size to get average
}
```

The API now clarifies what will be returned if the given set is empty, and there is no possibility of dividing by 0.

When working with sets, it is necessary to use a for-each loop rather than `while` or `for` loop. Moreover, we encourage the programmer to use for-each loops whenever possible, even if not required, for a few reasons:

- As you become familiar with for-each loops, it becomes clear that the syntax and logic is much more simple and clear than when using a `while` or `for` loop.
- We will see later that there are other list classes in addition to ArrayList; in this case for-each loops can be much more efficient.

For-each loops are easy and convenient; however they have one important drawback. The size of the collection involved cannot be changed in the body of the loop. This means that items cannot be added or removed.

5.4.2 Extrema problems

One very common task we will encounter in programming is the problem of finding the largest (or smallest) of a collection of values. This is called an *extremum* (singular) or *extema* (plural) problem. We suggest the follow strategy to find the smallest value in a list of values (with some simple modifications it can be used to find the largest):

1. Check to see whether the collection is empty, and handle this as a special case.

2. Initialize a local variable to the value of the first item in the collection. We will call this a ‘candidate’ for the smallest; it is the smallest we have seen thus far.

3. Set up a loop in which each item in the collection is visited.

4. If a visited item is smaller than the candidate, then the new value of the candidate should be the value of the visited item. This is now the smallest we have seen thus far.

5. When the loop terminates, we have visited all items in the collection, and the candidate will be the smallest

Here is a code segment in which we try to find the smallest number in an ArrayList of ints:

```
List <Integer> grades = new ArrayList <Integer> ();
// Assume the ArrayList grades has been assigned values
//  ....
int smallest;
if (!grades.isEmpty())             // Check size of the list
   {  smallest = grades.get(0);    // there is at least one number
      for (int grade : grades)     // for each grade
        if (grade < smallest)
          smallest = grade;        // new candidate for smallest
   }
// smallest grade is stored in the variable ‘smallest’.
```

For our next example, we will define a method which will return a reference to the best student in a given set of students. Using the same strategy as the previous example, we examine the gpa of each student, maintaining a candidate for the best as we cycle through the loop:

```
/**  @return the student with the highest gpa from the given Set of
  *  students, or null if the set is empty.
  */
public Student getBest (Set <Student> school)
{  Student best = null;             // candidate for best student
   for (Student st : school)        // for-each
     {
       if (best == null)            // first student in the set
          best = st;                // is the best seen so far
       if (st.getGPA() > best.getGPA())
          best = st;                // new candidate for best student
     }
   return best;
}
```

In chapter 2 we discussed null references in connection with object diagrams, but in this example we are making use of a null reference to indicate that the method could not return anything meaningful in the case where it is given an empty set. This is a very common usage of a null reference; it is a reference which refers to no object at all. Note that the local variable `best` is initialized to a null reference. If there are no students in the given set, the loop will not execute, not even once, and the value of `best` will be null when the return statement is executed. Also note that we can compare a variable against `null` with an == comparison, as though we are comparing primitives (we are actually comparing references). The logic in the example above is slightly different from the logic used in the previous example, because sets have no `get(int)` method. If you define a method which may return a null reference, be sure to clarify in the API under what circumstances this will actually occur.

5.4.3 Exercises

1. Which of the following contain syntax errors?

   (a)
   ```
   Set <Integer> roster = new HashSet <Integer> ();
   for (Student st : roster)
       System.out.println (st);
   ```

   (b)
   ```
   Set <Student> roster = new HashSet <Student> ();
   for (int i=0; i < roster.size(); i++)
       System.out.println (roster.get(i));
   ```

   (c)
   ```
   List <Student> roster = new ArrayList <Student> ();
   for (Student st : roster)
       System.out.println (st);
   ```

   (d)
   ```
   Set <Student> roster = new HashSet <Student> ();
   for (Student st : roster)
       System.out.println (st);
   ```

2. Define a method named `showPositive` with one parameter, a Set of Integers, which will print all the values in that set which are positive.

   ```
   /** Print all numbers which are positive.
    */
   public void showPositive (Set <Integer> numbers)
   ```

3. Define a method named `trueBits` with one parameter, a List of Booleans, which will return the number of Booleans in the list which are true. Use a for-each loop.

```
/** @return The number of true values in the parameter, bits.
 */
public int trueBits (List <Boolean> bits)
```

4. Define a method named `longestName`, with one parameter, a Set of Students. It should return the name of the Student in the given list who has the longest name, or null if the list is empty.

```
/** @return The longest name of all students in roster, or null
 *  if roster is an empty list.
 */
public String longestName (Set <Student> roster)
```

5. Define a method named `smallestPositive`, with one parameter, a List of Doubles. It should return the smallest positive number in the given list, or null if the list is empty. Use a for-each loop.

```
/** @return The smallest positive number in the given list,
 *  or null if the list is empty.
 */
public Double smallestPositive (List <Double> numbers)
```

5.5 Iterators and selective removal from a collection

If you have been looking at the API for `ArrayList` and `HashSet`, you may have noticed that there are several *remove* methods available, which enable you to remove an item from a collection. These methods can be used if you read the API carefully. However, we may wish to remove some of the items from a collection as we visit all items in a loop. In this case we *cannot* use a for-each loop, since a removal would change the size of the collection. Instead we will use an *Iterator*.

5.5.1 Iterators

An Iterator is a class[3] in the Java class library which is designed to help you visit all items in a collection. An Iterator object needs to be obtained, after which it can be used to:

- Obtain the next item from the collection. The method signature is `E Next()`. It returns the next item in the collection (the return type, `E`, must match the type of the items in the collection).

[3]Iterator is actually an *interface* which is similar to a class; we will discuss interfaces in chapter 6.

- Check to see if there are more items in the collection. The method signature is `boolean hasNext()`. It returns `true` only if there are more items in the collection which have not yet been visited since the Iterator object was created.
- Remove the last item obtained from the collection. The method signature is `void remove()`. This is the best way to selectively remove items from a collection

.

Since Iterator is in the package java.util, it must be imported at the beginning of your source file:
`import java.util.Iterator`
One unusual aspect of Iterators is that they cannot be instantiated with the `new` operator; rather we use a method from one of our collection classes called `iterator()` which returns an instance of the Iterator class. Once we have the Iterator object, we can use it to control the loop as shown in the following example, in which we print the names of all students in the ArrayList `roster`:

```
Student s;
Iterator <Student> itty;        // itty is an Iterator object
itty = roster.iterator();       // instantiate with a method call
while (itty.hasNext())          // are there more students in the list?
   {  s = itty.next();          // yes, get the next student in the list
      System.out.println (s.getName());
   }
                                // All student names have been printed
```

Notice that we must specify the type of the iterator in angle brackets – it must match the type of the collection with which it is associated, in this case, Student. This is called a *generic* type. The above example could have been done (and probably should have been done - for clarity) with a for-each loop. It also could have been done with a `for` loop, using the `get(int)` method. However, as mentioned previously, there are other list classes, in addition to `ArrayList` which do not have efficient implementations of the `get` and `set` methods. For this reason, we are advised to use a for-each loop or an Iterator when possible. The next example, however, points out a stronger need for Iterators.

5.5.2 Selective removal

Often we wish to visit every item in a collection, and change the collection by removing some of the items as we cycle through the collection. In the following example, we are writing a method which is supposed to remove all the failing students from a given list of students.

```
/** Remove all students with gpa under 1.0 from the given list
 */
```

```
public void flunkOut (List <Student> roster)
{  Iterator <Student> it;
   it = roster.iterator();
   while (it.hasNext())
      {  s = it.next();
         if (s.getGPA() < 1.0)
            it.remove();          // the iterator does the remove
      }
}
```

In this example we are removing some of the students, and not removing others; we call this *removing selectively*. We could *not* have used a for-each loop since we are changing the size of the list. Conceivably, we could have done this without using an Iterator, but the logic is so convoluted and difficult that you would probably not get it correct; we will not even show you how to do this – instead use an Iterator.

Notice in this example that we *use the iterator to remove*; this is easy to forget. The `remove()` method in the Iterator class will remove the last item obtained by a call to `next`. The `remove()` method can be called only once per call to `next`.

5.5.3 Exercises

1. Which of the following contain syntax errors, and which contain run-time errors?

(a)
```
Set <Student> school = new Set <Student> ();
school.add (new Student ("Jim", ""));
for (Student st : school)
    if (st.getSSN().length() == 0)
        st.remove();
```

(b)
```
Set <Student> school = new Set <Student> ();
school.add (new Student ("Jim", ""));
Iterator <Student> it;
it = school.iterator();
while (it.hasNext())
  { Student st = it.next();
    if (st.getSSN().length() == 0)
       remove(st);
  }
```

(c)
```
Set <Student> school = new Set <Student> ();
school.add (new Student ("Jim", ""));
Iterator <Student> it;
it = new Iterator (school);
```

```
while (it.hasNext())
  { Student st = it.next();
    if (st.getSSN().length() == 0)
       it.remove();
  }
```

(d)
```
Set <Student> school = new Set <Student> ();
Iterator it;
it = school.iterator();
while (it.hasNext())
  { Student st = it.next();
    if (st.getSSN().length() == 0)
       it.remove();
  }
```

2. Define a method named `retrieveStudent` with two parameters, a Set of Students, and a String representing an ssn. The method should return the Student in the set whose ssn matches the given ssn. If no students match the given ssn, the method should return a null reference. Use an Iterator to control the loop.

```
/** @return Any Student whose ssn matches the parameter ssn,
 *  or null if no such Student is found in the list.
 */
public Student retrieveStudent (Set <student> students, String ssn)
```

3. Define a method named `removeNegative` which will remove all the negative values from a given list of Integers.

```
/** Remove all negative values from nums.
 */
public void removeNegative (List <Integer> nums)
```

4. Define a method named `removeByName` with two parameters, a List of Students, and a student's name. It should remove all students with the given name from the List.

```
/** Remove all students with name given as parameter.
 */
public void removeByName (List <Student> roster, String name)
```

5. Consider the following method to print Students who have a GPA of 3.0 or greater:

```
/** Print students with GPA of at least 3.0
```

```
 */
public void showDeansList (List <Student> students)
{  Iterator <Student> itty = students.iterator();
   while (itty.hasNext())
      if (itty.next().getGPA() >= 3.0)
         System.out.println (itty.next() + " is on the Dean's List");
}
```

(a) Point out a subtle logic error in this method.

(b) Give an example of a list of students, for which this method will cause a run-time error.

(c) Give an example of a list of students, for which this method will not crash, but will produce incorrect output.

(d) Give an example of a non-empty list of students, for which this method will not crash, but will produce correct output.

(e) Show how this error can be corrected.

5.6 Arrays

The notion of ArrayList is actually a fairly recent invention in programming languages (late 1990's); ArrayLists are built upon a more primitive kind of collection called an *array*. The array dates back to the earliest days of programming languages; arrays provide a means for mapping the items in a collection directly to the computer's memory, for quick access to any item. Aside from some very different syntax in the usage of arrays, they differ from ArrayLists in the following ways:

- The size of an array cannot change, whereas an ArrayList can grow and shrink as a program executes.
- Arrays can be used without involving the Java class library (though there are classes designed to be used with arrays).
- No `import` statement is needed to use arrays.

To declare a variable as an array, use the following format:

```
type [] variable;
```

This means that the variable being declared does not store a single item, but a reference to an array of items, each of the same type. Then we can instantiate the array; it is at this point that we determine, once and for all, the size (or length) of the array:

```
variable = new type [length];
```

As an example we can create an array of ints, with length 12 as follows:

```
int [] grades;
grades = new int [12];
```

The length of the array can be determined when the program is run, but once the array is created, its length may not be changed:

```
int len;
len = getLength();    // get the length from a method call
int [] grades;
grades = new int [len];
```

The type of an array need not be a primitive type; we could create an array of 23 Student objects as shown below:

```
Student [] roster;
roster = new Student [23];
```

In this example each position in the array is storing a reference to a Student object. How would these Student objects be initialized? We would need to provide the Student class with a constructor that has no parameters, a *default constructor*:

```
// Default constructor for Student class
public Student ()
{  name = "";
   ssn  = "";
   gpa  = 0.0;
}
```

Once we have created the array, we can store a value of the appropriate type into any position of the array. As usual the position numbers begin at 0. The position of the last item in the array is length-1. This is done using square brackets, and the syntax is:

```
array-name [index] = expression;
```

For example:

```
grades[2] = 95;
```

This would mean that the value 95 is stored into position 2 (third item) of the array. Figure 5.7 shows an object diagram for the variable `grades`. Notice that since it is an array of ints, positions which have not been assigned an explicit value are given a default value of 0 (Warning: other programming languages might not be so kind as to do this for us).

To access a value from an array, again put the index in square brackets:

```
System.out.println ("The fourth grade is " + grades[3]);
```

Note that the index in square brackets can be any expression which evaluates

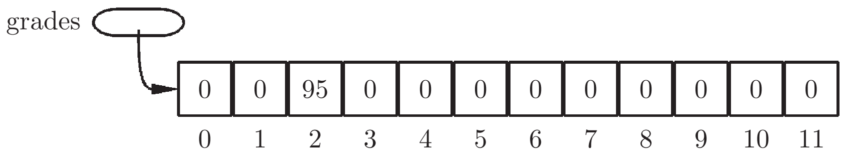


Figure 5.7: An array of ints, showing values and position numbers.

to an int. Be careful that the index is in the range 0..length-1, otherwise a run-time exception, `ArrayIndexOutOfBoundsException`, will cause your program to come to a crashing halt.[4] We have been using the word *length* instead of *size* with respect to arrays, because the length of an array can be obtained using the *field* name `length` (it is not a method name). An example which finds the average value of numbers in an array of ints is shown below:

```
// assume the array has been assigned values
int sum = 0;
double average;
for (int i=0; i<grades.length; i++)
   sum = sum + grades[i];
if (grades.length > 0)
   average = sum / (double) grades.length;
```

Because of the flexibility afforded by ArrayLists (they can grow and shrink), we will be using ArrayLists rather than arrays for most of our work. There are a few reasons, however, that programmers need to be aware of arrays:

- The ArrayList class is built using arrays; to understand how the class works one really needs to understand arrays.
- Some methods in the Java Class Library work with arrays rather than ArrayLists.
- When working as maintenance programmers, particularly when working with older, or so-called *legacy*, code, we will encounter arrays and should be able to read and understand the code.
- Arrays are closely associated with the hardware; arrays are mapped directly to the computer's memory. When you study *computer organization* or *computer architecture* you will gain a greater appreciation for arrays.

5.6.1 Initialization of arrays

Java offers a convenient way to initialize the elements of an array. The declaration can be followed by a list of items in curly braces, and separated by commas. Example:

[4]Other languages, such as C/C++ do not check for proper array indices; hence, they are vulnerable to *buffer over-run* attacks.

```
   int []nums = {4, 83, -5, 0};
```
The length of the array is determined by the number of initial values in curly braces, 4 in this example.

An array of Strings can also be initialized in this way:
```
   String [] names = {"joe", "jim", "sally"};
```
The length of this array is 3.

5.6.2 Passing arrays as parameters

A reference to an array may be passed into a method. As usual it is just the *reference* which is passed, not the entire array. This means that if the called method makes a change to the array, the calling method will see this update in the array; i.e. there are not two separate copies of the array.

To pass an array as a parameter, simply provide the name of the array as the actual parameter in the calling method, The formal parameter in the called method should include the type and empty brackets to show that it is an array, as shown in the example below.

```
   void callingMethod ()
   {   int [] numbers = new int[100];
       ...
       calledMethod(numbers);
       // numbers[2] is now -3
   }
 ...
   void calledMethod (int [] nums)
   {  nums[2] = -3;
   }
```

Note that, as usual, the name of the actual parameter (`numbers` in this case) does not have to be the same as the name of the formal parameter (`nums` in this case).

5.6.3 Vector product of numeric arrays

Arrays are often referred to as *vectors*. To find the vector product of two arrays which have the same length we simply multiply corresponding elements and sum the products. For example:

```
  [2 5 7] x [3 -1 2] = 2 x 3 + 5 x -1 + 7 x 2
                     = 6 + -5 + 14
                     = 15
```

A method to find the vector product of two arrays is shown below:

```
/** @return the vector product of two arrays of int
 *  @param a1 and a2 have the same length
```

```
*/
int vectorProduct(int [] a1, int [] a2)
{  int [] result = new int [a1.length];
   int sum = 0;
   for (int i=0; i<a1.length; i++)
      sum = sum + a1[i]*a2[i];
   return sum;
```

5.6.4 Exercises

1. Which of the following contain syntax errors and which contain run-time errors?

   (a)
   ```
   int grades;
   grades = new int [15];
   grades[2] = 90;
   grades[3] = 10;
   grades[4] = grades[2] + grades[3];
   grades[grades[3]] = 17;
   ```

   (b)
   ```
   int [ ] grades;
   grades[2] = 90;
   grades[3] = 10;
   grades[4] = grades[2] + grades[3];
   grades[grades[3]] = 17;
   ```

   (c)
   ```
   int [ ] grades;
   grades = new int [15];
   grades[2] = 90;
   grades[3] = 10;
   grades[4] = grades[2] + grades[3];
   grades[grades[2]] = 17;
   ```

   (d)
   ```
   int [ ] grades;
   grades = new int [15];
   grades[2] = 90;
   grades[3] = 10;
   grades[4] = grades[2] + grades[3];
   grades[grades[3]] = 17;
   ```

2.
   ```
   int [] nums = new int [10];
   for (int i=0; i<10; i++)
      nums[i] = i * 10;
   ```

   Assuming the code shown above has been executed, show what would be printed by each of the following:

$$\begin{pmatrix} 4 & 8 & -3 & 0 & 4 \\ 0 & 0 & 3 & 0 & 2 \\ 14 & 0 & -3 & 1 & 1 \end{pmatrix}$$

Figure 5.8: An example of a matrix of numbers with 3 rows and 5 columns

(a)
```
for (int i=0; i<10; i++)
    System.out.print (nums[i] + " ");
```

(b)
```
for (int i=1; i<10; i++)
    System.out.print (nums[i] + nums[i-1] + " ");
```

(c)
```
for (int i=1; i<10; i++)
    nums[i-1] = nums[i];
System.out.println (nums[3]);
```

(d)
```
for (int i=1; i<10; i++)
    nums[i] = nums[i-1];
System.out.println (nums[3]);
```

3. Given an array of Students named `students`, show the code which could find the average GPA for those students.
4. Given an array of Students named `students`, show the code which could be used to find the position of the Student with the highest GPA.
5. Given an array of ints named `fib`, show the code which could be used to fill that array with the numbers in the fibonacci sequence.

5.7 Matrices: Two Dimensional Arrays

In mathematics we define a matrix to be an arrangement of values into rows and columns, in which all rows have the same number of values (and all columns have the same number of values).[5] Fig 5.8 shows a matrix of numbers, as you might see it in a mathematics textbook.

In Java we can implement matrices as two-dimensional arrays. To declare a two-dimensional array of ints, use two pairs of square brackets:
`int [][] myMatrix;`
Then to instantiate the array, specify the number of rows in the first pair of brackets, and the number of columns[6] in the second pair of brackets. For ex-

[5]In Java this restriction does not apply, as the number of columns in each row may vary, but we will not be concerned with this capability.

[6]Alternatively, one could think of the first dimension as the number of columns, and the second dimension as the number of rows.

ample, to work with a matrix of ints with 3 rows and 5 columns:
`myMatrix = new int[3][5];`

As with one-dimensional arrays, two-dimensional arrays can be declared and initialized in one statment (and the sizes need not be specified):

```
int [] [] myMatrix = {{2,3,4},
                      {7,2,0}};
```

In this example the array has 2 rows and 3 columns.

To access a particular value from a two-dimensional array, provide integer expressions for both the row and column:
`myMatrix[0][1]`
For this example, the value would be 3.

To change a value in a two dimensional array, one also needs to provide integer expressions for the row and column numbers:
`myMatrix[1][0] = 19;`
In this example, the 7 would be clobbered, and replaced by 19.

5.7.1 Examples of Matrix Arithmetic

5.7.1.1 Multiplication by a scalar

As an example of arithmetic involving two-dimensional arrays, we discuss the multiplication of a two-dimensional array of numbers by a single number. The single number, mathematically, is called a *scalar*. The product of a matrix multiplied by a scalar is simply a matrix of the same dimensions in which each element is multiplied by the scalar. For example:

$$\begin{pmatrix} 4 & 8 & -3 & 0 & 4 \\ 0 & 0 & 3 & 0 & 2 \\ 14 & 0 & -3 & 1 & 1 \end{pmatrix} \times 3 = \begin{pmatrix} 12 & 24 & -9 & 0 & 12 \\ 0 & 0 & 9 & 0 & 6 \\ 42 & 0 & -9 & 3 & 3 \end{pmatrix}$$

To do this in Java, we will use a nested loop. The outer loop will repeat once for each row in the matrix, and the inner loop will repeat once for each column in a row. In the inner loop we multiply each element of the given matrix by the given scalar to produce the corresponding element in the result. A method to multiply a matrix by a scalar is shown below:

```
/** @return the product resulting when the given matrix is
 *  multiplied by the given scalar.
 *  @param matrix is not empty
 */
public static int[][] multByScalar(int[][]matrix, int scalar)
{  int rows = matrix.length;
   int cols = matrix[0].length;
   int[][]result = new int[rows][cols];
```

```
        for (int row=0; row<rows; row++)            // outer loop
             for (int col=0; col<cols; col++)       // inner loop
                 result[row][col] = matrix[row][col] * scalar;
        return result;
    }
```

In this method, note that to obtain the number of rows and columns in the given matrix, we use the `length` variable:
`rows = matrix.length;`
`cols = matrix[0].length;`
Also note that the result matrix must have the same dimensions as the given matrix

5.7.1.2 Addition of matrices

To add matrices we simply add corresponding elements of the two matrices to produce the sum matrix. The two matrices being added must have the same dimensions. Below we show the mathematical sum of two matrices:

$$\begin{pmatrix} 4 & 8 & -3 & 0 & 4 \\ 0 & 0 & 3 & 0 & 2 \\ 14 & 0 & -3 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 0 & -4 & 3 & 2 & -1 \\ 99 & 0 & -9 & 0 & 0 \\ 14 & 1 & 9 & 3 & 2 \end{pmatrix} = \begin{pmatrix} 4 & 4 & 0 & 2 & 3 \\ 99 & 0 & -6 & 0 & 2 \\ 28 & 1 & 6 & 4 & 3 \end{pmatrix}$$

To calculate a matrix sum in Java we use a nested loop, as in the previous section on multiplication by a scalar. In the body of the inner loop, we add corresponding elements of the two given matrices, to produce one element of the result matrix. A Java method to add two (non-empty) matrices is shown below:

```
    /** @return the matrix sum of the two given matrices.
     *  @param The given matrices have the same dimensions, and
     *          neither of the given matrices is empty.
     */
    public static int[][] add (int[][] m1, int[][] m2)
    {   int rows = m1.length;
        int cols = m1[0].length;

        int[][]result = new int[rows][cols];
         for (int row=0; row<rows; row++)
            for (int col=0; col<cols; col++)
                result[row][col] = m1[row][col] + m2[row][col];
         return result;
    }
```

5.7.1.3 Multiplication of matrices

Matrix multiplication is a bit more complicated than the other operations we have discussed. In order to multiply two matrices they must be *conformable*, i.e.

Dimensions of A	Dimensions of B	Dimensions of $A \times B$
`A[2][3]`	`B[3][5]`	`Product[2][5]`
`A[7][4]`	`B[4][9]`	`Product[7][9]`
`A[2][3]`	`B[5][3]`	Not conformable
`A[3][3]`	`B[3][3]`	`Product[3][3]`

Figure 5.9: Matrices must be conformable in order to be multiplied

$$\begin{pmatrix} \hline 2 & 5 & 3 \\ \hline 3 & 0 & 7 \\ 2 & 0 & 3 \\ 1 & 1 & 4 \\ 6 & 6 & 6 \end{pmatrix} \times \left(\begin{array}{|c|c} 1 & 3 \\ 2 & 1 \\ 4 & 2 \end{array}\right) = \begin{pmatrix} 2 \cdot 1 + 5 \cdot 2 + 3 \cdot 4 = 24 & . \\ . & . \\ . & . \\ . & . \\ . & . \end{pmatrix}$$

Figure 5.10: Calculation of the value at row 0, column 0 in the multiplication of two matrices

they must have the correct dimentsions. If A and B are matrices, and we are to find the matrix product $A \times B$, then the number of columns in A must equal the number of rows in B. The number of rows in the result would equal the number rows in A, and the number of columns in the result would equal the number of rows in B. Fig 5.9 shows the dimensions of the result for several examples of matrix multiplication. Note that matrix multiplication is not commutative; i.e. $A \times B = B \times A$ is not always true.

We now explain how to find the matrix product $A \times B$, assuming the matrices are conformable. To find the value of row 0, column 0, in the product, we use the vector product of row 0 of A with column 0 of B, as shown in Fig 5.10

To find the value at row r, column c, of the product we use the vector product of row r in A with column c in B. The complete product of the two matrices shown in Fig 5.10 is shown in Fig 5.11.

A Java method to multiply nonEmpty matrices is shown below. We use a loop within a loop within a loop to calculate the product.

```
/** @return the matrix product of the two given matrices.
*  @param The number of columns in m1 must equal the number
*         of rows in m2.
*  @param Neither of the given matrices are empty.
*/
```

$$\begin{pmatrix} 2 & 5 & 3 \\ 3 & 0 & 7 \\ 2 & 0 & 3 \\ 1 & 1 & 4 \\ 6 & 6 & 6 \end{pmatrix} \times \begin{pmatrix} 1 & 3 \\ 2 & 1 \\ 4 & 2 \end{pmatrix} = \begin{pmatrix} 24 & 17 \\ 31 & 23 \\ 14 & 12 \\ 19 & 12 \\ 42 & 36 \end{pmatrix}$$

Figure 5.11: Multiplication of matrices

```
public static int[][] mult (int[][] m1, int[][] m2)
{   int rows = m1.length;              // rows in the result
    int cols = m2[0].length;           // columns in the result
    int n = m2.length;                 // conforming dimension
    int[][]result = new int[rows][cols];

     for (int row=0; row<rows; row++)
        for (int col=0; col<cols; col++)
            { int sum = 0;
              for (int i=0; i<n; i++)  // find vector product
                  sum = sum + m1[row][i] * m2[i][col];
              result[row][col] = sum;
            }
     return result;
}
```

Matrix multiplication has many applications in simulations, weather forecasting, statistics, economics, and engineering. Researchers are always looking for fast ways of multiplying huge matrices.

5.7.2 Exercises

1. Given the matrix, `m`, show the value of each expression shown below:

   ```
   int [][] m = {{2,5,7},
                 {3,0,-2}};
   ```

   (a) `m[1][0]`

   (b) `m[0][1] - m[1,2]`

   (c) `m[m[1][0] + m[1][2]][1]`

2. Using the matrix given in the previous problem, show the matrix (as in Fig 5.8) after each of the following statements has executed.

   (a) `m[0][1] = 17;`

   (b) `m[0][1] = m[0][0];`

   (c) `m[0][m[1][1]] = m[0][0];`

3. Given the matrix, `names`, show the value of each expression shown below:

   ```
   String[][] names = {{"sal","jim"},
                       {"flo","Joe"},
                       {"ann","sal"}};
   ```

   (a) `names[1][1]`

   (b) `names[0][1] + names[1][0]`

(c) `(names[1][1] + names[2][0]).charAt(4)`

4. Given the following matrices, show the result of each operation shown below, if possible:

$$A = \begin{pmatrix} 4 & 8 & -3 & 0 & 4 \\ 0 & 0 & 3 & 0 & 2 \\ 14 & 0 & -3 & 1 & 1 \end{pmatrix} B = \begin{pmatrix} 2 & 5 & 0 & 2 & -1 \\ 9 & 0 & -9 & 0 & 0 \\ 4 & 3 & -3 & 1 & 1 \end{pmatrix} C = \begin{pmatrix} 1 & 3 \\ 3 & 1 \\ 2 & 2 \\ 0 & -1 \\ -2 & 5 \end{pmatrix}$$

   (a) $A \cdot 3$
   (b) $A + B$
   (c) $A + C$
   (d) $A \times B$
   (e) $B \times C$
   (f) $C \times B$

5. Write a java method to find the sum of a scalar plus a matrix. Simply add the scalar to each element of the matrix:

```
/** @return the sum of the given matrix and the given
 *          scalar.
 */
int[][] addScalar (int[][] m, int scalar)
{   . . .  }
```

6. Given the java code shown below, show what would be printed.

```
int m[][] = new int[3][4];
for (int r=0; r<3; r++)
   for (int c=0; c<4; c++)
       m[r][c] = r+c;

System.out.println (m[1][1]);
for (int r=0; r<3; r++)
   for (int c=0; c<4; c++)
       m[r][c] = m[r][(c+1)%4];

System.out.println (m[1][1]);
System.out.println (m[0][3]);
```

7. A Latin square is a square matrix of numbers in which:

   - The first row has no duplicate values.

- All values in the first row of the square appear in each row of the square.
- All values in the first row of the square appear in each column of the square.

Implement the methods shown in the API given below in the class `ArrayTester`.[7]

```
/** Exercise on two dimensional arrays
 *  @author ...
 *  @version (Feb 2019)
 */
public class ArrayTester
{
    /**@return an array containing the elements of column c of
     * arr2D in the same order
     * Pre:  c is a valid column index in arr2D
     * Post: arr2D is unchanged
     */
    public static int[] getColumn (int [][]arr2D, int c)
    {
        // put your solution here
    }

    /** @return true iff every value in arr1 is also in arr2
     *  Pre:  arr1 and arr2 have the same length.
     *  Post: arr1 and arr2 are unchanged.
     */
    public static boolean hasAllValues(int[] arr1, int[] arr2)
    {
        // put your solution here
    }

    /** @return true iff arr contains any duplicates */
    public static boolean containsDuplicates (int[] arr)
    {
        // put your solution here
    }

    /**  @return true iff the given matrix is a Latin Square
     *   @param:  square has equal number of rows and columns.
     *            square has at least one row.
     */
    public static boolean isLatin (int [][] square)
    {
```

[7]This was a free-response question on the 2018 Advanced Placement CS exam.

```
        // put your solution here
      }
  }
```

5.8 Collections in the GridWorld case study

5.9 Projects

1. Implement the following Set operations:

   (a) Define a method named `union` which has two parameters, each of which is a Set of Strings. The method should return a new set consisting of all Strings which occur in either, or both, of the given sets.

   ```
   /** @return a new set which is the union of set1 and set2.
    */
   public Set <String> union (Set <String> set1, Set <String> set2)
   ```

   (b) Define a method named `intersection` which has two parameters, each of which is a Set of Strings. The method should return a new set consisting of all Strings which occur in both of the given sets.

   ```
   /** @return a new set which is the intersection of set1 and set2.
    */
   public Set <String> intersection (Set <String> set1, Set <String> set2)
   ```

   (c) Define a method named `difference` which has two parameters, each of which is a Set of Strings. The method should return a new set consisting of all Strings which occur in the first set, but not in the second set.

   ```
   /** @return a new set which is the difference, set1 - set2.
    */
   public Set <String> difference (Set <String> set1, Set <String> set2)
   ```

   (d) Define a method named `concat` which has two parameters, each of which is a Set of Strings. The method should return a new set consisting of all Strings which result from concatenating each String in the first set with each String in the second set. For example, if the two sets are: {`jim, tom, john`} and {`my, son`}. The result would be {`jimmmy, jimson, tommy, tomson, johnmy, johnson`}.

   ```
   /** @return a new set which is the concatenation of set1 with set2.
    */
   public Set <String> concat (Set <String> set1, Set <String> set2)
   ```

2. Poker.

(a) Define a class named `Card` with two fields, rank and suit, both of which are Strings. Include public accessor methods for these fields. Also define a method named `toString()` which returns a String representing a Card. For example, if the rank is `"Jack"` and the suit is `"hearts"`, this method would return `"Jack of hearts"`.

(b) Define a class named `Deck` with at least one field, a List of Cards. The constructor should initialize the field to the 52 different cards in a deck of playing cards. The suits are `"spades", "hearts", "diamonds", "clubs"`. The ranks are `"Two", "Three", ... "King", "Ace"`.

Define a method named `getCard` with one parameter, an int representing the position of a card in the deck. This method should remove the card at the given position from this deck, and return the removed card.

```
/** Remove the Card at position ndx from this Deck.
 *  @return the removed Card
 *  @param ndx is not negative, and less than the size of this Deck.
 */
public Card getCard(int ndx)
```

Define a method which returns the size of this Deck (i.e. the number of cards currently in this Deck).

(c) Define a class named `Poker` with at least one field storing a Deck. This class should have a method named `dealHand(int n)` which will return a List of n cards from its deck. Those cards should also be removed from the deck, in case `dealHand` is called more than once.

```
   /** Deal a poker hand.
    * @return n cards from the deck.
    * These cards are removed from the deck.
    */
   public List <Card> dealHand (int n)
```

Help: To deal cards randomly from the deck, use a random number generator from java.util, Random:

`Random rand = new Random();`

Each time you call the `nextInt(int n)` method, it will return a random int in the range [0..n-1].

Write a method to test your work by dealing 4 Poker hands, with 5 cards in each hand (there should be no duplicate cards).

(d) Change the Deck class so that it uses an array of Cards instead of a List of Cards. You should not need to change any other classes because the API for the Deck class is not changing, only the implementation is changing. This is an example of *object abstraction* which is discussed in chapter 6.

3. Sorting a list of numbers

   (a) Define a class named `Sorter`. The purpose of this class will be to store a List of numbers, and to arrange them in increasing order. This is called *sorting* and is one of the most important applications of computers. This class shold have one field, a list of Doubles.

   (b) Include a constructor which will initialize the list of Doubles, perhaps using a random number generator (see previous project).

   (c) Define a private method named `swap` with two int parameters. This method will exchange the values in the given positions of the array of Doubles.

```
/** Exchange positions j and k in the list of numbers
 */
private void swap (int j, int k)
```

   (d) Define a method named `posSmallest` with one parameter. It should return the position of the smallest value in the list, beginning at the given start position.

```
/** @return The positon of the smallest value in numbers,
 *  beginning at position start.
 */
private int posSmallest (int start)
```

   (e) Define a method named `sort`. It should arrange the values in the list in ascending order. It can do this with one easy loop in which it calls posSmallest and swap. For each position, p, in the list, swap it with the position of the smallest from p to the end of the list.

```
/** Sort the list of numbers in ascending order
 */
public void sort ()
```

   This project describes an *algorithm* known as *selection sort*. If you continue to study computer science, you will learn many other (sorting) algorithms.

4. Build a simulation for weather forecasts. Use several two dimensional arrays (all of which have the same dimensions):

   - A matrix in which each cell stores the temperature, in degrees Fahrenheit, at that location in the simulation
   - A matrix in which each cell stores the barometric pressure, in mm of mercury, at that location in the simulation
   - A matrix in which each cell stores the relative humidity, as a percentage, at that location in the simulation

Your simulation should update these arrays in a series of steps, enabling you to predict the air temperature, pressure, and humidity at some location any time in the future.

Work on the assumption that if the pressure is lower in a cell, than in a neighboring cell, that will cause air to flow from the high pressure cell to the low pressure cell. This will effectively change the pressure, temperature, and humidity in both cells. The extent to which these things change depends on how much air move, which is determined by the difference in pressure in the two cells.

Chapter 6

Abstraction, Inheritance, and Polymorphism

In this chapter we return to the process of *class design*. We deal with the question, How can we design classes which which work correctly and are easy to use and maintain? We also deal with issues related to duplicated code (undesirable), code reuse (desirable), encapsulation (desirable), and object-oriented design of software.

6.1 Software engineering

In the early days of software development the immense complexity of software was not well understood. People assumed that by putting enough programmers on a project, and by testing the software which they produced, a large and reliable software system could be produced in a reasonable time. However, this was not the case; there were many failed projects, and many projects were not reliable or were otherwise faulty. In cases where the software performed adequately, it was rarely delivered on time and within budget. In the meantime huge advances have been made in the field of computer hardware. The speed and memory capacity of computers have both improved at an amazing rate, along with improved reliability of hardware.

It has been said that if the automotive industry had accomplished technical developments on par with the developments of the computer hardware industry, a Rolls-Royce would have a top speed of 200 miles per hour, fuel economy of 500 miles per gallon, and would cost $13.50.

When your own personal computer crashes, it is almost always the result of a software error, and rarely the result of a hardware failure. Why has the field of computer software lagged so far behind the field of computer hardware? We feel it is largely due to the complexity of software. In order to deal with this problem, a discipline known as *Software Engineering* was established to apply engineering principles to the design and development of software. This

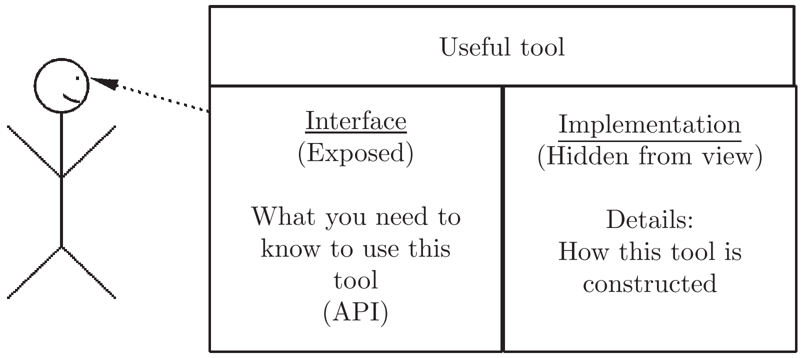


Figure 6.1: Abstraction: There is no need to know the implementation details of the tools being used.

chapter is essentially an introduction to software engineering, and in particular, object-oriented software engineering.

6.2 Abstraction

Because of the complexity of software, it is important that we be able to deal with a portion of a software system without having a detailed understanding of how the complete system works. Our goal will be to build tools (the Java Libary is a good example) which can be used to build other useful tools, which in turn are used to build other tools.... In the process we would like to be able to use a tool without worrying about its internal details – all we need to know is what the tool is supposed to do for us, and how can we use it properly. This is an example of *abstraction* and is depicted in Figure 6.1. Abstraction, in a more general sense, is the process of separating ideas from specific instances of those ideas.

6.2.1 Duplicated code

Suppose there is a segment of code in our program which seems to serve a useful purpose, and we find a need for this code segment in several other places in the program. For example, we are finding the average GPA of a set of Students:

```
// roster is a set of students
double average;
int sum = 0;
for (Student s : roster)
   sum = sum + s.getGPA();
average = sum / (double) roster.size();
```

Assume we have tested this code and it seems to be correct. Now we discover other places in the program where we need to find the average GPA of the students in roster. It would be easy to copy and paste this code where it is needed.

Alternatively, we could write a method to accomplish this task, and simply call the method when needed. Clearly, that would make our program shorter (fewer lines of code), but memory and storage are cheap – this is not a problem (also, some programmers get paid by the line of code produced).

Suppose that rather than writing a method to find the average, we have copied and pasted this code segment in over 50 different places in our program. We now learn that sometimes the set of students, roster, might be empty. The code that we have introduced will not work when the set of students is empty, because we would divide by 0 to calculate the average gpa – this is a bug. By using copy and paste we have introduced over 50 bugs in our program. In many cases this cannot be fixed with a simple editor command to find and replace. Instead the programmer will have to search and find every place this code segment was used and correct it manually. This is even more of a problem if the faulty code had been pasted in many different source files. This is a serious problem resulting from *duplicated code*.

Now consider the alternate strategy; instead of using copy and paste, we define a method to find the average GPA:

```
/** @return the average gpa of the given set of students
 */
public double average (Set <Student> roster)
{
  int sum = 0;
  for (Student s : roster)
    sum = sum + s.getGPA();
  return sum / (double) roster.size();
}
```

We need to correct the mistake in *one place only*, the method which calculates average GPA. The corrected version is shown below:

```
/** @return the average gpa of the given set of students
  *  or 0.0 if roster is an empty set.
 */
public double average (Set <Student> roster)
{
  int sum = 0;
  if (roster.isEmpty())              // avoid division by 0
     return 0.0;

  for (Student s : roster)
    sum = sum + s.getGPA();

  return sum / (double) roster.size();
}
```

In every place where we need to find the average GPA we simply call the method which returns the average. This has two clear advantages over the copy and paste strategy:

- When the bug surfaces, we need to make the correction in *only one place*:
- It is now easy to find the average GPA for *any* set of students, not just the one named `roster`:

  `double avg = average (someRoster);`

This example points out the potentially disastrous pitfalls that can result from duplicated code. Try to avoid duplicated code if at all possible. The avoidance of duplicated code is one example of an abstraction.

6.2.2 Method abstraction

If we were to view the API for the `average` method in the previous section, without looking at the method body, this would be another example of abstraction, which we call *method abstraction* and is also called *control abstraction*. The API tells us how to use the method and what it returns; there is no need to look at the details of how it works.

Another example of method abstraction would involve using methods to build other methods. A generalized version of this concept would be the building of *software tools* to be used in the construction of other software tools. This idea is depicted in Figure 6.2.

As an example we consider the problem of arranging a list of items in correct order. For example, we may wish to arrange the items in a particular subsequence of a list in ascending order (a more general version of this problem is called *sorting* and was introduced as a project in chapter 5). In other words if we are given the list [4,0,9,2,1,6,-2] and we wish to arrange the sequence at positions 2,3,4 in order, the list would become: [4,0,1,2,9,6,-2].

We now define a method to arrange an arbitrary subsequence of length 3 in ascending order. Here is version 1:

```
/** Arrange the 3 items in the given list, at positions start, start+1,
 *  start +2 in ascending order.
 *  @param start The first position of the subsequence to be arranged
 *  order; start must not be negative and start must be less
 *  than numbers.size()-2.
 *  @param numbers A list of numbers, size is at least 3.
 */
public void sort3(ArrayList <Integer> numbers, int start)
{  int tmp;
   if (numbers.get(start) > numbers.get(start+1))
       {  tmp = numbers.get(start);       // swap first and second
          numbers.set(start, numbers.get(start+1));
          numbers.set(start+1, tmp);
```

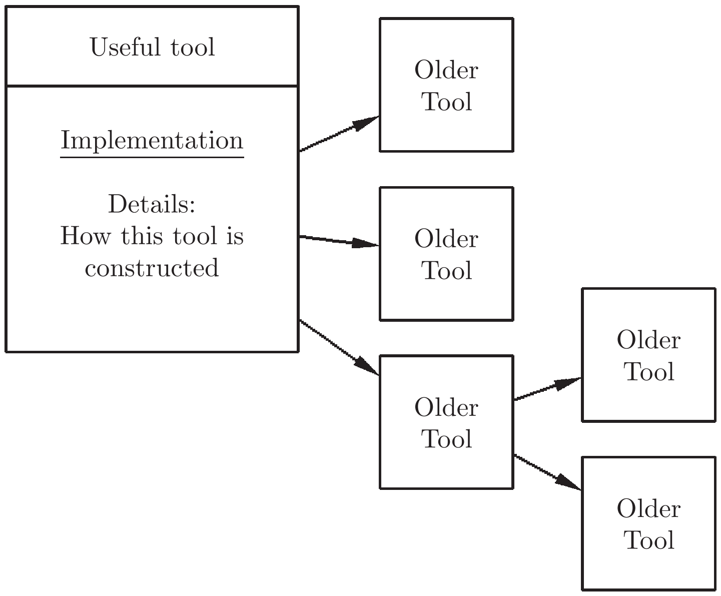


Figure 6.2: Abstraction: Build useful tools in order to build other useful tools

```
        }
    if (numbers.get(start+1) > numbers.get(start+2))
        {  tmp = numbers.get(start+1);       // swap second and third
           numbers.set(start+1, numbers.get(start+2));
           numbers.set(start+2, tmp);
        }
    if (numbers.get(start) > numbers.get(start+1))
        {  tmp = numbers.get(start);       // swap first and second
           numbers.set(start, numbers.get(start+1));
           numbers.set(start+1, tmp);
        }
}
```

In order to swap, or exchange, the values at two positions in the list, we use a local variable, `tmp`:

1. Store the first item in `tmp`.
2. Store the second item into the position of the first item.
3. Store `tmp` into the position of the second item.

Does the method shown above actually accomplish what we claim? We would recommend extensive testing before actually using this method – the logic is not simple (and many students have attempted this problem with much more complicated logic). Rather than addressing this issue, we are going to use method abstraction to simplify this method.

The first thing we notice is that we have some code which is not perfectly duplicated code, but pretty close to it. Look at the sections where we are swapping the values at two positions in the list. Method abstraction suggests that this operation be done in a method, which is then invoked when needed; we'll call this method `swap`. Here is version 2:

```
/** Arrange the 3 items in the given list, at positions start, start+1,
 *  start +2 in ascending order.
 *  @param start the first position of the subsequence to be arranged
 *  order; start is not negative and start is less
 *  than numbers.size()-2.
 *  @param numbers A list of numbers, size is at least 3.
 */
public void sort3(ArrayList <Integer> numbers, int start)
{  if (numbers.get(start) > numbers.get(start+1))
         swap (numbers, start, start+1);
   if (numbers.get(start+1) > numbers.get(start+2))
         swap (numbers, start+1, start+2);
   if (numbers.get(start) > numbers.get(start+1))
         swap (numbers, start, start+1);
}

/** Exchange the values in positions first and second in the given list
 */
private void swap(ArrayList <Integer> numbers, int first, int second)
{  int tmp;
   tmp = numbers.get(first);
   numbers.set(first, numbers.get(second));
   numbers.set(second, tmp);
}
```

We have abstracted the process of swap to its own method. Our sort3 method is now easier to read (it is shorter), and if we had an error in our swapping, it would occur only once, in the `swap` method. Incidentally we have made `swap` a *private* method because we see no immediate need for this method outside of the class in which it exists. Here we are hiding details that are not needed by someone who is using the sort3 method. The private `swap` method is sometimes called a *helper* method.

We see version 2 as a considerable improvement over version 1, but we can do better. In version 3, shown below, we will define another private helper method to sort a subsequence of length 2. Then we can use that in our `sort3` method:

```
/** Arrange the 3 items in the given list, at positions start, start+1,
 *  start +2 in ascending order.
 *  @param start the first position of the subsequence to be arranged
 *  order; start is not negative and start is less
 *  than numbers.size()-2.
 *  @param numbers A list of numbers, size is at least 3.
 */
public void sort3(ArrayList <Integer> numbers, int start)
{   sort2(numbers, start);
    sort2(numbers, start+1);
    sort2(numbers, start);
}

/** Arrange the 2 items in the given list, at positions start and
 *  start+1 in ascending order.
 *  @param start the first position of the subsequence to be arranged
 *  order; start is not negative and start is less
 *  than numbers.size()-1.
 *  @param numbers A list of numbers, size is at least 2.
 */
private void sort2 (ArrayList <Integer> numbers, int start)
{  if (numbers.get(start) > numbers.get(start+1))
      swap (numbers, start, start+1);
}

.... swap method same as in version 2.
```

Method abstraction has once again provided a nice solution to this problem. Version 3 has some distinct advantages over version 2:

- We have shortened the sort3 method considerably by introducing another helper method.
- If we needed to define a sort4 method, to sort subsequences of length 4, it would be a fairly easy extension to what we have done; we could use just three calls to sort3.

6.2.3 Object abstraction and encapsulation

We would now like to address the issue of abstraction with respect to objects. Objects consist of state (fields or instance variables) and behavior (methods). Abstraction tells us to hide unnecessary details, so in designing a class we will make the fields *private*. This will have several advantages:

- Any programmer who needs to use our class will look at the API. They will not see anything which is private, and moreover they should not *need* to see anything which is private. The details are hidden from view.

- Any method in some other class (call it a *foreign* method) which tries to access a field in our class will not compile. This affords the following advantages:
  - Foreign methods are not able to store erroneous or non-valid values into our fields. If the field `gpa` in our `Student` class were public, a method in some other class would be able to assign it a negative value, which is clearly not appropriate.
  - We now have the freedom to change our mind about field names and types. Changing things which are public would require users of our class to recompile and retest – a major inconvenience.
- The fields constitue the internal state of an object of our class; all access from foreign classes should be through public methods.

The practice of making fields private is often referred to as *encapsulation*. Our fields are 'protected' from the abuse of foreign classes, just as the capsule of a pill protects the contents from the external environment.

6.2.4 Exercises

1. Define the method `sort4` which will arrange a subsequence, of length 4, of a list in increasing order. Use calls to the `sort3` method.

```
/** Arrange the 4 items in numbers at positions start through
 *  start+3 in ascending order.
 *  @param numbers A list of whole numbers with length at least 4.
 *  @param start A position in the the list; not negative and
 *  less than size of the list - 3
 */
public void sort4 (ArrayList <Integer> numbers, int start)
```

2. We have defined a method which will print the best Student in each of four lists of Students (see `ch6/TopStudents.java` in the code repository).

   Use method abstraction to eliminate duplicated code from the method `topStudents`.
   Hint: Define a private helper method to return the best Student in a List of Students.

3. We wish to find the prime factors of a given int. We will do this by building some useful tools first.

   (a) Define a method named `isPrime()` which determines whether a given int is prime.

```
/** @return true if n is a prime number
 */
private boolean isPrime (int n)
```

(b) We can now produce a list of prime numbers fairly easily. Define a method named `primeList()` with an int parameter, which returns a list of all prime numbers less than or equal to the given parameter.

```
/** @return list of primes less than or equal to n
 */
private List <Integer> primeList (int n)
```

(c) Use the primeList method to find the prime factors of a given int. Define a method named `primeFactors` which returns all the prime factors of a given int in a list.

```
/**
 * @return a list of the prime factors of n.
 * @param n > 0
 */
public List <Integer> primeFactors (int n)
```

6.3 Inheritance

We now wish to extend our Student class example. We have graduate students and undergraduate students, and there are some differences between these two kinds of students:

- Undergraduate students are allowed to participate in intercollegiate sports. Graduate students are not permitted to do so (NCAA rules).
- All graduate students hold a bachelor's degree; undergraduate students generally do not.
- Graduate students are permitted to register for courses at the 600 level; undergraduate students are not permitted to do so.
- All graduate students are registered for a 3-credit 'Thesis' course, which is not included in their GPA.

We could replace our Student class with two new classes, Undergrad and GradStudent:

```
/** An Undergraduate student has a name, an ssn, and a gpa.
 *  Also, undergrads are permitted to play sports.
 */
public class Undergrad
{
private String name;
private String ssn;
private double gpa;
private boolean athlete;
```

```
/** Construct a new Undergrad with the given name
 *  and ssn.  gpa is initially 0.0
 */
public Undergrad (String newName, String newSSN)
{  name = newName;     // initialize fields from parameters
   ssn = newSSN;
   gpa = 0.0;          // initialize field to default value
   athlete = false;    // initialize to default value
}

// accessor methods
/** @return The name of this Undergrad
 */
public String getName()
{   return name;  }

/** @return The ssn of this Undergrad
 */
public String getSSN()
{   return ssn;  }

/** @return The gpa of this Undergrad
 */
public double getGPA()
{   return gpa;  }

/** @return true only if this Undergrad is an athlete
 */
public boolean getAthlete()
{   return athlete;  }

// mutator methods
/** Change the name of this Undergrad to the given name
 */
public void setName (String newName)
{   name = newName; }

/** Calculate the gpa of this Undergrad, if
 *  if the number of credits is positive
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits > 0)
      gpa = gradePoints / (double) credits;
}
```

```
/** Change the 'athlete' status of this Undergrad
 */
public void setAthlete(boolean ath)
{  athlete = ath;  }
}
```

We also have a Java class for graduate students:

```
/** A GradStudent student has a name, an ssn, and a gpa.
 *  Also, GradStudent has an undergrad degree
 */
public class GradStudent
{
private String name;
private String ssn;
private double gpa;
private String degree;

/** Construct a new GradStudent with the given name
 *  and ssn.  gpa is initially 0.0
 */
public GradStudent (String newName, String newSSN, String degr)
{  name = newName;     // initialize fields from parameters
   ssn = newSSN;
   gpa = 0.0;          // initialize field to default value
   degree = degr;
}

// accessor methods
/** @return The name of this GradStudent
 */
public String getName()
{   return name;  }

/** @return The ssn of this GradStudent
 */
public String getSSN()
{   return ssn;  }

/** @return The gpa of this GradStudent
 */
public double getGPA()
{   return gpa;  }

/** @return This the degree of this GradStudent
 */
public String getDegree()
{   return degree;  }

// mutator methods
/** Change the name of this GradStudent to the given name
 */
public void setName (String newName)
{   name = newName; }

/** Calculate the gpa of this GradStudent, if
 *  if the number of credits is more than 3.
 *  Assumes this GradStudent is registered for Thesis
 *  which is excluded from GPA.
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits > 3)
      setGPA (gradePoints / (double) (credits-3));   // exclude Thesis
}
}
```

This may seem like a lot of work for some minor changes to our program, but with copy and paste it does not take long. The problem, as noted earlier in this chapter, is that there is a lot of duplicated code here (it's so easy to do that with copy and paste). Much of these two classes are identical. Fortunately, object oriented languages such as Java give us a way to eliminate this duplicated code; it is called *inheritance*. We can define a new clss which inherits the (non-private) fields and methods of an existing class, in the same way that a person might inherit the traits of their parents.

Inheritance allows us to define a new class using an existing class. The existing class is sometimes called a *base class* or a *superclass*. The new class is called a *subclass*. The subclass automatically has access to all fields and methods from the superclass which are not private. This means that we will be able to eliminate the duplicated code by using Student as a superclass, and Undergrad and GradStudent as subclasses. A superclass may itself be a subclass of some other class; moreover, a class may have more than one subclass, but may have only one superclass. There is an existing class called *Object* which is, directly or indirectly, a superclass of all classes.

A simpler way of restating the above paragraph is that inheritance forms a tree-like *hierarchy* of classes with a class named `Object` at the *root* (i.e. at the top). This hierarchy is depicted in Figure 6.3. Take note of a few aspects of this class diagram:

- The arrows always point from subclass to superclass, and *never* from superclass to subclass.
- The arrowheads are hollow (unfilled) triangles.

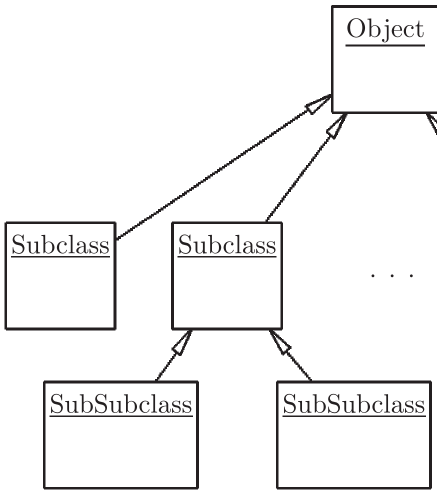


Figure 6.3: Class diagram showing Object at the root

- The format of a class diagram is specified in some detail by the Unified Modeling Language (UML) which we will not go into further at this time.

6.3.1 Is-a versus Has-a

To clarify the notion of inheritance we say that every instance of a subclass *is-an* instance of its superclass. In this case every Undergrad *is-a* Student, and every GradStudent *is-a* Student. But it is NOT the case that every Student *is-an* Undergrad; nor is it true that every Student *is-a* GradStudent. Inheritance is clearly a one-way street.

We should be careful to distinguish between inheritance and *composition*. Composition refers to the fields of a class. To describe the composition of the Undergrad class we would say that every Undergrad *has-a* name, every Undergrad *has-an* ssn, every Undergrad *has-a* gpa, and every Undergrad *has-an* athlete status.

Figure 6.4 shows some examples to help distinguish between *is-a* (inheritance) and *has-a* (composition).

When designing classes for your program, if it makes sense that every X *has-a* Y, then you should make Y a field in the X class. If it makes sense that every X *is-a* Y, then X should be a subclass of Y. As you'll see if and when you study object-oriented design in more depth, there will be cases where it is not clear whether inheritance or composition is appropriate; in such cases it is usually better to use composition.

Inheritance	Composition
Every UnderGrad *is-a* Student	Every Student *has-a* name
Every GradStudent *is-a* Student	Every GradStudent *has-a* degree
Every Car *is-a* Vehicle	Every Car *has-an* Engine
Every Whale *is-a* Mammal	Every Whale *has-a* Habitat
Every HashSet *is-a* Set	Every HashSet *has-a* size
Every ArrayList *is-a* List	Every ArrayList *has-an* array of items

Figure 6.4: Distinguishing between Inheritance (*is-a*) and Composition (*has-a*)

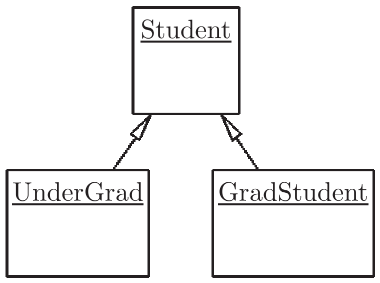


Figure 6.5: Class diagram showing relationship of Student classes

6.3.2 Factoring duplicated code and defining subclasses

To define a subclass in Java we use the keyword *extends*. This conveys the intent that the subclass consists of everything in the superclass, in addition to other fields and/or methods. The general format, when defining a subclass is:

```
 public class subclass-name extends superclass-name
{ ...  fields, constructors, methods ...  }
```

Note that there can be only one superclass name after the keyword `extends` because a class can have only one superclass.

We can now redefine our Student classes using inheritance. `Student` will be the superclass; `GradStudent` and `Undergrad` will be the subclasses. Figure 6.5 shows the class diagram which will result.

In order to decide which fields and methods are to be placed in which classes, we return to the notion of *factoring* duplicated code. All fields and methods which are identical in `GradStudent` and `Undergrad` will be factored into the `Student` class, whereas all fields and methods which are different will be retained in `GradStudent` and `Undergrad`.

Figure 6.6 shows which fields of `GradStudent` and `Undergrad` are identical and therefore can be factored to the `Student` class. Fields or methods which occur in one class but not the other are indicated by a $\surd$. For fields and methods which occur in both classes, the figure shows whether they are the same or different.

Using the information in Figure 6.6 we can now define our three classes. All fields which are the same in the `GradStudent` and `Undergrad` classes will be fac-

Fields	UnderGrad	GradStudent
name	same	same
ssn	same	same
gpa	same	same
athlete	√	
degree		√

Methods	UnderGrad	GradStudent
getName()	same	same
getSSN()	same	same
getGPA()	same	same
getAthlete()	√	
setName()	same	same
calcGPA()	different	different
setAthlete()	√	
getDegree()		√

Figure 6.6: Fields and methods which are the same in `UnderGrad` and `GradStudent` can be factored to the `Student` super-class

tored to the `Student` class: `name, ssn, gpa`. All methods which are the same in both classes will be factored to the `Student` class: `getName(), getSSN(), getGPA(), setName()`. Note that the method `calcGPA(int,int)` occurs in both the `GradStudent` and `Undergrad` classes; however, it cannot be factored to the `Student` class because the method body (i.e. the implementation) is different in `GradStudent` and `Undergrad`.

```
/** A Student has a name, an ssn, and a gpa.
 *  This class serves as a superclass for various
 *  kinds of students.
 */
public class Student
{
private String name;
private String ssn;
private double gpa;

// accessor methods
/** @return The name of this Student
 */
public String getName()
{   return name;  }

/** @return The ssn of this Student
 */
public String getSSN()
{   return ssn;  }

/** @return The gpa of this Student
 */
public double getGPA()
{   return gpa;  }

// mutator methods
/** Change the name of this Student to the given name
 */
public void setName (String newName)
{   name = newName; }

/** Change the gpa of this Student to the given gpa
 */
public void setGPA (double newGPA)
{   if (newGPA >=0)          // check for valid value
       gpa = newGpa;
}
}
```

We now handle the two subclasses, GradStudent and Undergrad. In the Undergrad class we will exclude those fields and methods which have been factored to the Student class:

```
/** Every Undergrad is a Student
 *  Undergrad is a subclass of Student.
 *  Every Undergrad has an Athlete status (boolean).
 *  Undergrads are not registered for Thesis.
 */
public class Undergrad extends Student
{
private boolean athlete;

/** @return true only if this Undergrad is an athlete
 */
public boolean getAthlete()
{   return athlete;  }

/** Calculate the gpa of this Undergrad, if
 *  if the number of credits is positive
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits > 0)
      setGPA (gradePoints / (double) credits);
}

/** Change the 'athlete' status of this Undergrad
 */
public void setAthlete(boolean ath)
{  athlete = ath;  }
}
```

In the `GradStudent` class we will exclude those fields and methods which have been factored to the `Student` class:

```
/** Every GradStudent is a Student.
 *  GradStudent is a subclass of Student.
 *  A GradStudent has a degree.
 *  All GradStudents are registered for 3-credit Thesis,
 *  which is not part of the GPA.
 */
public class GradStudent extends Student
{
private String degree;

/** @return This the degree of this GradStudent
 */
public String getDegree()
{   return degree;  }

/** Calculate the gpa of this GradStudent, if
 *  if the number of credits is more than 3.
 *  Assumes this GradStudent is registered for Thesis
 *  which is excluded from GPA.
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits > 3)
      setGPA (gradePoints / (double) (credits-3));
}
}
```

6.3.2.1 Constructors

The reader may have noticed that constructors are absent from the code presented thus far. Constructors require careful attention when using inheritance. We will include constructors in all three of our classes, and each constructor will be responsible for initializing the fields of its own class. In the Student class the constructor is the same as shown previously:

```
/** Initialize the fields of this Student
 *  @parm ssn Must be a valid ssn.
 *  gpa is initially 0.0
 */
public Student (String name, String ssn)
{   this.name = name;
    this.ssn = ssn;
    gpa = 0.0;
}
```

In the Undergrad class, keep in mind that every Undergrad has a name and an ssn. Therefore when an Undergrad is created, the creator will have to provide a name and an ssn. The constructor will then call the constructor in the superclass to initialize the appropriate fields. This is done with a call to *super*. A call to `super` in a constructor is a call to the constructor in the superclass, and the actual parameters in the call should correspond to the formal parameters in the superclass' constructor. This call to `super` must be the first statement in the subclass' constructor. After calling `super`, the Undergrad constructor will then initialize the athlete status to a default value, `false`. The constructor for Undergrad is:

```
/** Initialize the fields of this Undergrad.
 *  @param ssn Must be a valid ssn.
 *  Athlete status is initially false.
 */
public Undergrad (String name, String ssn)
{  super (name, ssn);       // call constructor in Student class
   athlete = false;
}
```

We use a similar strategy for the constructor in the GradStudent class. In this case the constructor will need another parameter for the GradStudent's degree:

```
/** Initialize the fields of this GradStudent.
 *  @parm ssn Must be a valid ssn.
 *  @param degree should include degree title and institution
 */
public GradStudent (String name, String ssn, String degree)
{  super (name, ssn);       // call constructor in Student class
   this.degree = degree;
}
```

6.3.3 Making use of inheritance

We close this section with a brief example showing how these classes can be used. Some other class, call it the *client*, could have a method containing the following code segment:

```
UnderGrad  younger = new GradStudent("jim", "322-23-3234");
GradStudent older = new GradStudent("sue", "240-44-2222");
younger.setAthlete(true);
older.setDegree ("B.A. from Penn State");
System.out.println ("Our new students are " +
                     younger.getName() + " and " +
                     older.getName());
```

Conceivably, we could also create a Student who is neither an UnderGrad nor a GradStudent, but just a plain Student:

`Student stud = new Student("joe", "223-98-1782");`

This student would be neither a GradStudent nor an UnderGrad, and this begs the question: does it make sense to have this kind of student in our program? We'll come back to this question later.

6.3.3.1 Assignment of references to variables

Once we have declared variables which store references to various kinds of students, we can instantiate those classes and assign the reference to the appropriate variable. A reference to an Undergrad can be assigned to a variable declared as `Undergrad`, and a reference to a GradStudent can be assigned to a variable declared as `GradStudent`, and a reference to a Student can be assigned to a variable declared as `Student`.

Moreover, since every `Undergrad` *is-a* `Student` and every `GradStudent` *is-a* `Student`, we can do the following:

```
Student stud1, stud2;
stud1 = new Undergrad("jim", "322-23-3234");
stud2 = new GradStudent("sue", "240-44-2222");
```

It would be a mistake to go the other way:

```
Undergrad younger = new Student ("jim", "322-23-3234");   // ERROR
```

The compiler will not allow this because it is *not* true that every `Student` is-an `Undergrad`.

The variables `stud1`, declared to be of type `Student` is now storing a reference to an `Undergrad`, and `stud2`, also declared to be of type `Student` is storing a reference to an `GradStudent`. Also note that these can change as the program executes:

`stud1 = new GradStudent("joe", "323-87-0102");`

So `stud1` is now storing a reference to a GradStudent.

	Variable			
	st		ug	
Code	static type	dynamic type	static type	dynamic type
Student st;	Student			
Undergrad ug;	Student		Undergrad	
ug = new Undergrad(”jim”,”22”)	Student		Undergrad	Undergrad
st = ug;	Student	UnderGrad	Undergrad	Undergrad
st = new GradStudent (”joe”,”33”);	Student	GradStudent	Undergrad	Undergrad

Figure 6.7: Static type versus dynamic type. Static type changes as the code executes.

The question to be addressed now is, What is the type of stud1 – UnderGrad or Student? We need to distinguish *two kinds of type*: *static type* and *dynamic type*.

Static type:

- Static type is the type of the variable shown in the declaration.
- Static type is determined when the program is compiled.
- Static type does not change as the program executes. It is in effect for the lifetime of the variable.

Dynamic type:

- The dynamic type of a variable is the type of the reference assigned to the variable.
- The dynamic type of a variable is determined when the program executes.
- The dynamic type of a variable can change as the program executes.

Figure 6.7 shows a code segment in which the the static and dynamic types of variables are shown as the code is executed. Static type is important because it will determine whether your program compiles without errors. Dynamic type is important, as we shall see in the next section, because it will determine which method is being called.

6.3.3.2 Assignment to subclass from superclass – casting

We now consider the case where we may wish to assign to a variable whose static type is a subclass, but we are assigning from a variable whose static type is the superclass; this can work, but only if the dynamic type of the variable being assigned is correct. Here is an example:

```
Student stud1 = new Undergrad("jim", "322-23-3234");
Undergrad younger;
```

Code	Comments
Student st;	
Undergrad ug;	
st = new Undergrad(”joe”, ”23”);	
ug = st;	ERROR (at compile time)
ug = (Undergrad) st;	cast down to Undergrad
st = new GradStudent(”jim”,”32”);	
ug = (Undergrad) st;	ERROR (at run time)

Figure 6.8: Assignment to super-class variable, with a cast

Here we should be able to store the reference, `stud1` , into the variable `younger` because the dynamic type, `Undergrad` matches the static type of `younger`, but the compiler will not accept it:

`younger = stud1; // ERROR`

The problem is the compiler is not convinced (and not smart enough to know) that the dynamic type of stud1 is `Undergrad` (recall that dynamic type is determined at execution time). We need to convince the compiler that everything will be okay at execution time; this is done with a *cast*. A cast forces the type of a reference to a particular type. We have already seen casts with respect to primitive types; they can also be used with reference types. To apply a cast to an expression, simply pubt the casting type in parentheses and preceding the expression. The format is:

(type) expression

For example, `(Undergrad) stud1;` produces a reference to an Undergrad, which can now be assigned to a variable whose static type is Undergrad:

`younger = (Undergrad) stud1; // OK`

However, if the the dynamic type of stud1 is not truly `Undergrad` as we are claiming, we will get a runtime error when the cast is applied. Figure 6.8 shows another code segment, similar to Figure 6.7, in which we show some valid and non-valid operations.

When considering the use of a cast, remember that you are always casting *down* the class diagram, from a superclass to a subclass. `(Undergrad) student` is fine, but `(Student) undergrad` is not permitted.

6.3.3.3 Importance of inheritance

Inheritance is *extremely* important in Java programming. Looking at the Java class library, you will see inheritance *everywhere*. Moreover, we can extend classes from the Java class library to form extensions. For example, we might want an improved version of the ArrayList class which can tell us whether the items are in ascending order. This can be done easily by extending ArrayList with a subclass that has the desired method:

```
public class BetterArrayList extends ArrayList
{
```

```
public boolean isAscending()
{  ///  method body here  }
}
```

6.3.4 Exercises

1. Fill in the missing entries in the table below with one of the following:
    - Is-a
    - Has-a
    - Neither
    - Both

Car	Has-a	Engine
SUV		Car
Student		Person
Person		SSN
Petunia		Plant
Petunia		Flower
Petunia		Stem
Student		University
University		Collection of Students
Windows Folder		Collection of Documents and Folders
Java method		Signature

2. Assume `Person` and `Animal` have been defined as classes. Which of the following contain syntax errors?

    (a)
    ```
    public class Student subclass Person
    {  }
    ```

    (b)
    ```
    public class Student extends Person
    {   }
    ```

    (c)
    ```
    public class Student extends Person, Animal
    {   }
    ```

3. Refer to the classes `Student` and `GradStudent` described in this section. Define classes named `PhdStudent` and `MastersStudent`. Both should be sublcasses of `GradStudent`. A PhdStudent should have two fields:
    - A boolean field, true only if the PhdStudent has passed qualifying exams.
    - A String field, storing the PhdStudent's dissertation topic, or null if the PhdStudent has no dissertaion topic.

Newly created PhdStudents have not passed qualifying exams and have no dissertation topic.

A MastersStudent has one field, a boolean which is true if the MastersStudent is on a thesis track. When a MastersStudent object is created, it should be possible to specify whether the MastersStudent is on a thesis track.

Include appropriate public accessor and mutator methods in these classes.

4. Assume we have defined a class named `Vehicle` which has two subclasses named `Bicycle` and `Car`. These classes all have default constructors. Which of the following contain syntax errors, and which contain run-time errors?

   (a)
   ```
   Vehicle v;
   Bicycle b;
   v = new Bicycle();
   ```

   (b)
   ```
   Vehicle v;
   Bicycle b;
   b = new Vehicle();
   ```

   (c)
   ```
   Vehicle v;
   Bicycle b;
   v = new Bicycle();
   b = v;
   ```

   (d)
   ```
   Vehicle v;
   Bicycle b;
   v = new Bicycle();
   b = (Bicycle) v;
   ```

   (e)
   ```
   Vehicle v;
   Bicycle b;
   Car c;
   v = new Bicycle();
   c = (Car) v;
   ```

5. Look at the API for the `ArrayList` class in the `java.util` package.

   (a) What is the superclass of `ArrayList`?

   (b) What are the (direct) subclasses of `ArrayList`?

6.4 Polymorphism and dynamic method look-up

In this section we will examine how dynamic type is used in method calls. We will also see that objects can exhibit different behavior, depending on their dynamic types. The word *polymorphism*, in the natural sciences, means 'taking on different forms or appearances.' In object-oriented programming polymorphism is exhibited when two identical method calls can result in the invocation of different methods. This can be particularly useful when we have a collection of objects, with different dynamic types, and the action we wish to perform on each of those objects will depend on its dynamic type.

6.4.1 Dynamic method look-up

Before going into the details of polymorphism, we need to take a closer look at the mechanism which is used when methods are called. Continuing with our Student classes, as shown in Figure 6.3, assume we wish to print the name of a particular student:

```
Student st;
st = new UnderGrad("jim", "22");
System.out.println ("The student's name is " + st.getName());
```

The method call `st.getName()` means to apply the `getName()` method to the `st` object. However, `st` refers to an object of type `Undergrad` and that class has no `getName()` method. We need to understand that methods are invoked by a process known as *dynamic method look-up*. When a method is called via `object.method()`:

- Look in the class of the given object's dynamic type. If the given method is there, that is the method which is invoked.
- If the method is not there, look in the superclass of the given object. If the method is found there, that is the method which is invoked.
- If the method is not there, continue to look in the super-super-superclass.
- Ultimately the method will be found, or the Object class will be reached (Object is always at the root of the class hierarchy). If the method is not found anywhere on the path to Object, an error is produced (as we will see later, this could be a compile-time error or a run-time error).

In the code segment shown above the call `st.getName()` will find no such method in the Undergrad class, so it will look in the superclass, `Student` and find the method to be invoked in that class.

6.4.2 Polymorphism

Having defined dynamic method look-up, we are now in a position to understand polymorphism in object-oriented programming. Consider the case where we wish to invoke a method such as calcGPA on a variable whose static type is Student:

```
Student st = new Undergrad("jim", "22");
int cr = readCredits(st);
int gp = readGradePoints(st);
st.calcGPA(gp,cr);
```

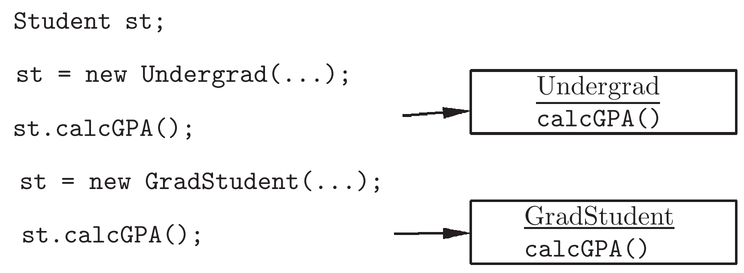
In this case the compiler will issue an error on the call to calcGPA because there is no such method in the Student class. Unfortunately we know how to calculate a GPA only for GradStudents and Undergrads, but not for ordinary Students. To remedy this we can include a method in the Student class as a place-holder, simply to satisfy the compiler, as long as we are sure that it never actually gets called (in the next section we'll see a better way to handle this). In the `Student` class:

```
/** This method should never be invoked.  It is here only as
 *  a place-holder, so that calls to calcGPA() can be compiled
 */
public void calcGPA(int gradePoints, int credits)
{ // do nothing
}
```

Now the method call `st.calcGPA(int,int)` will work fine; it will call the `calcGPA(int,int)` method in the `Undergrad` class because the dynamic type of `st` is `Undergrad`. To further explain polymorphism, we could extend that code segment with two more statements:

```
Student st = new Undergrad("jim", "22");
int cr = readCredits(st);
int gp = readGradePoints(st);
st.calcGPA(gp,cr);
st = new GradStudent ("mary", "32");
cr = readCredits(st);
gp = readGradePoints(st);
st.calcGPA(gp,cr);
```

The first call `st.calcGPA(gp,cr)` will invoke the `calcGPA(int,int)` method in the `Undergrad` class, and the second call `st.calcGPA(cr,gp)` will invoke the `calcGPA(int,int)` method in the `GradStudent` class. Two identical statements result in different methods being invoked. This is polymorphism and is depicted in Fig 6.9



```
Student st;

st = new Undergrad(...);

st.calcGPA();

st = new GradStudent(...);

st.calcGPA();
```

Figure 6.9: Polymorphism: Identical method calls invoke methods in different classes

6.4.2.1 Polymorphism with collections

Polymorphism is most often used in connection with collections. For example, assume that `roster` stores a reference to a list of students:

```
List <Student> roster;
roster = new ArrayList <Student>();
roster.add (new UnderGrad ("jim", "22");
roster.add (new GradStudent ("mary", "32");
roster.add (new UnderGrad ("joe", "56");
```

We now wish to print the name and GPA of each Student in the list. Before printing a student's gpa, we will make sure it has been calculated. Polymorphism handles this perfectly:

```
for (Student st : roster)
  {  int cr = readCredits(st);
     int gp = readGradePoints(st);
     st.calcGPA(gp,cr);
     System.out.println ("The GPA for " + st.getName() + " is " +
                          st.getGPA();
  }
```

The method call to `calcGPA(gp,cr)` will invoke the appropriate method using dynamic method look-up.

6.4.3 Exercises

1. Assume that we have a class named `Vehicle` with two subclasses: `Car` and `Bicycle`. These classes all have default constructors. The subclasses each has a method named `getMPG()`.

   In the Car class:

   ```
   public double getMPG()
   {  return 35.0;  }
   ```

In the Bicycle class:

```
public double getMPG()
{  return 0.0;  }
```

(a) Which line(s) shown below will cause syntax error(s)?

```
Bicycle b = new Bicycle();
Car c = new Car();
Vehicle v = new Bicycle();
System.out.println (b.getMPG());
System.out.println (c.getMPG());
System.out.println (v.getMPG());
```

(b) Show change(s) to any of these classes which will prevent the syntax error(s) from the previous problem.

(c) In the previous problem, the method call `v.getMPG()` will result in a call to the `getMPG()` method in which class (after the correction has been made)?

(d) Define a method named `getMPG()` with one parameter, a List of Vehicles, which will return the average MPG those Vehicles:

```
/** @return the average MPG of the given vehicles, or 0 if the List
 *  is empty
 */
public double averageMPG (List <Vehicle> vehicles)
```

2. This exercise refers to dynamic method lookup. Figure 6.10 depicts a class diagram showing public void methods that are defined in each class. Notice that the same method signature can occur in several different classes. Assume the following declarations:

```
Class2 c2;
Class3 c3;
Class4 c4;
Class5 c5;
```

Assume the variables declared above have been assigned non-null values. In each of the following show which method is invoked, by giving the name of its class, or indicate that an error will occur.

(a) `c4.method1();`
(b) `c4.method2();`
(c) `c5.method1();`
(d) `c2.method2();`
(e) `c2.method1();`

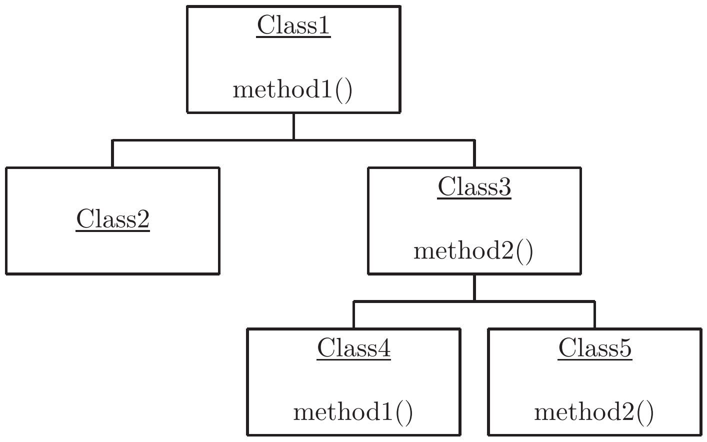


Figure 6.10: Class diagram for exercise on dynamic method lookup. Each method is public void.

6.5 Overriding methods from the Object class

Recall that the `Object` class is (directly or indirectly) a superclass of every Java class. The `Object` class is in the package `java.lang` and does not need to be imported. Looking at the API for the `Object` class we see at least three interesting methods:

- `public String toString()` - An object of this class can be represented by a String.
- `public boolean equals (Object)` - An object of this class can be compared for equality with any other object (more on this in chapter 7).
- `public int hashCode()` - An object of this class can produce an int likely to be unique for unequal objects of the same class (more on this in chapter 8).

6.5.1 Overriding the `toString()` method

The purpose of the `toString()` method is to produce as a result a String representation of an object. This will be useful when one needs to display data for a user; the `toString()` method should format the data to be readable and clear to the user. It should return that formatted result as a single String (it may contain newline characters). As an example, we could override the `toString()` method in our Student class as shown below:

```
/** @return this Student as a String */
public String toString()
{   String result = "Name: " + name + "\n";
```

```
    result += "SSN:  " + ssn + "\n"; // concatenate ssn
    result += "GPA:  " + gpa + "\n"; // concatenate gpa
    return result;
}
```

The following code

```
Student s1 = new Student ("Joe", "123-45-6789");
System.out.println (s1.toString());
```

would produce the following output:

```
Name: joe
SSN:  123-45-6789
GPA:  0.0
```

The `toString()` method becomes even more useful when we learn that it is called automatically by the Java runtime environment when:

- An object which is not a String is concatenated with a String. The toString() method produces a String for the concatenation.
- An object is passed as a parameter to the System.out.println method since println is expecting its parameter to be a String.

This means that we can concatenate a String with a Student object:

```
"Best student is " + s1
```

and we can simplify the call to println:

```
System.out.println (s1);
```

without explicitly calling toString().

Most of the classes in the Java class library override the `toString()` method. This means that you can print objects of those classes easily, and expect to get a pretty good looking result. Even Collection classes such as ArrayList have a toString() method. In the case of an ArrayList the toString() method will produce a result consisting of:

1. An open bracket - [
2. String representations of all the elements in the ArrayList (these are produced by calling toString() on each element), separated by commas
3. A close bracket -]

An ArrayList of 3 Strings might appear like this:

```
["joe","jim","mary"]
```

Incidentally, primitives can also be converted to Strings with concatenation. If the variable `sum` is an int, with value 23, the value of `"Sum is " + sum` is the String `"Sum is 23"`. However, Figure 6.11 shows instances where concatenation might produce unexpected results. To understand Figure 6.11 recall that when there are several + operators in an expression, they are executed left to right. To create a String representation of a primitive, simply concatenate it with a String of length 0:

`17 + ""` produces the String `"17"`.

Plus operation(s)	Result	Explanation
`2 + 3`	`5`	Addition of ints
`"2" + "3"`	`"23"`	Concatenation of Strings
`2 + "3"`	`"23"`	Concatenation of Strings
`"2" + 3`	`"23"`	Concatenation of Strings
`2 + 3 + " is the result"`	`"5 is the result"`	Addition done first
`"Result is " + 2 + 3`	`"Result is 23"`	Concatenation done first
`"Result is " + (2 + 3)`	`"Result is 5"`	Parentheses take precedence

Figure 6.11: The overloaded + operator can mean addition of numbers or concatenation of Strings, depending on the context.

6.5.1.1 Failing to override toString()

What would have happened in the prior examples if we had not included a `toString()` method in our Student class? The compiler will allow a call to `s1.toString()` because the `toString()` method is defined in a superclass (Object). Then at runtime when `toString()` is called, dynamic method lookup tells us that it will search for this method in superclass(es), until it is found. In this case it will be found in the `Object` class. A quick look at the API for `Object` shows that `toString()` will call `hashCode()` (see below) which returns an int. This int is formatted in hexadecimal (base 16), concatenated with the name of the class, and returned as a String. This is most likely not what you wish to happen. In summary, if you are printing an object, and you see some strange looking output, such as `Student@49a3c30`, you need to define a `toString()` method in the Student class.

6.5.2 Exercises

1. Consider the following class

   ```
   public class Vehicle
   {   int wheels;

       public Vehicle (int wheels)
       {  this.wheels = wheels;  }
   }
   ```

   In some other class define a method in which you have the following:

   ```
   Vehicle v1 = new Vehicle(18);     // semi
   System.out.println (v1);
   ```

   If the output makes no sense, fix the Vehicle class so that the output will be more readable, such as:

   ```
   Vehicle with 18 wheels
   ```

2. Test your solution to the previous problem by creating a List of at least 3 Vehicles, all with different number of wheels. Print the list without using a loop.

3. Show the output in each case:

   (a) `System.out.println (" 5 + 4 + is " + 5 + 4) ;`

   (b) `System.out.println ( 5 + 4 + " is " + 5 + 4) ;`

   (c) `System.out.println (" 5 * 4 + is " + 5 * 4) ;`

   (d) `System.out.println (" 5 + 4 + is " + (5 + 4)) ;`

6.6 Abstract methods and classes

6.6.1 Abstract methods

We now return to the definition of the `calcGPA(int,int)` method in the Student class. Recall that it was included simply as a 'place-holder', to satisfy the compiler; we expect that it will never be invoked:

```
public void calcGPA(int gradePoints, int credits)
{          // do nothing
}
```

If it seems strange to you that there should be a method which does nothing at all, you are not alone. This occurs so often that Java has a designation for this kind of method called *abstract*. A method which exists in a superclass merely to support the existence of methods having the same name in subclasses should be declared as `abstract`:

```
public abstract void calcGPA(int gradePoints, int credits);
```

Note that instead of a method body, there is a single semicolon; abstract methods have no body, and need no body, because they are never invoked. With this slight change our Student classes should work just as well.

Remember the following concerning abstract methods:

- Used as a place-holder for methods of the same name in subclasses
- Declared with the `abstract` keyword in the signature
- Semicolon after the parameter list
- No method body, not even the curly braces
- Must be implemented in subclasses
- May be used only in abstract classes (see next section)

6.6.2 Abstract classes

Any class which has at least one abstract method must be declared as an *abstract class*. This is done with the keyword `abstract` in the declaration of the class:

```
public abstract class Student
```

Earlier we said that we were not sure that it would make sense to instantiate a Student, not knowing what kind of Student he/she is. If this truly is not desirable, an abstract class is exactly what we need. When a class is abstract it cannot be instantiated:

```
new Student("jim", "33"); // ERROR
```

Thus by making the Student class abstract, we ensure that no client will ever be able to instantiate a Student, but will be able to instantiate GradStudent and Undergrad because they are *concrete* classes (i.e. not abstract).

When using abstract classes which may have one or more abstract methods, we must be sure that the methods are implementd in subclasses. If an abstract method were not implemented in one of the subclasses, a method call to an object of that subclass would have no method to invoke. For example, if the `Undergrad` class had no `calcGPA(int,int)` method, then dynamic method look-up would fail for a call to `st.calcGPA(gp,cr)` in the case that the dynamic type of `st` is `Undergrad`.

Remember the following about abstract classes:

- Declared with `abstract` keyword at the top
- May have one or more abstract methods
- Cannot be instantiated
- Abstract methods must be concrete (not abstract) in subclasses

A subclass can also be abstract, in which case it would not be required to implement abstract methods inherited from a superclass, but *its* (concrete) subclasses would be required to implement the abstract methods. In other words, viewing the class diagram from top to bottom, all abstract methods must be implemented somewhere on a path from the top to a concrete class, as depicted in Figure 6.12.

Notice in Figure 6.12 that the method `meth1` is implemented (i.e. concrete) in all concrete subclasses, but method `meth2` need not be implemented in class `Sub3` nor in class `Sub4` because it is implemented in class `Sub2` and is therefore available in classes `Sub3` and `Sub4`.

6.6.3 Exercises

1. Point out the syntax error, if any, in each of the following:

   (a)
   ```
   public class Class1
   {  public abstract void method1();  }
   ```

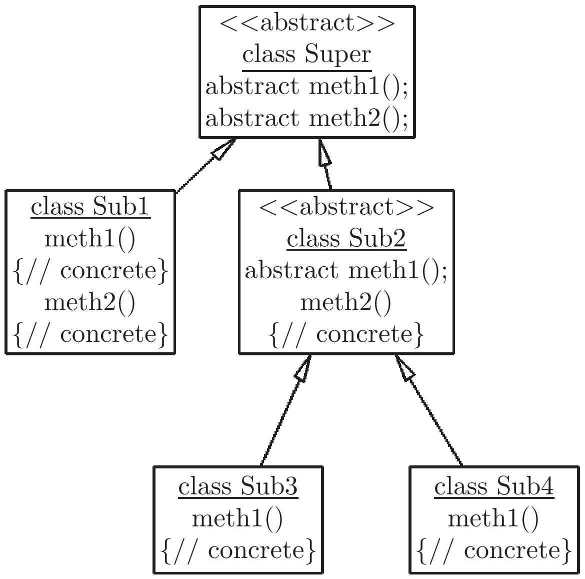


Figure 6.12: Class diagram showing an abstract method which must be implemented (concrete) for use in concrete sub-classes

(b)
```
public abstract class Class1
{  public void method1()
   {     }  // do nothing
}
```

(c)
```
public abstract class Class1
{  public abstract void method1();  }

public class Class2 extends Class1
{  int field1;
   public void method2()
   {   }    // do nothing
}
```

(d)
```
public abstract class Class1
{  public void method1()
   { Class1 c1 = new Class1();    }
}
```

2. Use the Student, GradStudent, and UnderGrad classes from the project university-ch6 in the code repository. We wish to calculate the GPA for a list of Students:

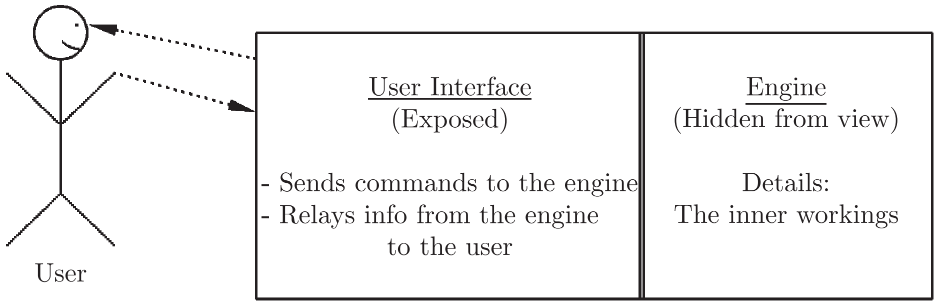


Figure 6.13: User Interface: All communication with the engine is through the user interface

```
List <Student> roster = new ArrayList <Student>();
//  several students added to roster
for (Student st : roster)
    st.calcGPA(17,3);
```

Make the necessary changes to the Student class to get this code to compile and execute. Assume that Student will never be instantiated.

6.7 Java Interfaces

There are at least three different, but related, meanings of the word *interface* in computer science:

- A user interface is that which stands between an entity and the user of that entity. All communication with the entity generally goes through the interface (in both directions) as shown in Figure 6.13. We'll call the entity being used the *engine*. User interfaces generally simplify usage of the engine for the user and can provide the user with useful information about the state of the engine while hiding unnecessary details of the internal workings of the engine.

  Examples of user interfaces include:

  – The API for a class constitutes an interface showing a potential user how the class can be used.

  – The API and signature for a method constitutes an interface showing a potential user how the method can be used.

  – The command language for an operating system such as DOS or Unix constitutes an interface between the user and the services available in the operating system.

  – An automobile's dashboard, gear stick, foot pedals, etc. constitute an interface with the engine of an automobile.

Many feel that user interfaces should be standardized, rather than proprietary (owned by a single company). What would happen if an automobile manufacturer designed a car in which the brake pedal was on the right, and the accelerator pedal was on the left?

- A *graphical user interface* (GUI) generally refers to software which allows the user to communicate with a program while it is executing using a graphic images (icons, trash basket, folders, etc) and some sort of pointing device such as a mouse. The first such GUI was the desktop GUI provided on the early Apple Macintosh computers (actually derived from a system developed at Xerox PARC). The desktop metaphor provided users with a means of coommunication with the operating system which was fairly intuitive and easy for the user. Today most software applications provide a GUI for the user.

  Related to the question of standardization of user interfaces, when Microsoft followed Apple's strategy by introducing a GUI for the PC – Windows, Apple filed a copyright infringement lawsuit, claiming that Microsoft had stolen the *look and feel* of their GUI (ironically, Apple had taken the idea from Xerox years earlier).

- A *Java interface* is similar to an abstract class which has no fields and no concrete methods. In this section we will provide some motivation for, and examples of, Java interfaces.

6.7.1 The need for Java interfaces – multiple inheritance

In this section we provide some motivation for the need for Java interfaces, but first we should discuss *multiple inheritance*.

We have said that a Java class may not have more than one superclass, but some progamming languages will actually permit this so-called multiple inheritance. To motivate this discussion, we return to our example involving the classes Student, Undergrad, and GradStudent. We now add two more classes to this project: `Prof` and `Instructor`. A Prof is a member of the faculty whose job it is to teach classes and conduct research. An Instructor is anyone who teaches classes. Often at a research university, grad students are asked to teach classes. This would mean that a GradStudent *is-a* Student, and a GradStudent *is-an* Instructor. As we saw previously, the is-a relationship implies the need for inheritance, as shown in Figure 6.14. We now have a problem because Java will not allow multiple inheritance:

```
public class GradStudent extends Student, Instructor // ERROR
```

A discussion of the pros and cons of allowing multiple inheritance in a programming language is beyond the scope of this book; suffice it to say that multiple inheritance can complicate things for both the programmer and the compiler writer.

Java provides a good solution to this problem: the *Java interface*. A Java interface is similar to an abstract class which has no fields and in which all the

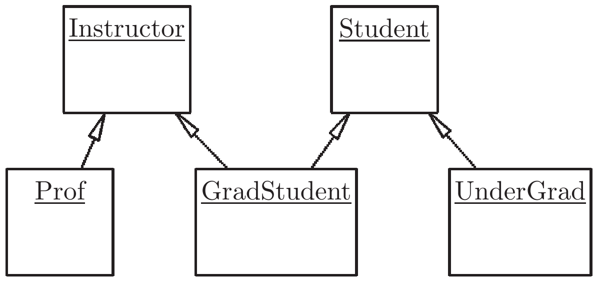


Figure 6.14: Class diagram showing multiple inheritance for the GradStudent class; this is not permitted in Java

methods are abstract. An interface is declared with the keyword `interface` instead of `class`. It is basically a template for subclasses, showing all the methods which must be implemented:

```
public interface Instructor
{   public abstract List <Course> getCourses();
    public abstract String getName();
}
```

This is an interface with 2 methods, both abstract. Note the following concerning Java interfaces:

- An interface must not contain any instance variables (i.e. non-static fields).
- An interface must not contain a constructor.
- All methods in an interface must be public abstract. If not declared as such, the compiler will assume they are public abstract.
- An interface, like an abstract class, must not be instantiated.

We can redefine our interface more briefly as:

```
public interface Instructor
{   List <Course> getCourses();
    String getName();
}
```

Now we can deal with the problem of multiple inheritance. To specify a subclass relationship with an interface, we say that a class *implements* the interface:

```
public class GradStudent extends Student implements Instructor
```

Our class diagram can now be drawn as shown in Figure 6.15 It shows that every GradStudent *is-a* Student and every GradStudent *is-an* Instructor.

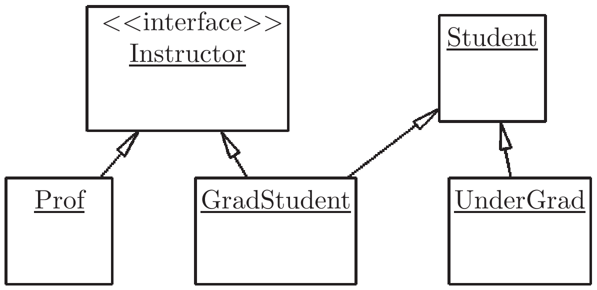


Figure 6.15: Class diagram showing multiple inheritance with an interface: Instructor

The compiler is satisfied because GradStudent extends only one class: Student. Instructor is an interface, not a class.

Since the class GradStudent implements the Instructor interface, GradStudent will have to implement all the methods in that interface: `getCourses()` and `getName()`.

A class may implement more than one interface; they are listed in the declaration separated by commas. For example if there was another interface called `Researcher`, we could define the GradStudent class as shown below:

```
public class GradStudent extends Student implements Instructor,
Researcher
```

6.7.2 Interfaces which we've already been using

Interfaces are very common in the Java class library; we've already started using them: List and Set.

If you look at the API in the Java.util package, you'll notice that:

- Interfaces and classes are separated.
- List is an interface; ArrayList implements List. Every ArrayList *is-a* List. There are other classes which implement List, which we have not yet used.
- Set is an interface; HashSet implements Set. Every HashSet *is-a* Set. There are other classes which implement Set, which we have not yet used.

When we declare a list:
`List <Student> roster;`
we are saying that the variable `roster` may store a reference to any kind of list. When we instantiate the list:
`roster = new ArrayList<Student> ();`
we determine the specific kind of list – ArrayList.

This points out another advantage of interfaces which you will learn if and when you study Object-oriented Design: It is better to program to an interface rather than an implementation whenever possible.

While we are on the subject of interfaces in the Java class Library, take a look at Iterator; you'll see that it is an interface, not a class. That's ok because nowhere do we instantiate Iterator - we always obtain an instance from a collection, using the method `iterator()`. The particular kind of Iterator obtained is determined by the collection, but that is of little concern to us; we just use it as some kind of Iterator.

6.7.3 Exercises

1. Point out the syntax error, if any, in each of the following (refer to Figure 6.15):

   (a)
   ```
   public class SeniorStudent
                extends GradStudent, Undergrad
   {
     // 5 year BSMS program
     private boolean fiveYear;

       public SeniorStudent (String name, String ssn,
                             boolean fiveYear)
       {  super (name, ssn);
          this.fiveYear = fiveYear;
       }
   }
   ```

   (b)
   ```
   public interface Administrator
   {  private int salaryLevel;
      public abstract int getSalaryLevel();
   }
   ```

   (c)
   ```
   public interface Administrator
   {
      public Administrator (String name, String ssn);
      int getSalaryLevel();
   }
   ```

   (d)
   ```
   public interface Administrator
   {
      int getSalaryLevel()
      {   return 17;  }
   }
   ```

   (e)
   ```
   public interface Administrator
   {
   ```

```
   int getSalaryLevel();
}
```

(f) The following code is included in a method in some other class of the same project:

```
Student junior = new Student ("jim", "222");
```

(g)
```
public class MyList extends List
{  private boolean isSorted;    // true only if the elements are
                               // in increasing order.

   /** @return true only if the elements of this MyList
    *  are in increasing order
    */
   public boolean getSorted()
   {  return isSorted;  }
 }
```

2. Show the class diagram, similar to Figure 6.15, which would result from the following class and interface declarations (the fields and methods of each are not shown). Be sure to designate interfaces as such to distinguish them from classes.

```
public class Mammal
{  ... }

public class Fish
{  ... }

public interface Swimmer
{  ... }

public class Whale extends Mammal implements Swimmer
{  ... }

public class Guppy extends Fish implements Swimmer
{  ... }
```

3. Show the class and interface declarations (no need to show any fields nor methods) corresponding to the class diagram shown in Figure 6.16.

4. Arrange the following concepts into an appropriate class (or interface) hierarchy, and show the class or interface declarations and the class diagram,

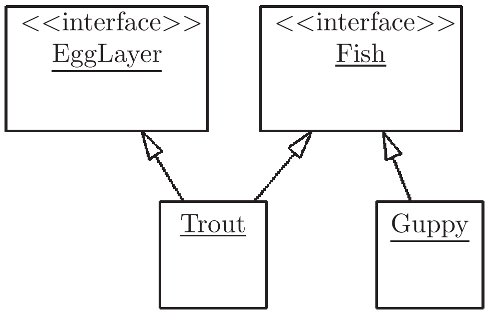


Figure 6.16: Class diagram showing multiple inheritance with an interface for exercise set 6.7

as shown in the previous two exercises. Each interface must have at least one implementing class.

`Bear, Fish, Mammal, DangerousAnimal, Shark`

6.8 Inheritance and Polymorphism in the Grid-World case study

6.9 Projects

1. Use the project `dome` from the code repository. The word 'dome' as used here is an acronym for 'database of multimedia entertainment'. In this project we allow the user to maintain information on a collection of CD's, DVD's, etc. In the project that we start with there are three classes:

   - `Database` – This class stores a List of CDs (Compact Disks) and a List of DVDs (Digital Video Disks). There are methods which allow the user to:
     - Add a CD to the List of CDs
     - Add a DVD to the List of DVDs
     - Print all the CDs and DVDs currently stored
   - `CD` – This class stores information for one CD:
     - The title of the CD (String)
     - The performing artist on the CD (String)
     - The number of tracks on the CD (int)
     - The playing time of the CD (int)
     - Whether we currently own this CD (boolean)
     - A comment providing other information on this CD (String)

     There are accessor and mutator methods for some of the above fields, and a method which will print the CD.

- `DVD` – This class stores information for one DVD:
    - The title of the DVD (String)
    - The director of the movie on the DVD (String)
    - The playing time of the DVD (int)
    - Whether we currently own this DVD (boolean)
    - A comment providing other information on this DVD (String)

  There are accessor and mutator methods for some of the above fields, and a method which will print the DVD.

We wish to make some improvements, in stages, to this project.

(a) Make a list of fields and methods which the CD and DVD classes have in common. Factor out these fields and methods to a superclass called `Item`. You will need some accessor methods in the Item class. Be sure to include a constructor in the Item class, and make the appropriate changes to the constructors in the CD and DVD classes. Caution: The print methods in CD and DVD are different, so don't factor them yet.

(b) Change the Database class so that instead of storing two Lists, it stores one List of Items. The Database class will have a method to add any Item to the database. The Item class will need a print() method; for now, give it a print() method with an empty body. Test your work by creating a Database object and adding several CDs and DVDs to it. Then invoke the list() method to print everything in the database.

(c) The print() method in CD and DVD have some duplicated code which needs to be factored to the Item class. The print() method in each class should print only the fields in that class. The print() method in the Item class can be invoked

(d)

2. This project involves the simulation of traffic in a city. We wish to simulate the activities of vehicles, such as busses, taxis, and cars. We also wish to include people who will walk to and from bus stops, or be picked up by taxis.

   We will start with a few simple classes:

   - Location - This class encapsulates locations within the city. At this point it will consist of a square grid, so a Location has x and y coordinates, but future implementations could involve actual city streets. As entities move in the city they can move to any adjacent Location. This class should have the following capabilities:
       - Construct a new Location with random x and y values.
       - Construct a new Location with given x and y values.

  - Find the distance from this Location to another Location (i.e. the number of distinct moves to get from one to the other).
  - Determine which way to go in order to move closer to a given target Location.

- Passenger - This class deals with Passengers who will be riding on various kinds of Vehicles (such as Busses). A Passenger has a current Location and a destination Location. This class should have the following capabilities:
  - Produce a new Passenger with random origin and destination.
  - Move toward the nearest bus stop, if not on a vehicle.
  - Move toward destination after exiting a vehicle.
- Vehicle - An abstract class which is a superclass of Bus, Taxi, Car, etc. A Vehicle has a capacity (i.e. number of passengers it can accommodate), a current Location, a destination Location, and possibly a speed. This class should have the following capabilities:
  - Move closer to its destination Location.
- Bus - A Bus is a Vehicle. It should have:
  - a List of Locations which are the Locations where it stops to pick up or discharge Passengers. The Bus should move from one stop to the next, in a continuous circle (after the last stop in the List has been reached, it should then proceed to the first stop).
  - It should have a List of Passengers who are riding on the Bus.

  This class should have the following capabilities (in addition to the capabilties inherited from Vehicle):
  - Pick up passengers waiting at a bus stop.
  - Discharge passengers who have arrived at their stop.
  - Provide the nearest stop to a given Location.
- Actor - An Actor is anything which acts, or takes part in the simulation. This includes Passengers and all Vehicles. Since a Bus is a Vehicle, and a Bus is an Actor, Actor will have to be an interface. It will have only one method - `void act()`. Each class which implements Actor will have its own implementation of the `act()` method.
- Simulation - This class will initialize and drive the simulation. It will have a List of Actors and possibly a List of Passengers who have been generated during the Simulation. This class will have methods to:
  - Initialize the Simulation with one or more Vehicles. Each of these should be added to the List of Actors.
  - Run the simulation for one step. On each step, the Passenger class should be given the opportunity to create a new Passenger. If one is created, it should be added to the List of Actors (and Passengers, if necessary). Then each Actor in the simulation

should be told to act. For Passengers this could mean moving toward a Bus stop or toward a destination. For Vehicles this could mean moving toward a destination. For Buses this could mean picking up or discharging passengers if it is at a stop.

– Run the simulation for a given number of steps.
– Provide the nearest bus stop to a given Location.
– Provide a List of Passengers at a given Location.

To test your simulation, print all the Actors on each step to see if they are behaving in a sensible way (this means you will need `toString()` methods).

These specifications allow for some leeway in implementation, and no two solutions will be the same. If time permits, you can extend the simulation to include Cars, Taxis, Bikes, Pedestrians, etc.

4.

Chapter 7

Maps, Collections Revisited

7.1 Fast look-up

One of the most important functions of any computer system is to store large quantities of data, and provide quick access to any portion of that data storage. The ability to store a lot of data is of little value if we cannot access what we need quickly.

Consider the young researcher who is visiting the New York Public Library (one of the world's largest) and needs the answer to a question: What is the relationship, if any, between per-capita income and suicide rate across countries in the world? Our young friend has been assured that the information needed to answer the question is stored somewhere in that library, but how can he/she find it? One solution would be to walk to the nearest stack and read through every book on the shelf, then proceed to the next stack, and so on until he/she eventually finds the necessary information or determines that it is not to be found in all the volumes of that huge library.

This strategy is probably doomed to failure from the start; the researcher will not find what he/she is looking for in his/her lifetime, and will probably tire of the task long before that. The point is that smart search methods are critical to the information retrieval problem. Consequently the stored data must be organized in such a way that it can be searched quickly.

7.1.1 Exercises

1. Use a hard-copy dictionary to find a solution to one of the following problems:

    (a) Find a word which means "(adj) Covered or marked with numerous shallow depressions, grooves, or pits."

    (b) Find the definition of the word *wollasonite*.

2. In 1945 the author Max Shulman published a collection of humerous short stories titled *The Many Loves of Dobie Gillis*. The title character was a university student who spent more time chasing after women than he did studying. In one of the stories, "The face is familiar but...", Dobie is introduced to a beautiful girl at a dance but doesn't hear her name. He spends the rest of the evening trying to get her name, but fails at every attempt. He takes her home that evening, and she gives him her phone number; he now knows her home address and phone number but not her name. Among the several tactics that Dobie uses to discover this girl's name is the following:

   He approaches a pledge to his fraternity named Ed and says:

   > Varlet, I have a task for you. Take yon telephone book and look through it until you find the name of the people who have telephone number Kenwood 6817.

   [This was in the days before cellular phones.] The story continues, narrated by Dobie Gillis:

   > In ten minutes Ed was in my room with Roger Goodhue, the president of the fraternity. "Dobie", said Roger, "you are acquainted with the university policy regarding the hazing of pledges... You know very well that hazing was outlawed this year by the Dean of Student Affairs. And yet you go right ahead and haze poor Ed."

   (a) Explain why Dobie Gillis was accused of hazing a pledge.

   (b) Would Dobie have been accused of hazing if he had told the pledge to search for the girl's home address instead of her phone number?

7.2 Sequential search

The strategy of searching by starting at the beginning (of a list, for example) and examining every item until you find the one you are looking for is called a *sequential search*. For relatively small lists it is not unreasonable, and it is easy to implement. The following method will return the position of the target in a list of numbers, or -1 if the target is not found:

```
/** @return Position of target in the given list, or -1 if not found
 */
public int sequentialSearch(List <Integer> numbers, int target)
{  int pos = 0;
   for (int n : numbers)
     {  if (n == target)
           return pos;          // found the target
        pos++;
```

```
    }
  return -1;                    // target not found
```

If the size of the list is less than a few million, this will not take long. However, for much larger lists a sequential search will not be acceptable. The algorithms which may be involved for more effective searching are discussed in chapter 12 In addition, Java provides some fairly clear classes which can be used to access data quickly.

7.2.1 Exercises

1. If it takes 5 nanoseconds for one iteration of the loop in the `sequentialSearch` method shown above, how long would it take to search a list of 50,000 numbers:

   (a) In the best case (the target is the first number in the list)?

   (b) In the worst case (the target is not in the list)?

   (c) In the average case (the target is in the list, but could be anywhere in the list)?

   Hint: A nanosecond is 10^{-9} sec.

2. Assuming that `nums` is a List of 1250 numbers, given the following code, how many times will the comparison in the `sequentialSearch` method (`if (n == target)`) be executed?

```
int countMissing = 0;
for (int i=0; i<10000; i++)
    if (sequentialSearch(nums, i) == -1)
       countMissing++;
```

7.3 Java maps

In an English dictionary we have a list of words, with a definition for each word. We will call each word, together with its definition, an *entry* in the dictionary. It is easy to look up a word in the dictionary to find its definition, but it is very difficult or time consuming to look up a definition. For example, what is the word which means 'A fruiting body or the stalk of a fruting body in a fungus?' You can find it by starting on page 1 and looking at the defintion of each word until you arrive at this definition. This is a sequential search and would probably take a long time. The words in a dictionary are called *keys*. The definition of a word is called a *value*. When using the dictionary, we always search for an entry using its key, and we never search for an entry using its value. There are a few classes in the Java Class Library which give us a key-value look-up capability, and they are all implementations of the *Map interface*. Maps can be found in the java.util package, along with collections, though strictly speaking a Map is not a collection. Looking at the API for the Map interface we see that there are methods which allow us to add an entry to a map, search a map using a key, remove an entry from a map, etc. As with lists and sets, we can specify the type of the items stored; however, with maps we will specify two types: the type of the keys (K) and the type of the values (V). Each entry in a map consists of a key-value pair. Here are a few of the most useful methods which are available for all maps:

- Put a new entry into a map. If the key of the new entry is already in the map, the new value replaces the existing value, and the old value is returned.

  ```
  /** Put the given key-value pair into this Map.
   *  If the key is already in this Map, replace the existing
   *  value with the*  given value.
   *  @return The existing value for the given key,
   *  or null if the given key is not found in this Map.
   */
  V   put (K key, V value);
  ```

  Get a value from a map. Provide a key to obtain its corresponding value.

-
  ```
  /**
   *  @return The value corresponding to the given key, or null if the
   *  given key is not found in this Map.
   */
  V   get (Object key);
  ```

  Note that the parameter need not be of any particular type.

- Determine whether a given key is in this Map.

  ```
  /**
   *  @return true only if the given key is in this Map.
   */
  boolean   containsKey (Object key);
  ```

  Note that the parameter need not be of any particular type.

- Remove the entry with the given key from a map.

  ```
  /** Remove the entry with the given key from this Map.
   *  @return The existing value corresponding to the given key,
   *  or null if the given key is not found in this Map.
   */
  V   remove (Object key);
  ```

  Note that the parameter need not be of any particular type.

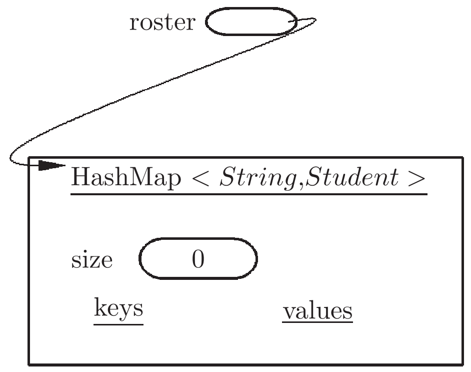


Figure 7.1: An empty map in which the keys are Strings and the values are Students

- Determine the number of entries in a map.

```
 /** @return The number of entries in this Map.  */
int size();
```

- Obtain a set of all the keys in a map.

```
/**
 *  @return All the keys from this Map, as a Set.
 */
Set<K> keySet();
```

In a map, the keys must be *unique*; i.e. there cannot be two entries with the same key. This should be evident from the description of the `put` method above. On the other hand there may be several entries with the same value.

We can further explain the structure of a map by looking at object diagrams. In an object diagram we will show the size of a map, as we did with lists and sets, at the top of the object. The size of a map is simply the number of entries contained. This is followed by two columns: the left column is for the keys, and the right column is for the corresponding values. As with lists and sets, we will treat wrapper classes and Strings as primitives, showing their values directly rather than as references to other objects.

Figure 7.1 shows a newly created map which is empty. No entries have been put into this map. The keys in this map are student numbers (as Strings), and the corresponding values are Students. The variable `myMap` does not store a null reference; it stores a reference to the map object, which has a size of 0.

Figure 7.2 shows the object diagram for a map into which three entries have been put. The size is now 3, and each of the three entries consists of a key (a String) and a value (a reference to a Student).

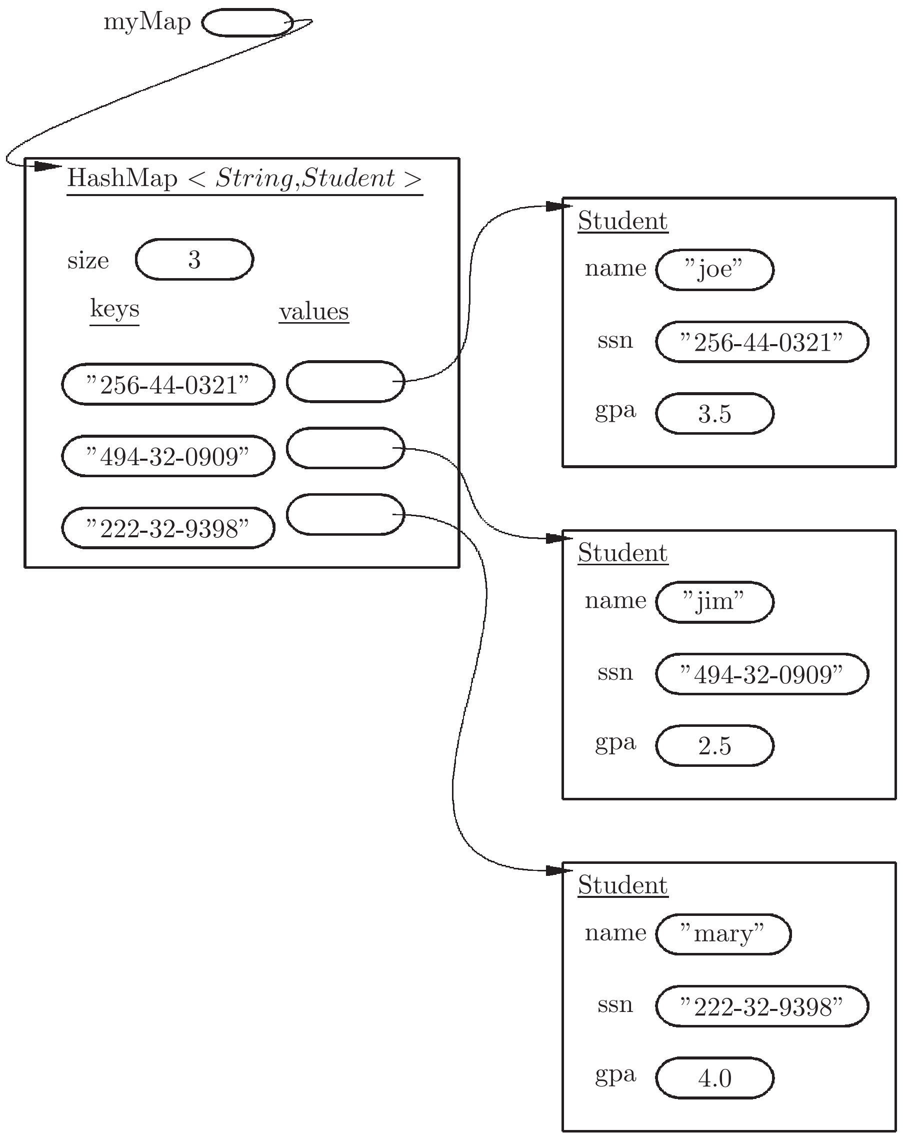


Figure 7.2: An object diagram showing the value of the variable `myMap` storing a reference to a map, after three entries have been added

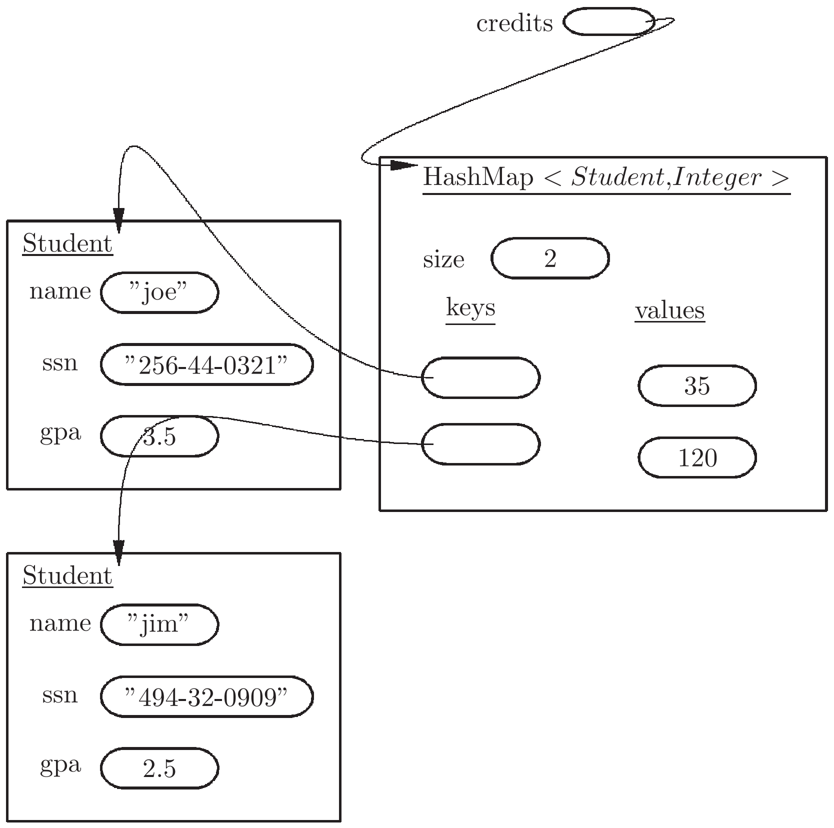


Figure 7.3: An object diagram showing the value of the variable `credits` storing a reference to a map, in which the keys are students and the values are the number of credits accrued, after two entries have been added

Note that the keys need not be primitive types and the values need not be reference types. Figure 7.3 shows the object diagram for a map in which the keys are Students and the values are the number of credits accrued by the corresponding Student. Two entries have been put into this map.

7.3.1 Exercises

1. What value is returned by the `put` method when the key of the entry bein put into a map is not already in the map?

2. What is the effect on a map when the key of the entry being put into the map is already in the map? In this case what value is returned by the `put` method?

3. What value is returned by the `get` method when the key is not in the map?

4. What method (other than `get`) can be used to determine whether a given key is already in a map?

5. What is returned by the `remove` method when removing an entry from a map?

6. True or false:

   (a) The values in a map must be unique.

   (b) The keys in a map must be unique.

   (c) The size of a map is twice the number of entries because each entry consists of a key and a value.

7.4 Examples of methods which use maps

Before we see how to instantiate maps, we will show a few examples of methods which make use of maps. These methods assume the map has already been created and may contain several entries.

```
/** @return The name of the student with the given ssn,
    or "NOT FOUND"  if the student is not in the given map.
    @param roster is a map in which the key is an ssn,
    and the value is the corresponding Student.
 */
public String getName (String ssn, Map<String,Student> roster)
{
    String result = roster.get(ssn);
    if (result == null)
       return "NOT FOUND";
    return result.getName();
}
```

In this example each entry in the map named `roster` stores an ssn as key, and a reference to the corresponding Student as the value. This method uses the `get` method to extract the Student with the given ssn. If the ssn is not in the map, a null reference is returned, and the method shown here checks for null and returns the String ”NOT FOUND” in that case. Otherwise it will invoke the getName() method on the Student, to obtain the Student’s name.

In the next example we show a method which will count the number of students in the given map who have a perfect GPA.

```
/** @return the number of students in the map roster who
 *  have a perfect 4.0 GPA.
 */
public int perfectCount (Map <String,Student> roster)
{  Set <String> ssns = roster.keySet();    // set of all keys in the map
   int count = 0;                          // result
   Student student;

   for (String ssn : ssn)
      {  student = roster.get(ssn);        // get the student
         if (student.getGPA() == 4.0)
            count++;
      }
   return count;
}

In this method we use the \texttt{keySet()} method to obtain a Set
of all the keys in the given map.
Then we use a for-each loop to cycle through the set of ssns,
incementing the \texttt{count} variable each time we encounter a Student
with a GPA of 4.0.
If the map is empty, the loop repeats 0 times, and the returned
value is 0.
Note that the call to \texttt{roster.get(ssn)} cannot return a null
reference, because the ssn was obtained from the map in the first place.
```

7.4.1 Exercises

1. Define a method which will count the number of occurrences of a student with a given name in a given map in which the keys are students' ssns, and the values are the corresponding students.

```
/** @param roster Is a Map in which the keys are ssns, and the
                  values are the corresponding students.
 *  @param name  Is the name of a student which may be in the given map.
 *  @return The number of students in roster with the given name.
 */
public int countNames (Map <String,Student> roster, String name)
```

2. Define a method which will return the number of ssns in the given list occur in the given map. Do not use the `get` method.

```
/** @param roster Is a Map in which the keys are ssns, and the
                  values are the corresponding students.
 *  @param ssns  Is a List of ssns.
 */
public int countSSNs (Map <String, Student> roster, List <String> ssns)
```

3. We wish to count the number of occurrences of various words in some text. We will use a map to accomplish this; the keys will be Strings (i.e. the words) and the corresponding values will be Integers (the number of occurrences of the corresponding word). For example, if the text is `"I yam what I yam"` then the map will contain the following information:

   ```
   key           value
   ---           -----
   "I"           2
   "yam"         2
   "what"        1
   ```

   Define a method named `update` which will put a word into a given map, so as to tabulate the distribution of words as shown above. (To create the map shown above, your method would have been called 5 times, once for each word to be entered)

   ```
   /** Include the given word into the distribution map.
    *  @param word An (additional) word to be included in the map
    *  @param distribution A Map storing the number of occurrences of each of
    *         several words.
    */
   public void update (String word, Map <String, Integer> distribution)
   ```

   Hint: Before putting an entry in the map, check to see whether the word to be entered is already in the map.

4.

7.5 Instantiating maps

We are now ready to create maps. There are two kinds of maps available in the Java class library: *HashMaps* and *TreeMaps*. Both of these implement the Map interface. As with lists and sets, when *declaring* a variable, we will simply call it a Map, and this means that it is capable of storing a reference to some kind of Map. When *instantiating* an object, we will have to decide what kind of Map it should be.

7.5.1 HashMap

HashMap is a class in the package java.util which implements the Map interface. It is designed to provide quick access to any of its entries if you provide the key to the entry you are seeking. The order in which the entries are stored is determined by the HashMap class, and is not likely to be the same order in which the entries were put into the map. In this respect a HashMap is similar to a HashSet.

To declare a variable which can store a reference to any kind of Map, use the following format:

`Map <keyType, valueType> variableName;`

To instantiate the Map as a HashMap, and store the reference in the variable:

`variableName = new HashMap <keyType, valueType> ();`

The following code shows how to declare a variable, create an instance of a HashMap, and put three entries into it. The keys of this Map are Strings and the values are Students:

```
Map <String,Student> roster;        // roster is null
roster = new HashMap <String,Student> ();   // roster is not null
                                    // size of roster is 0 entries
Student st = new Student ("jim","254-33-3221");
roster.put ("254-33-3221",st);      // size of roster is 1 entry
st = new Student ("sue","873-34-856");
roster.put ("873-34-8563", st);     // size of roster is 2 entries

System.out.println (roster.get("873-34-8563"));     // prints sue
System.out.println (roster.get("999-32-2222"));     // prints null
                                                    // key not found

st = new Student ("sueAnn","873-34-8563", st);      // same ssn
st = roster.put (st.getSSN(), st);     // size of roster is still 2 entries
System.out.println (st);               // prints sue
System.out.println (roster.get ("873-34-8563");      // prints sueAnn
```

Note that using the `put` method does not necessarily increase the size of the map. When sueAnn was put into the map, her ssn was already in the map (as a key). So the reference for the corresponding value was simply replaced. In this case the `put` method returned a reference to the old value, sueAnn.

7.5.1.1 HashSets revisited

In chapter 5 we pointed out that when creating a set of objects, those objects must have two methods defined:

- `boolean equals (Object obj);`

  This method is needed because the items in a Set must be unique. When you attempt to add an item to a Set, the implementation (e.g. HashSet) needs to compare the item you are adding with the items already present in the Set to make sure there are no duplicates. Suppose you are working with a Set of Students. How can the add method in HashSet determine whether the given Student is already in the HashSet? It will need to compare the given Student with each Student in the Set. But how can equality of Students be determined? This must be decided by the Student class; this is where it is decided whether this Student is equal to some other Student. Since we designed the Student class, we could decide, for example, that two Students are equal if and only if they have the same ssn.

In that case we could define a method in the Student class to determine whether two Students are equal, as shown below:

```
/** @return true only if the ssn of this student equals the
 *  ssn of the other Student.
 */
public boolean equals (Object other)
{  if (! (other instanceof Student)    // is other a Student?
      return false;
   Student otherAsStudent = (Student) other;  // cast to Student
   return this.ssn.equals (otherAsStudent.getSSN());
}
```

Here we are making use of the fact that the String class itself has an `equals(Object)` method. The parameter, `other`, in the above method is declared as Object, so as to be the same as the `equals` method in the Object class. If the parameter `other` is anything but a Student, our method needs to return `false`. That is the purpose of the *instanceof* operator. `someObject instanceof someClass` will return true only if `someObject` is an instance of `someClass`.

- `int hashCode();`

  Unfortunately a complete explanation of the need for this method is beyond the scope of this book. Suffice it to say that HashSets (and HashMaps) use hash tables for implementation, which require the use of a hash code. A good `hashCode()` strategy recommended by Bloch is shown below:

  1. Start with an inital value of 17.
  2. For each field which is used in the equals method:
     (a) If the field is a primitive type, use the field's value as an int.
     (b) If the field is a reference type, use the hashCode of that field.
  3. Multiply the result by 31 and add the field's value to the result

  A hashCode() method for the Student class is shown below:

```
/** @return a value such that two objects have the same
 *  hash code if they are equal, and two objects are
 *  likely to have different hash codes if they are not
 *  equal.
 */
```

```
   public int hashCode()
   {  int result = 17;
      result = 31 * result + ssn.hashCode();
   }
```

With these two methods in the Student class, we can now work with a HashSet of Students:

```
Set <Student> roster = new HashSet <Student> ();
roster.add (new Student ("jim", "343-55-8494"));
roster.add (new Student ("mary", "242-87-5943"));
```

7.5.1.2 HashMaps: Using our own class as a key: `equals(Object)` `hashCode()`

Figure 7.3 is a bit premature; in a HashMap the keys must have the methods `equals(Object)` and `hashCode()`. Now that we have these methods in the Student class, we can work with a HashMap in which the keys are Students.

As an example, suppose we wish to store the number of courses taken by each student. We could use a map in which the keys are Students and the values are Integers. Each value represents the number of courses taken by the corresponding Student. Each time a student registers for a course, we increment the value of that student's entry; each time a student drops a course, we decrement the value of that student's entry. Now that we have `equals(Object)` and `hashCode()` methods we can build the map:

```
public class CourseCounter
{
private  Map <Student, Integer> courseCounts =
         new HashMap <Student, Integer> ();

/** This method is called when a Student registers
 *  for one course.
 */
public void reg (Student st)
{  int count = 1;
   if (courseCounts.containsKey(st))
      { count = courseCounts.get(st);   // number of courses for st
        count++;
      }
   courseCounts.put(st,count);    // increment count for st
}

/** Drop one course for the given Student
 *  @param st A student who is registered for at
 *  least one course.
 */
public void drop (Student st)
{  int count;
   count = courseCount.get(st);
   count--;
   courseCount.put(st,count);
}

/** @return the number of courses for the given Student
 */
public int getCount (Student st)
{  int result = 0;
   if (courseCounts.containsKey(st))
      result = courseCounts.get(st);
   return result;
}

public String toString()
{   return courseCounts.toString();  }
}
```

In the example above when the `reg` method is called the student's course count is extracted from the map, incremented, and put back into the map. When the `drop` method is called, the student's course count is extracted from the map, decremented, and put back into the map.

We could use this class as shown below:

```
CourseCounter cc = new CourseCounter();
Student joe = new Student ("joe", "256-44-0321");
Student jim = new Student ("jim", "494-32-0909");
cc.reg(joe);
cc.reg(jim);         // registered for 1 course
cc.reg(joe);
cc.reg(joe);         // registered for 3 courses
```

The resulting object diagram for `cc` is shown in Figure 7.4

7.5.2 Exercises

1. Find the syntax error, if any, in each of the following:

   (a) `Map <String> myMap;`

   (b)
   ```
   Map <String, Student> myMap = null;
   myMap = new HashMap <Student,String> ();
   ```

   (c)
   ```
   Map <String, Student> myMap;
   myMap = new HashSet <String, Student ();
   ```

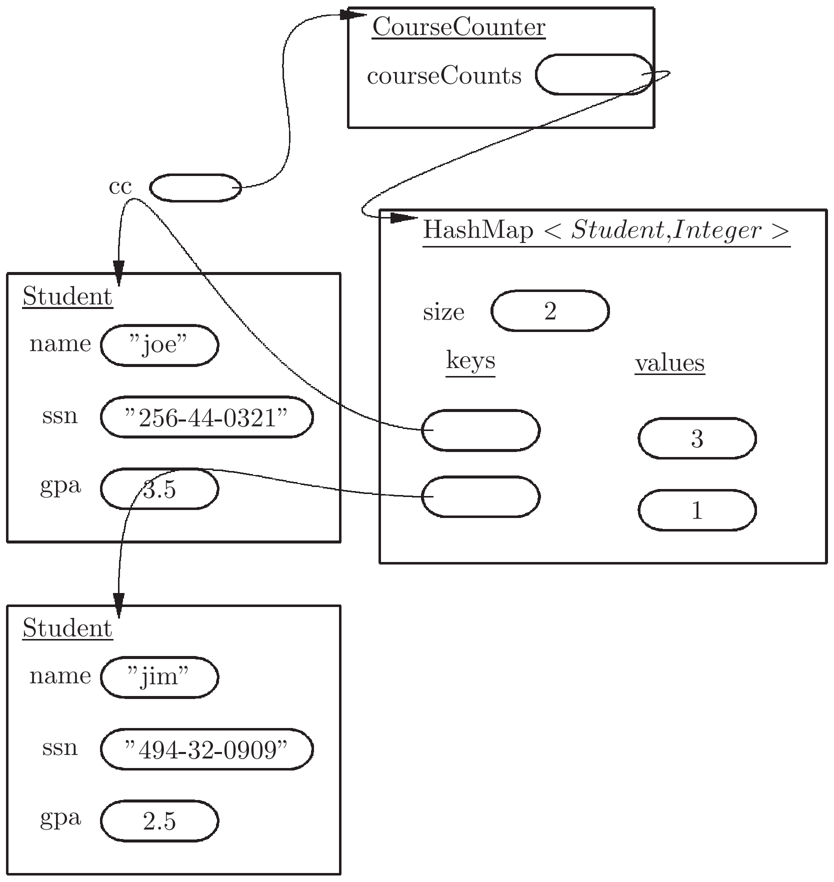


Figure 7.4: An object diagram showing the value of the variable `cc` storing a reference to a CourseCounter object, which stores a reference to a map, in which the keys are students and the values are the number of courses for which the corresponding Student has registered

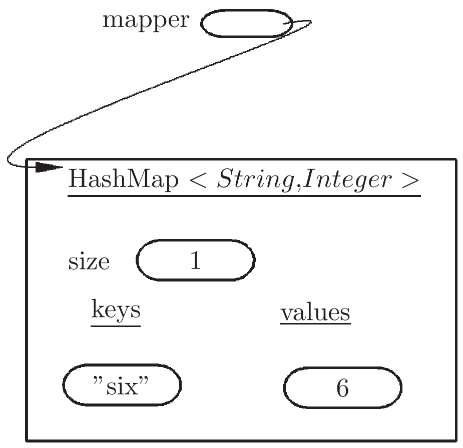


Figure 7.5: Object diagram for a variable storing a reference to a Map (see Exercises)

2. Show the code which will declare a variable which stores a reference to a Map in which the values are Strings and the keys are Integers. It should initialize that variable with a reference to an empty HashMap.

3. Draw object diagrams for the variables *map1*, *map2*, and *map3*, after the code shown below has executed:

```
Map <Integer, String> map1;
Map <Integer, String> map2 = new HashMap <Integer,String>();
Map <Integer, String> map3 = new HashMap <Integer,String>();
map3.put (5, "five");
```

4. Show the code which would have produced the object diagram shown in Figure 7.5.

7.6 TreeMap and Collections revisited: TreeSet and LinkedList

Now that we have a better understanding of inheritance, interfaces, and maps, we are ready to take a look at another class that implements the Set interface, and another class that implements the Map interface.

7.6.1 TreeSets

In chapter 5 we mentioned that there is more than one class in the Java class library which implements the Set interface. In that chapter we discussed Hashsets; we now present another class which implements the Set interface – *TreeSet*.

We mentioned that sets, in general, are not concerned with the order in which its items are stored. The TreeSet, in a sense, contradicts this property; a TreeSet will maintain its elements in increasing order. If we have a TreeSet of numbers, and we iterate through that TreeSet, we will obtain the numbers in increasing order, regardless of the order in which they were added to the TreeSet. Strings would be obtained in alphabetic order. Other than that, TreeSets and HashSets have very similar behavior. They both provide fast access to any item in a set.

To instantiate a TreeSet of Strings, and add three Strings to it:

```
Set <String> names;
names = new TreeSet <String> ();
names.add ("jim");
names.add ("al");
names.add ("mary");
```

If we were to cycle through this set with an iterator, we would obtain the items in the following order:

`"al", "jim", "mary"`

For a TreeSet storing objects of some other class, that class must have a *compareTo* method which can be used to determine whether a given object is less, equal, or greater than another object of the same class.

For example, if we were to create a TreeSet of Students, our Student class would need a `compareTo` method. Here is an example of a `compareTo` method for the Student class, assuming that we wish Students to be ordered by SSN:

```
/** @return  a negative number if this Student precedes the other Student,
             0 if this Student equals the other Student,
             a positive number if this Student follows the other Student.
 *  Students are to be ordered by SSN
 */
public int compareTo (Student other)
{   return ssn.compareTo(other.ssn);    }
```

The `compareTo` method is a standard method found throughout the Java class library. In the comparison `foo.compareTo(other)`, the returned value is negative if foo is less than bar (foo precedes bar) and positive if foo is greater than bar (foo follows bar). The easiest way to remember this convention is to imagine that the compareTo method simply does the subtraction: `foo - bar` (which is what it actually does).

One additional modification is needed in the Student class. The class should implement the `Comparable` interface:

`public class Student implements Comparable<Student>`

Thus, the compiler will require you to include the compareTo method in the Student class, which will permit the client to compare this Student with any other Student. In the implementation of this method you will need to decide how to order Students. You may decide, for example, to use the Student's ssn as the sole criterion for ordering Students, as shown above. The `compareTo` method is needed by `TreeSet` so that the `TreeSet` class can keep the elements in order.

Having included this method in our Student class, we can now create a TreeSet of Students:

```
Set <Student> roster;
roster = new TreeSet <Student> ();
roster.add (new Student ("al", "832-43-4342"));
roster.add (new Student ("mary", "135-34-7839"));
roster.add (new Student ("joe", "135-27-3482"));
```

An iterator would produce these Students in the order `joe, mary, al`.

7.6.2 TreeMaps

One of the methods in the Map API is *keySet*. It returns a Set of all the keys in the map. If we wish to search the entries in a Map, we must first obtain a Set of all keys in the Map. We can then iterate through the Set of keys, using each key to obtain an entry in the Map. The example shown below is a method which will print all Students who have a sufficiently high GPA.

```
/**
  * @param roster is a Map in which the keys are Students' SSNs and
  * the values are the corresponding Students.
  * @param min The minimum GPA required to be considered a good Student.
  * This method will print all Students with a gpa of min or greater.
  */
public void showGoodStudents (Map <String,Student> roster, double min)
{  Set <String> ssns = roster.keySet();   // obtain the set of ssns
   Iterator <String> itty = ssns.iterator();
   String ssn;
   Student st;
   System.out.println ("Students with a minimum GPA of " + min);
   while (itty.hasNext())
     {  ssn = itty.next();
        st = roster.get(ssn);
        if (st.getGPA() >= min)
           System.out.println (st + "\n");
     }
}
```

Notice that this method knows that `roster` stores a reference to a Map, but doesn't know what kind of Map it may be. If it happens to be a HashMap, the sequence in which the Students are obtained with the Iterator is unspecified.

They could be obtained in any order; moreover, if entries are added or removed, the ordering could be totally different.

We now introduce another kind of Map called *TreeMap*. A TreeMap is similar to a TreeSet in that the entries will be ordered according to the `compareTo` method of the Map's keys. If the keys are Strings, the keys will be in alphabetic order. More precisely, they will be obtained in alphabetic order by an Iterator.

If the parameter `roster` in the method shown above happens to be a reference to a TreeMap, the Students will be printed in order of increasing SSN, since the SSN is a String.

7.6.3 LinkedList

In chapter 5 we introduced the List interface, and an implementing class called ArrayList. We also mentioned that there could be other implementing classes for the List interface; i.e. there could be other kinds of Lists. In this section we introduce one such List, called *LinkedList*. Because a `LinkedList` implements List, it has all the methods shown in the List interface: `add, get, set, size, remove, ....`

As an example, we show how to instantiate a LinkedList and add some items to it in the code segment below:

```
List <Integer> grades = new LinkedList <Integer> ();
grades.add (92);
grades.add (88);
grades.add (100);
System.out.println (grades.get(2));
```

In this code segment the only unusual aspect is that we have instantiated a LinkedList instead of an ArrayList.

For the most part, everything you can do with an ArrayList you can also do with a LinkedList, and vice versa. So why is there a need for LinkedList at all? The answer concerns run-time efficiency. These two classes can differ in the time required to work with long lists. When processing a long list, some operations can take a long time for ArrayLists, but a short time for LinkedLists, and vice versa.

To decide which kind of List should be used in a particular application, we offer the following general guidelines:

- If the size of the list will be changing (increasing and decreasing) as the program executes, LinkedList will be faster than ArrayList.
- If the size of the list will remain stable, ArrayList might be faster than LinkedList.
- The `get` and `set` methods are slow for LinkedList.
- The `add` and `remove` methods are slow for ArrayList (they both change the size of the list).

Operation	ArrayList	LinkedList
get(int inx)	fast	slow
set(int ndx, E item)	fast	slow
add(E item)	ok	fast
add(int ndx, E item)	slow	fast
remove (int ndx)	slow	fast

Figure 7.6: Relative efficiency of operations on ArrayLists vs. LinkedLists

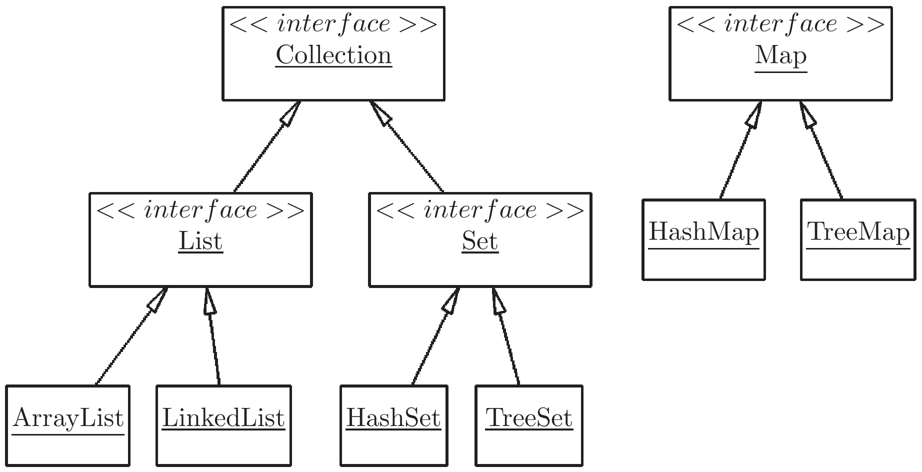


Figure 7.7: Class diagram showing some of the interfaces and classes in the package java.util

- When working with a variable declared as `List` and the actual kind of List is unknown, use an Iterator to cycle through the items of the List.

These performance characteristics are summarized in Figure 7.6. The `add (E item)` method adds an item at the end of a List. The figure indicates this operation is 'ok'. By this we mean that adding at the end of an ArrayList will generally be fast; there might be occassions when it would slow down a little, but this is not a major concern. However, when inserting an item at an arbitrary position in a List – `add(int ndx, item)` – LinkedList is clearly faster than ArrayList. You will understand why these performance characteristics are as shown here, and you will be able to describe them mathematically, when you study Data Structures.

Now that we have taken a look at several of the classes in the java.util package of the Java class library, we can examine the way in which they relate to each other. The best way to do that is with a class diagram. Figure 7.7 shows a class diagram for some of the classes in the Java class library (Not shown is the `Object` class which is a super-class, directly or indirectly, of all classes).

7.6.4 Exercises

1. Point out the syntax error, if any, in each of the following (assume the Student class implements Comparable¡Student¿):

   (a) `Set <String> names = new Set <String> ();`

   (b) `Set <String> names = new TreeSet <Student> ();`

   (c) `Map <Integer,Student> grades= new TreeMap <Integer, Student> ();`

   (d) `List <Integer,Student> grades = new LinkedList <Integer, Student> ();`

2. Define a method which, given a Set of Students, will print all Students whose names are longer than 10 characters. In what order will those students be printed?

```
/** Print all Students in roster whose name is longer than 10
 *  characters.
 */
public void showLongNames (Set <Student> roster)
```

3. Define a method similar to the previous problem in which the parameter is a Map rather than a Set. In what order will those students be printed?

```
/** Print all Students in roster whose name is longer than 10
 *  characters.
 */
public void showLongNames (Map <String, Student> roster)
```

4. Define a method which, given a Set of Students, will print all the Students in order, by ssn, one student per line. Assume the `compareTo` method in the Student class orders Students by ssn.

```
/** Print all Students in roster whose name is longer than 10
 *  characters.
 */
public void showBySSN (Set <Student> roster)
```

   Hint: Copy the Students to a TreeSet and print the TreeSet.

5. Define a method which, given a list of integers, will return the largest integer in the list. Be concerned with the efficiency (i.e. execution time) of your method.

```
/** @return The largest value in numbers.
 */
public int largest (List <Integer> numbers)
```

6. Define a method which, given a List of Students, will return a List of those with a perfect GPA (4.0). Be concerned with the efficiency (i.e. execution time) of your method.

   ```
   /** @return A List of all Students in roster who
    *  have a perfect 4.0 GPA.
    */
   public List <Student> perfect (List <Student> roster)
   ```

7. For each of the following determine whether the List involved should be an ArrayList or a LinkedList (or whether either is ok), for purposes of efficiency:

   (a) A method which, given a List, determines whether the elements are in ascending order.

   (b) A method which, given a List of items, obtains other items from the user's keyboard, and, one at a time, deletes each of those items if found in the given list.

   (c) A method which, given a List of items in ascending order, obtains other items from the user's keyboard and inserts them at the appropriate places in the given List.

   (d) A method which, given a List of items, will arrange those items in ascending order. The given List may contain duplicate values.

7.7 Projects

1. Use the project `university` for this project. It should contain the `Student` class which we have been using. We wish to implement a simple information system for a university in this project. Define a new class named `UniversityInfoSys`. This class should maintain information on all students admitted to a university. It should have a field which stores all the students in a Map in which the keys are Strings (ssns) and the values are Students. In addition to a constructor, it should have the following methods:

   - A method to add a new student to this university.

     ```
     /** Add the Student st to this UniversityInfoSys
      *  @return false If not added (e.g. already in the system)
      */
     public boolean addStudent(Student st)
     ```

   - A method to obtain a reference to a list of all students who are active at this university.

```
/** @return a List of all Students in the system
 */
public List <Student> getStudents()
```

- A method to search for a particular student, given the student's ssn. This method should not involve a loop.

```
/**
 * @return the student with the given ssn, or
 * null if not found.
 */
public Student searchBySSN(String ssn)
```

- A method to search for all students, who have a given name.

```
/**
 *  @return the set of students with the
 *  given name. Must match exactly, case
 *  sensitive.
 */
public Set <Student> searchByName(String name)
```

- A method to find the average GPA of all students at this university.

```
/** @return Average GPA of all students
 */
public double averageGPA()
```

- A method to remove those students whose GPA has fallen below a given minimum value.

```
/** Remove all students who have a GPA less than minimum.
 *  @return Number of students dismissed.
 */
public int dismiss (double minimum)
```

2. In a new project, we wish to develop a class which will enable us to encrypt and decrypt secret messages. We will use a few maps for this purpose. Cryptologists call this kind of encryption scheme a *codebook*. Open the project `crypto` from the repository for this chapter. Define methods to:

   - Build a map which stores English words as keys, and the corresponding encrypted words as the values. Make up your own entries, such as:

plain text	cipher text
the	spritz
fox	glmph
jump	foo

   - Build the inverse map, i.e. the map in which the keys are encrypted (i.e. cipher) words, and the corresponding values are real English words (i.e. plain). Build this map automatically from the other map, using a loop.

- Given a List of English words, return a List of the corresponding encrypted words.
- Given a List of encrypted words, return a List of the corresponding English words.
- A method to test your work is provided. It encrypts the words "I have things to do", producing cipher text. It then decrypts the cipher text to produce the original plain text.

3. You have intercepted a message sent by the enemy, and you need to decode it to save us from attack. All you know is that the enemy is using a permutation cipher; each word in the intercepted message is simply a permutation (i.e. an anagram) of an English word. The message is:

```
niaiuanmrisotzrtiiocm fo uealcisonlesm npesoaw tpso nialtrenosotiaertuc
```

Use the project `unscramble` from the repository. In this project a class named `WordProduce` is provided. It has methods which will provide you with over 10,000 English words, one at a time, each time you call the `getWord()` method. This class also has a boolean method which will tell you whether there are more words yet to be obtained. Use this method to control a loop in which you call `getWord()`.

The strategy here is to use a map in which the keys are Strings in which the characters are in alphabetic order. The corresponding values are sets of Strings which are the anagrams of the keys. For example, one entry in the map could be:

```
arst = {"arts", "rats", "star", "tars", "tsar"}
```

Here is what you'll need to do:

(a) In the MapBuilder class implement a method to build the map, using wordProducer. It should return a reference to the map.

Hint: In the String class there is a method which will produce an array of chars from a String. In the Arrays class there is a static method which will sort an array of chars.

(b) In the Unscrambler class:

i. In the constructor instantiate MapBuilder, and use it to create the map.

ii. Define a method which will get the anagrams of a sorted String from the map.

iii. Define a method which will show all the anagrams of each word in a given list of words.
iv. Define a method which will allow you to test what you have done. See if you can decode the secret message shown above.

Chapter 8

Exceptions - Handling Errors

Unfortunately, most software projects of appreciable size are not flawless. Because of the complexity of software, it is not unusual to encounter incorrect behavior (bugs) when using software. If the project is viable, the bugs are corrected as they are encountered and reported; over time the quality of the software improves. Some software applications which have been widely used for a long period of time are relatively robust. However, getting to that point is a long and difficul road.

Some bugs result in a minor inconvenience to the user; the program comes to a crashing halt and needs to be restarted. In other cases incorrect output can go undetected, until it is too late. There are also safety-critical applications in which bugs are completely unacceptable:

- Software controlling the flight patterns of aircraft at a busy airport.
- Software controlling medical machinery such as X-ray machines, heart-lung machines, intensive care monitor machinery, etc.
- Software used for national defense – early attack warning systems, radar systems, secure international communications software.
- Military applications – drone control systems, missile guidance systems, field communications cryptologic software, etc..
- NASA flight control software.

There are many more applications, too numerous to mention, where we rely for our own safety on computerized systems. Such systems are tested rigorously, and often have fail-safe redundancies built in. Nevertheless, in developing software we will find that debugging is an inevitable part of the process.

In this chapter we examine some of the run-time errors which can occur when a Java program executes. We will also attempt to 'trap' or *catch* those

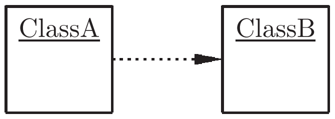


Figure 8.1: Class diagram showing a client (ClassA) and server (ClassB)

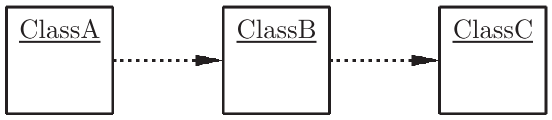


Figure 8.2: Class diagram showing a class (ClassB) which acts as both client and server

errors, and handle them in such a way that the program continues to execute without crashing. In the best of all worlds, if an error does occur, the system will tolerate the error and continue execution, and the the user will be unaware that all this has transpired. We call this *error-handling* or *fault-tolerant computing.*

8.1 Client/Server terminology

Before discussing the handling of errors, we should introduce some terminology having to do with the providing of services. This terminology can refer to classes which provide services to other classes, or to methods which provide services to other methods.

If classA uses the services of classB, we say that classA is a *client* of classB and that classB is a *server* for classB. A server class is simply the one providing one or more services to one or more client classes. This is often indicated in class diagrams as shown in Figure 8.1. A dashed arrow pointing from classA to classB means that classA *uses* classB. In this case classA is the client, and classB is the server. Note that the arrow does *not* represent inheritance; a solid arrow with hollow arrowhead is used to represent inheritance.

Suppose we introduce a third item, classC, and that classB uses the services of classC. The class diagram is shown in Figure 8.2. We now see that the terms 'client' and 'server' are relative to a context. As in Figure 8.1, classA is still a client of classB, and classB is still a server for classA, but now classB is a client of classC, and classC is a server for classB. In other words, classB acts as both client and server.

This terminology can also be applied to methods. If methodA calls methodB, methodA is a client of methodB, and methodB is a server for methodA. As with classes, a method can be both a client and a server (and this is very often the case).

8.1.1 Exercises

1. Refer to the project `university` in the repository for chapter 7.
   (a) Which class is a client and which is a server?
   (b) In the `Student` class which methods are client methods, and which methods are server methods?
2. True or False:
   (a) A *client*, as defined in this section, is a person.
   (b) It is possible for one class to act as both client and server.
   (c) The method `getName()` in the `Student` class is a client of the method `searchByName()` in the `UniversityInfoSys` class.

8.2 Assertions

Before getting to Exceptions, we would like to address the problem of determining the actual location of a program error (bug). Java has the capability of reporting to the programmer the method and line number within the source file where the error took place, along with a brief description of the nature of the error. Older programming languages and run-time systems provided little more than a hexadecimal memory dump, leaving the programmer the rather difficult task of tracking down and fixing the error.

Despite all the information received from the Java run-time environment, finding the actual bug is not always a simple task. Consider a division-by-zero error. This might occur because a parameter passed to a method stored, erroneously, a value of zero. The real programming error took place in the calling method, i.e. the client method (or perhaps the *client's* client). The incorrect line(s) of code could be far removed from the actual line where the error occurred, as shown in Figure 8.3.

Because the actual cause of a run-time error can be buried deep in many levels of method calls, we could simplify the debugging process if we could somehow detect that our program is in a non-valid state before the error actually occurs. In the example above, in the method meth1, if the value of `a` is 4, there will eventually be a division-by-zero error in meth3. We can save ourselves a lot of time if we could detect the problem in meth1, before any other methods are called. This can be done with an *assertion*.

An assertion is a true/false statement about the state of the variables at some point in a method. An assertion *should be true*; if it is false, we'd like to know about it before going any further. Assertions come in two formats; the first format is:

```
assert booleanExpression;
```

The booleanExpression is an expression which evaluates to true or false. If the booleanExpression is true, execution continues as if the assertion did not exist. However, if the booleanExpression is false, the program terminates,

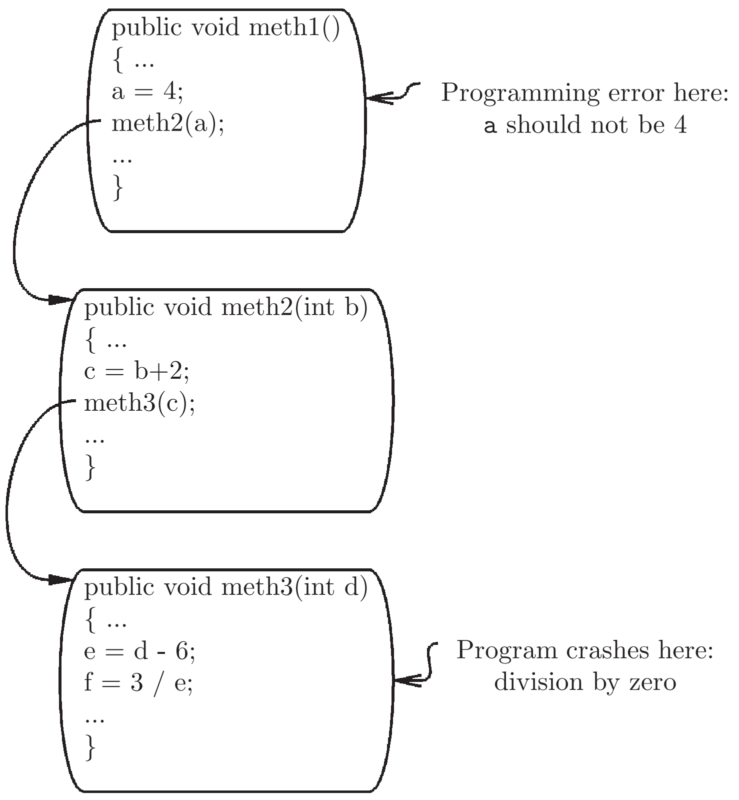


Figure 8.3: A run-time error buried deep in a series of method calls

providing information as to the location of the assertion which failed (name of method, and line number of the source file). In our example, we know that if the parameter `b` in `meth2` is 4, the program will ultimately fail, so we use that in our assertion:

```
public void meth2(int b)
{   assert b != 4;       // b should not be equal to 4
    ...                  // other statements not shown
    c = b+2;
    meth3(c);
    ...                  // other statements not shown
}
```

The second format for assertions allows us to put out more information about the cause of the problem:

`assert booleanExpression :  String ;`

The String should provide some helpful clues as to exactly what has gone wrong; the String need not be a String constant, but may be any expression which evaluates to a String:

```
public void meth2(int b)
{   assert b != 4 : "The value of b should not be 4";
    ...                  // other statements not shown
    c = b+2;
    meth3(c);
    ...                  // other statements not shown
}
```

In this case when the assertion fails the programmer will see the message, "The value of b should not be 4" in addition to the method name and line of the source file for the assertion.

Once we have finished testing our program and are ready to release it to users, we may wish to remove all the assertions. If we do so, and then a bug is encountered, we would have to put all the assertions back in place (a time consuming task).

Instead there is a better solution. Assertions can be disabled or enabled. When assertions are disabled, they are still in the source file, but the compiler does not include them in the output (.class file); in other words it is as if they have been removed. If further testing reveals a bug, we simply enable assertions to help discover the source of the error.

We caution the programmer to avoid doing something like the following:

`assert roster.remove(6) != null :  "Position 6 of roster is null";`

The remove method will remove the item at position 6 of roster, and return the reference at that position, which we compare against null. The problem with this is that when assertions are enabled, the size of `roster` is changed, but when assertions are disabled, the size of `roster` is not changed. The program will exhibit different behavior depending on whether assertions are enabled or disabled – not a a good thing.

In summary, an assertion is a statement about the state of things as the program executes which should be true, and is generally false only if something unexpected or incorrect has occurred. As a college professor, I make the assertion: "All my students work hard and will pass this course". It is my hope that this assertion will always be true, but if it happens to be false, I'd like to understand what has caused it to be false.

8.2.1 Exercises

1. Refer to the `university` project in the repository for this chapter. What will be printed when each of the following code segments is executed?

   (a)
   ```
   String name = "joe";
   Student frosh;
   assert frosh != null : "reference to Student is null";
   frosh = new Student (name, "232-34-9756");
   System.out.println (frosh);
   ```

   (b)
   ```
   String name = "joe";
   Student frosh;
   assert name.length() > 0 : "Name " + name + " is not valid";
   frosh = new Student (name, "232-34-9756");
   System.out.println (frosh);
   ```

2.

8.3 Exceptions

8.3.1 Run-time errors resulting in an Exception

When a Java program is executing, the Java run-time system can detect an incorrect operation and determine that the program needs to be terminated. The particular kind of error is called an *Exception*, and the process in which it occurs is called a *throw*. This can result in program termination, also known as a *crash*. Java has the capability of providing the method and line number within the source file where the error took place, along with a brief description of the nature of the error (Exceptions are similar to assertions; Java assertions are actually implemented using Exceptions). Older programming languages and run-time systems provided little more than a hexadecimal core dump, leaving the programmer the rather difficult task of tracking down and fixing the error.

You may have encountered several examples of Exceptions already:

- `NullPointerException` – This occurs when you attempt to use a null reference inappropriately. In the expression `foo.bar` or `foo.bar()` the variable `foo` must not be `null`; if `foo` is null, a `NullPointerException` is thrown, and the program terminates. Note that comparing a reference will not cause a problem: `if (foo == null) ...` is not a problem. When you get a NullPointerException, always look at the variable to the left of the dot as the likely culprit.

- `IndexOutOfBoundsException` – An array index or List index is not in the correct range. If `grades` is a List of size 5, the range of indices is [0..4] inclusive. Thus, either of the following will cause an IndexOutOfBoundsException:
  - `grades.get(5)`
  - `grades.get(-1)`
- `ArithmeticException` – This is typically a division by zero, though there could be other causes resulting from a calculation that cannot produce a valid result.
- `ConcurrentModificationException` – This results when changing the size of a Collection in a for-each loop. If you need to selectively remove or add items to a Collection in a loop, use an Iterator (a ListIterator will allow you to add items to a List).
- `ClassCastException` – This Exception is thrown when the Java run-time environment is unable to perform the requested cast. For example:

  ```
  Student st = new GradStudent ("jim", "353-33-9303");
  Undergrad under = (Undergrad) st;
  ```

  It is not possible to cast the variable `st` as an Undergrad because its dynamic type is GradStudent.
- `IllegalArgumentException` – This Exception is thrown when a method's parameter has a non-valid value. [1]

These are just a few of the more common Exceptions; they are all described in the API for the Java class library (see, for example, the package `java.lang`). As we will see, Exception is a class which has many sub-classes. Each package can have its own Exception classes.

In addition to the information provided by the Java run-time system, most IDEs provide a *debugger*, which is a utility that allows one to step through the statements of a program one at a time while watching the values of variables change. Though a debugger will not eliminate bugs for you, it is a useful tool that allows you to diagnose a problem and decide on an appropriate fix. Debuggers will be discussed in more detail later in this chapter.

[1] An *argument* is terminology carried over from other programming languages and is essentially the same as a parameter.

8.3.2 Throwing exceptions in a server method

Every public method should have an API which describes pre and post conditions for that method. These pre and post conditions constitute a contract for all client methods: If the pre conditions are satisfied, the post conditions will also be satisfied; if any pre condition is not satisfied, the server method is not under any obligation to do anything. This 'contract' is vital to the proper interaction of methods.

In the case where a precondition is not satisfied (the server method is not able to complete its obligations) the server method may wish to *throw* an Exception. This is a way of signalling that something is drastically wrong, and normal execution cannot continue. The throwing of an Exception differs from a failed assertion in that the client method can handle the Exception and attempt to continue executing the program; the program will not necessarily come to a crashing halt.

To throw an Exception, use a `throw` statement in which you instantiate the Exception being thrown:
`throw new Exception-Name();`
For example: `throw new IllegalArgumentException();`
When this is executed,

- The server method is terminated immediately; no return statement is executed, and even a non-void method is not required to return a value.
- Control returns to the statement in the client method which invoked the server method. The client method can then handle the Exception as described in the next section. If the client method chooses not to handle the Exception, the client method will throw the same Exception, to be handled by *its* client method. If no method handles the Exception, the Java runtime environment will bring the program to a crashing halt.

When instantiating the Exception, the particular Exception being instantiated may have several different constructors. They usually have at least a constructor with no parameters, and a constructor with a String parameter (as in the example above). The purpose of the String is to provide information for the programmer as to why the Exception was thrown. See the API for the Exception class to understand all the options available when instantiating an Exception.

If a server method can potentially throw an Exception, it is essential that this be made clear in the method's API. There is a javadoc keyword, *throws* used for this purpose. Consequently, the programmer writing a client method will understand that an Exception might be thrown, and be prepared for it. As an example, we return to our Student class in which we wish to calculate a Student's GPA:

```
/**
 *  @param gradePoints The number of gradePoints earned by this Student.
 *  @param credits The number of credits earned by this Student,
 *  should be positive.
 *  @throws IllegalArgumentException if credits is less than or equal
 *  to zero.
 */
public void calcGPA (int gradePoints, int credits)
{  if (credits <= 0)
      throw new IllegalArgumentException ("credits is " + credits);
   gpa = gradePoints / (double) credits;
}
```

If and when the IllegalArgumentException is thrown, the gpa is not calculated; instead we are back in the client method, where the thrown Exception can be handled or ignored.

One final caveat on throwing Exceptions is in order. Exceptions should be used only for exceptional, unexpected, or erroneous events. Do not use Exceptions in place of ordinary logic. Do NOT do the following (we have actually seen students do this):

```
// The WRONG way to control execution of a loop
try {
      int i=0;
      while (true)
         {  System.out.println (roster.get(i));
            i++;
         }
    }
catch (IndexOutOfBoundsException ioobe)
   {  }
```

8.3.3 What to do when an Exception is thrown

In this section we address the issue of handling a thrown Exception; we are concerned with the client method here, not the server method. Our client method has called a server method which has thrown an Exception and we need to decide how it should be handled or ignored in the client method:

- Should we ignore the exception? If so, our method automatically throws the same Exception, so that it can be handled in the client method which called our method.
- Should we throw the same Exception explicitly? This has the same effect as ignoring the Exception (but may be required depending on the kind of Exception – see checked vs. unchecked Exceptions, below)
- Should we try to handle the Exception right here in our method? This would involve a *try/catch* block.

8.3.4 Handling exceptions with try/catch in a client method

If we choose to handle a thrown Exception in our client method, we must use a statement called a *try/catch block.* This is a statement with at least two parts: a try block and one or more catch blocks. The format is:

```
try {  // statement(s) containing a call to a method which might
       // throw an Exception.
       // Include all statements which depend on the result of
       // the method call.
    }
catch (ExceptionClass name1)
    {  // Statements to handle the Exception
    }
catch (ExceptionClass name2)
    {  // Statements to handle the Exception
    }
.
.
.
catch (ExceptionClass name)
    {  // Statements to handle the Exception
    }
}
```

The try block includes a statement which calls a server method – the one which potentially will throw an Exception. The try block should also contain all statements which *depend on* the result of the server method.

If an Exception is thrown, control is immediately transferred to the catch blocks. The first catch block to correctly name the Exception is selected, and the statements in its block are executed. Control then falls through to whatever follows the try/catch statement. If none of the named Exceptions match the thrown Exception, this method throws the same Exception (the Exception is not handled here).

As an example, consider the following method which attempts to calculate the GPA of each student in a given List.

```
/**  @param min The minimum GPA needed to be on the dean's list
 *   @return A List of all students on the roster
 *   who qualify for the dean's list.
 *   Uses getGradePoints() and getCredits() to obtain data for
 *   a student.  Ignores students with non-valid data.
 */
public List<Student> deansList (List <Student> roster, double min)
{  List <Student> result = new LinkedList <Student> ();
   int credits, gradePoints;
   for (Student st : roster)
```

```
       {  credits = getCredits(st);
          gradePoints = getGradePoints(st);
          try {
                 st.calcGPA (gradePoints, credits);
                 if (st.getGPA() >= min)
                    result.add (st);
              }
          catch (IllegalArgumentException iae)
              {  System.err.println ("Illegal data for " + st);
                 System.err.println ("This student not processed");
              }
       }
}
```

In the `deansList` method shown above note that the `try` block contains all statements which depend on the result of the call to `calcGPA()`. In this example there is only one `catch` block, but if we knew that `calcGPA` threw other Exceptions we could include additional `catch` blocks. Also note that the try and catch blocks require the curly braces, even if there is only one statement in the block.

8.3.4.1 Checked and unchecked Exceptions

As with most classes in the Java class library, Exception classes form a hierarchy of sub-classes. This hierarchy is critical to understanding the requirements placed on a program by the compiler. The compiler will *require* handling certain Exceptions and will allow other Exceptions to be ignored.

Some selected Exception classes (and anything which can be thrown) from the Java class library are shown in Figure 8.4. In this diagram a `Throwable` is a super-class representing anything that can be thrown (at this point the only things we can throw are Exceptions). An `Error` represents an error in the Java class library or run-time system; it is safe to say that you will not encounter this error. This software has been thoroughly tested by millions of users for many years. If you do encounter this error, rest assured it is not your mistake, and you will have to find a work-around (report this to Oracle).

We should concern ourselves with the Exception class in Figure 8.4 and its sub-classes. These classes fall into two categories:

- Unchecked Exceptions – `RunTimeException` and all of its sub-classes are said to be *unchecked*.
- Checked Exceptions – All other sub-classes of `Exception` are said to be *checked*.

The difference between checked and unchecked Exceptions has to do with whether the compiler allows you to ignore them:

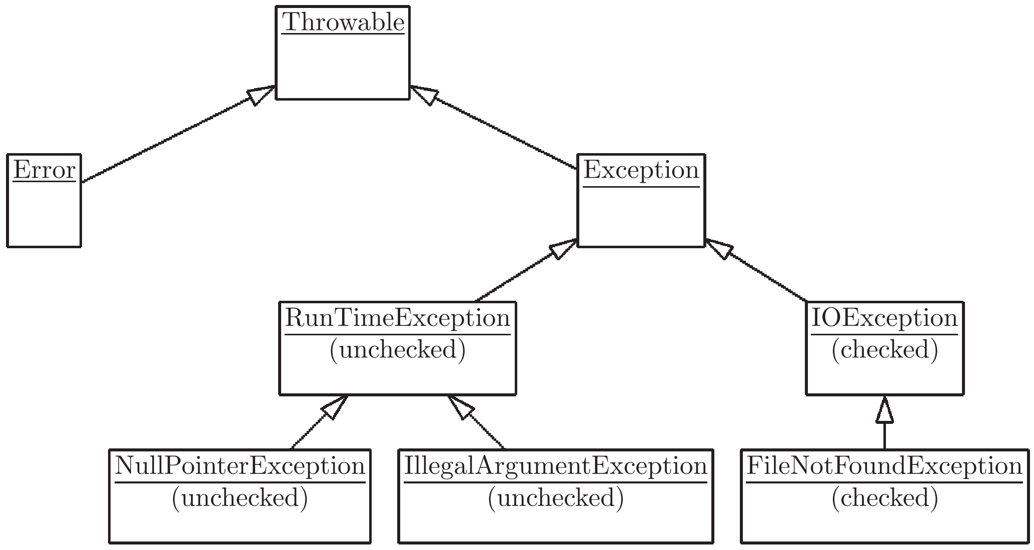


Figure 8.4: Class diagram showing some of the Exception classes in the Java class library

- Unchecked Exceptions – The client method is not required to check for any unchecked Exceptions that might be thrown by server methods. It is permissable to use a try/catch block, but not required.
- Checked Exceptions – The client method is required to check for any checked Exceptions that might be thrown by server methods. In this case, the client method has two choices:
  - Use a try/catch block to handle the Exception as discussed previously.
  - Declare, in the method signature, that the client method *throws* the Exception. In this case the method which called the client method is faced with the same requirement (we call this 'passing the buck'). The API of the client method should also have an @throws line describing the circumstances under which this Exception would be thrown.

Why would the client method wish to pass the buck rather than handling the Exception? It could be that the client method does not have enough information to handle the Exception appropriately, but the method which called it would be better able to handle the Exception. Thus Exceptions can be propagated from any depth all the way to your top-level method (the one that started everything at the beginning). If your top-level method throws an Exception, the Java runtime environment will catch it and your program will come to a crashing halt.

Methods which involve input and output are often subject to external constraints which are difficult, if possible, to deal with in a program. For example, when attempting to open a data file, the file may not exist, the disk might be full (when opening for output), or network problems may exist (for drives mapped to a network). This is why all sub-classes of IOException are checked Exceptions; the likelihood of an Exception occurring is so great that the developers of Java decided to force you to check for them. We will discuss I/O exceptions further in chapter 9.

As an example, reconsider our `deansList` method to return a List of all students from the given List who have a given minimum GPA. It called a few methods: `getGradePoints(Student)` and `getCredits(Student)`. Suppose these methods obtain information by reading data from a disk file; it is likely that they would throw some sort of (checked) IOException. If this is the case, we should see it described in the API, and the compiler would see the `throws` keyword in the signature:

```
/**  @param st A Student in our disk file of Students
 *   @return the number of credits earned by st.
 *   @throws IOException
 */
public int getGradePoints (Student st)
       throws IOException
{ //  Code to open a data file and find the number
  //  of grade points for the given Student...
}
```

Because this method specifies `throws IOException` in the signature, the compiler will force us to make a choice: handle the Exception right here in the deansList method or pass the buck to the calling method.

If we choose to handle the Exception here in the deansList method, it could be done as shown here with a try/catch:

```
/**  @param min The minimum GPA needed to be on the dean's list
 *   @return A List of all students on the roster
 *   who qualify for the dean's list.
 *   Uses getGradePoints() and getCredits() to obtain data for
 *   a student.
 */
public List<Student> deansList (List <Student> roster, double min)
{  List <Student> result = new LinkedList <Student> ();
   int credits, gradePoints;
   for (Student st : roster)
       {  try                // handle a possible IOException
            {  credits = getCredits(st);
               gradePoints = getGradePoints(st);
               st.calcGPA (gradePoints, credits);
               if (st.getGPA() >= min)
                    result.add (st);
            }
          catch (IOException ioe)
            {  System.err.println ("Unable to obtain data for " + st); }
       }
}
```

Note that the calls to getCredits and getGradePoints are in the try block, as well as all statements which need the results of those methods. It would make no sense to call calcGPA if we have failed to obtain the information which it needs.

Our other option is to pass the buck; i.e. notify the calling method that we are unable to handle this Exception so the calling method must decide what to do:

```
/**  @param min The minimum GPA needed to be on the dean's list
 *   @return A List of all students on the roster
 *   who qualify for the dean's list.
 *   Uses getGradePoints() and getCredits() to obtain data for
 *   a student.  Ignores students with non-valid data.
 *   @throws IOException if credits and/or grade points cannot
 *   be obtained from the data file.
 */
public List<Student> deansList (List <Student> roster, double min)
       throws IOException       // pass the buck
{  List <Student> result = new LinkedList <Student> ();
   int credits, gradePoints;
   for (Student st : roster)
       {  credits = getCredits(st);
          gradePoints = getGradePoints(st);
          st.calcGPA (gradePoints, credits);
          if (st.getGPA() >= min)
               result.add (st);
       }
}
```

Note that the @throws keyword in the API is for the benefit of people (the programmers who will be writing methods that call this method), and the throws keyword in the method signature is primarily for the compiler.

8.3.5 Defining your own Exception classes

Classes in the Java class library can be extended (sub-classed) by classes that we create. This is certainly true for Exception classes. When we define our own Exception class, we must decide which of the Exception classes in the Java class library is to be the super-class. If we wish our class to be an unchecked

Exception, it should extend RunTimeException, but if we wish our class to be a checked Exception, it should extend Exception (or some sub-class of Exception other than RunTimeException).

When doing so, we will use the existing Exception classes as models of good behavior, and design our classes in a similar way. In particular, we will have a few constructors: a default constructor with no parameters, and a constructor with a String parameter, to be displayed for the programmer of the client method.

Continuing with our Student example, suppose we wish to throw an Exception when a GPA cannot be calculated. We may wish to define our own Exception specifically for this error. If we wish the compiler to force checking for this Exception, we would want it to be a checked Exception, and therefore it could be a sub-class of Exception:

```
/** This Exception should be thrown when a Student’s GPA cannot
 *  be calculated.
 *  Detailed message and the Student can be included, but are
 *  optional.
 */
public class BadGpaException extends Exception
{  private String message;     // error detail
   private Student student;    // Student with the problem

/** Construct a BadGpaException with null as its
 *  error detail message and null as its
 *  offending Student;
 */
   public BadGpaException()
{  }

/** Construct a BadGpaException with the given String as its
 *  error detail message, and the given Student as the
 *  offending Student.
 *  @param s Detaileds info on the error, or null if no info
 *           is available.
 *  @param st Student who caused the problem, or null if
 *           not available.
 */
   public BadGpaException(String msg, Student st)
{  message = msg;
   student = st;
}

/**
 *  @return the details of the error,
 *  which could be a null reference,
 *  if not available.
 */
public String getMessage()
{  return message;  }

/**
 *  @return the offending Student,
 *  which could be a null reference,
 *  if not available.
 */
public Student getStudent()
{  return student;  }

/**
 *  @return this BadGpaException as a String
 */
public String toString()
{  String result = "GPA could not be calculated";
   if (message != null)
      result = result + " because " + message;
   if (student != null)
      result = result + "\nCaused by " + student;
   return result;
}
}
```

We can now add the BadGpaException class to our class diagram; it is shown in Figure 8.5.

In the `BadGpaException` class note that:

- There are two fields in this class:
  - A String containing details on the cause of the Exception
  - A reference to the Student who caused the Exception to be thrown (the offending Student).
- There are two constructors in this class.
  - A constructor with no parameters (default constructor) which is used when the error details and the offending Student are unavailable. This constructor leaves both fields initialized to null references.
  - A constructor with two parameters which is used when either, or both, the error details and the offending Student are available. It initializes at least one of the two fields.
- If a method wishes to throw this Exception, there are a few ways this can be done:

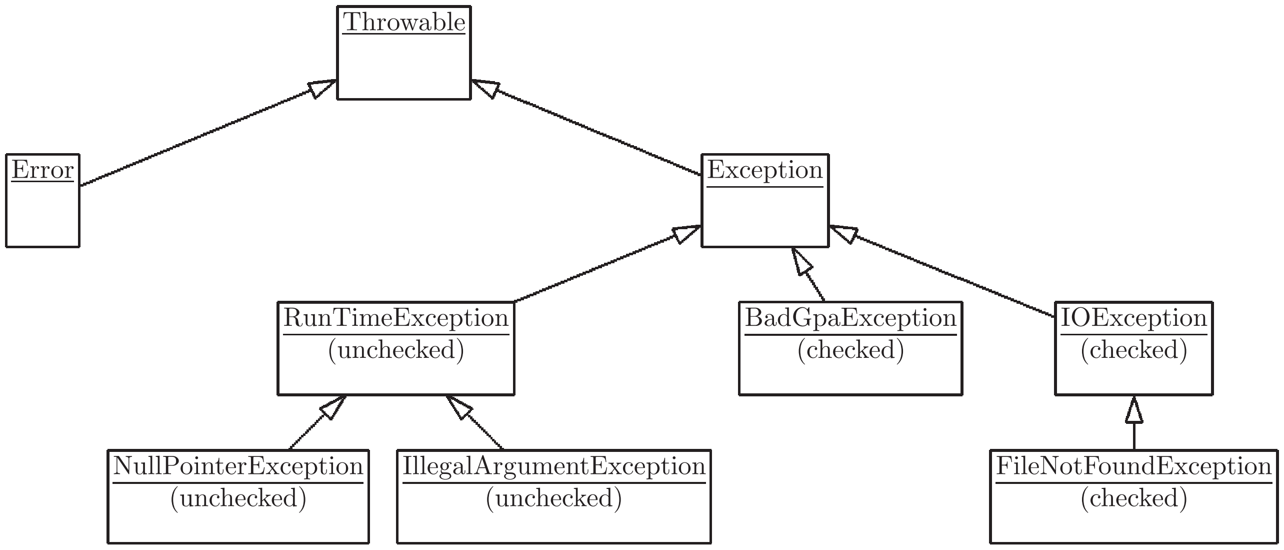


Figure 8.5: Class diagram showing some of the Exception classes in the Java class library with our own class, BadGpaException

- If it has no information on both the error details and the offending Student, it could throw an Exception using the default constructor:
  `throw new BadGpaException ( );`
- If it has information on both the error details and the offending Student, it would call the constructor with two parameters:
  `throw new BadGpaException ("No credits earned", someStudent);`
- If it has information on either the details of the error or the offending Student, but not both, it would call the constructor with two parameters, with `null` as a parameter:

```
// Offending Student not available
throw new BadGpaException ("Student not in database", null );

// Error details not available
throw new BadGpaException (null,someStudent);
```

8.3.6 Exercises

1. List a few java Exceptions not mentioned in this section.
2. Can a NullPointerException be created with a String as a parameter?
   `NullPointerException npe = new NullPointerException ("foo is null");`
   If so, what is the purpose of the parameter?

Hint: See the API for the package `java.lang`.

3. Refer to the `university` project in the repository for this chapter. Which of the following code segments will cause an Exception to be thrown, and if so, what is the class of the Exception?

   (a)
   ```
   Student frosh;
   frosh.calcGPA(12,3); // 12 grade points
   System.out.println (frosh);
   ```
   (b)
   ```
   Student frosh = new Student ("jim", "232-34-3333");
   frosh.calcGPA(12,0); // 12 grade points
   System.out.println (frosh);
   ```

4. In the `Student` class of the `university` project in the repository for this chapter, there are accessor methods for a Student's name and ssn. Make the following modifications to these accessor methods, and include an explanation for the reason the that it is being thrown (don't forget to update the API).

   (a) Modify the `getName()` method so that it will throw an `IllegalStateException` if the name is a null reference.

   (b) Modify the `getSSN()` method so that it will throw an `IllegalStateException` if the ssn is not valid. The format of an ssn should be "999-99-9999", where the "9" represents any numeric digit.

5. Use the `university` project in the repository for this chapter. In the `UniversityInfoSys` class define a method named `buildPassword` with one parameter, a `Student`. The method should return an initial password for the given Student consisting of the first initial of the Student's name and the last two digits of the Student's ssn. If one of the server methods throws an Exception, it should be caught and handled in the `buildPassword` such that:

   - An error message is sent to stderr (System.err.print...)
   - Execution continues; the program does not crash.

```
/** @return An inital password for the given Student,
 *  consisting of first initial of name and last two digits
 *  of ssn.
 *  Execution continues if an IllegalArgumentException is thrown,
 *  with a message sent to stderr.
 */
 public String buildPassword (Student st)
```

8.4 Debuggers

A software tool which is helpful in finding and fixing programming errors is called a *debugger*. The debugger will not help you with compile-time errors; for these you must rely on the error message provided by the compiler. But for complex run-time errors, which may be buried deep in several nested method calls, or nested loops, a good debugger is essential. There may be some disagreement over the value of a debugger versus *code review*. Code review, recommended by many software engineers, is the process of examining the program carefully, either alone or with other programmers, to make sure that every possible problem or error is avoided. We agree that code review is valuable, but there always seem to be bugs which slip through after the most careful code review. For these, one needs to have the necessary skills to use the debugger effectively.

Each Integrated Development Environment (IDE) has its own debugging tool with a user interface. These user interfaces may appear different, and somewhat daunting, at first. Here we attempt to describe some of the features available in most debuggers, without going into the specific details of any one particular debugger. We stress that the debugger is used to find the line(s) of source code which are causing erroneous output, a run-time exception, or an invalid program state. Note that a debugger will NOT help with:

- A debugger will not help you with compile-time errors. The compiler will provide an error message and line number in the source file, either of which might be accurate or helpful.
- A debugger will not help you discover that a flaw in the program exists. For this we use code review and thorough testing.
- A debugger will not automatically show you where the bug is; it is simply a tool which will help you find the bug.
- A debugger will not fix the bug for you. A good understanding of the program logic is necessary to fix the bug yourself.

The features of a typical debugger include:

- Single Step (or Step) – Step through the statements of the program one line at a time. Click a button to allow the next line to be executed. There are usually two ways to do this:
  - Step Over – Execute the current line. If the line contains a method call, run the called method at full speed before returning control to the debugger, and move on to the next line of code.
  - Step Into – If the current line contains a method call, step into that method and return control to the debugger to execute the called method one line at a time.

- Watch variables – Show the value of selected variables as the program executes. Many debuggers show all active variables (local variables, instance variables, class variables) by default. Many debuggers will also show you the call stack – the sequence of method calls which led to the current line.

- Set Breakpoint – Any executable line may be set with a breakpoint. When running a method at full speed, execution pauses when encountering a breakpoint. Control is returned to the debugger so that one can single-step from that point.

- Continue – Continue executing at full speed, pausing at the next breakpoint, if any.

- Conditional commands – Choose one of the options shown above conditionally, depending on the value(s) of variable(s). This feature is available only with the more advanced debuggers.

- Step backwards – If you have gone past the bug and do not wish to start over from the beginning, you can step to the previous line. This feature is available only with the more advanced debuggers (notably in the Eclipse IDE).

We cannot overemphasize the importance of learning to use a debugger skillfully. Once you have experience with one particular debugger, it is relatively easy to learn a different debugger.

8.4.1 Exercises

1. Briefly describe the difference between the meaning of *step into* and *step over* for a typical debugger.

2. True or False: A Debugger is a tool which can be used to patch a defective program, after the programmer has determined which statement is at fault.

3. Consider the following code segment:

```
for (int i=0; i<100; i++)
  for (int j=0; j<10000; j++)
     System.out.println (i*j - i/j);
```

   Enter this code into a method in a new class, and execute it with your debugger. Step through the statements and pause when the value of i is 25 and the value of j is 10.

8.5 Debugging with print statements

One of the oldest and most obvious debugging techniques involves the use of print statements. Insert print statements at various points in the program to make sure that variables contain correct values. This is sometimes easier than using a debugger, but if your print statements are in nested loops, you could get large quantities of output, making it difficult to search for useful information. The following print statement assures the programmer that the value of the variable `credits` contains the correct value, 45:

```
   System.out.println ("credits should be 45, credits = " + credits);
```

8.6 Projects

1. Use the project `university-debug` in the repository. In this project the class `UniversityInfoSys` is responsible for providing information on courses and students enrolled. The `Student` class has been enhanced to store a collection of courses in which the Student has enrolled. You should start out by viewing the API (Documentation View) for each of the classes.

   This project compiles without errors (ignoring the class `Driver2`), but it contains at least 6 run-time errors. Some of these will cause the program to crash, and some will cause incorrect output. Correct all run-time errors (they will be found in the Student and UniversityInfoSys classes only); it may be helpful to use a debugger. When you test your solution using the `main` method in the `Driver1` class, the output should look like the output in the plain text file `university-debug-output1.txt`. (Do not try to compile the class `Driver2`; you could temporarily remove it from the project. It is for the next problem.)

   Hint: Your debugger should be able to distinguish between local variables and fields.

2. Use the same project that you used in the above problem; here you will test with the `Driver2` class.

   In this problem a Student's ssn must be in the format '999-99-9999', i.e. all characters must be numeric (0-9) except for the two dashes, and the dashes must be in positions 3 and 6.

   Define an Exception named `InvalidSSNException`. This Exception will be thrown when a Student is given an invalid ssn. We will handle the exception by attempting to put the ssn in the correct format, if possible. It that is not possible, we will use the given ssn, print an error message, and continue processing. Be sure that your `InvalidSSNException` class has the appropriate javadoc comments for a useful API. It will be necessary to make some changes to the `Student` class.

   The output should match what is shown in the plain text file `university-debug-output2.txt`.

Chapter 9

Console Applications – Input and Output

Most programs need to communicate, in some way, with the world outside. They may need information from some external source, or they may need to communicate results to an external device. In this chapter we examine some common operations which are used to bring data into an executing program (input) and operations used to send data out from an executing program (output).

The terms *input* and *output* are used from the point of view of the primary memory (RAM). The input operation brings data into RAM from some external source such as a disk, a USB port, a keyboard, or a network port. The output operation sends data out from RAM to some external device such as a disk, a USB port, a monitor, a printer, or a network port. Figure 9.1 shows a simple diagram of these hardware components, with arrows showing the direction of data transfer for input and output.

This chapter will discuss these input and output operations for *console* applications only. A console application is one in which the data being input, or put out, for the user is plain text (i.e. ASCII or Unicode characters). In communicating with the user, console applications cannot use graphics of any kind, including graphical user interfaces. Nor do console applications normally utilize images or sound.

9.1 Standard io files

Most systems which accommodate console applications make use of *standard IO* files. These are somewhat abstract concepts that represent the source or destination of data being input or put out, respectively. Standard IO is usually abbreviated `stdio`, and there are three such files which we discuss here:

- `stdin` – The standard input file. Normally this refers to the user's keyboard. When we read from from `stdin` we are obtaining plain text that

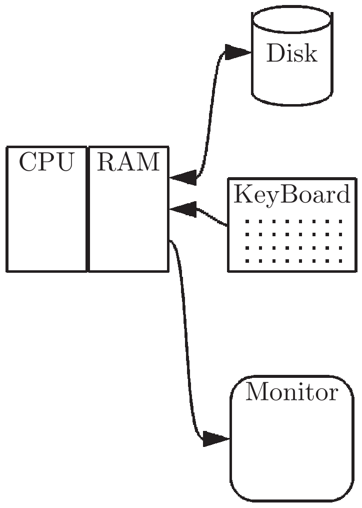


Figure 9.1: Simplified diagram of a computer with peripheral devices

the user types on the keyboard.

- `stdout` – The standard output file. Normally this refers to the user's monitor (or terminal window in an IDE such as BlueJ). When we print to `stdout`, we are sending plain text to the user's monitor.
- `stderr` – The standard error file. Similar to `stdout`, `stderr` is normally used for error messages directed to the user. When running an application from the system command line, text sent to `stdout` will not be distinguished from text sent to `stderr`. Some IDEs, such as BlueJ, will show `stdout` and `stderr` in separate windows, with the `stderr` output in red.

The user can redirect any, or all three, of these standard files to some other source or destination.

9.1.1 Exercises

1. True or false:
   (a) Both `stderr` and `stdout` are initially directed to the user's monitor but can be redirected to some other destination.
   (b) `stdin` is initially directed to the user's keyboard but can be redirected to some other source.
   (c) Console applications may utilize images or sound to improve communication with the user.
2. Is a printer an input device, or does it perform output as well?
3. Is a scanner primarily an input or output device?

9.2 Output to stdout or stderr

In this section we explain how to send plain text to the user, on `stdout` or `stderr`. Normally the output will be sent to the user's monitor, but it can be redirected to some other destination such as a disk or USB port.

9.2.1 Output to stdout

One way to send text to `stdout` we've already seen:

```
System.out.println("some message");.
```

`System` is a class in the package `java.lang` which does not need to be imported. `System` has other interesting features, but we will be using it for output of plain text to `stdout`. When the `println` method is executed, its parameter, which is a String, is sent to `stdout`, and the user is able to see the output. If the parameter of println is not a String, it is automatically converted to String via a call to the toString() method. All primitive types can be converted to String automatically by the run-time system, and all reference types inherit a toString() method from Object if they don't define one. The toString() method inherited from Object is probably not going to do what you want. It will call the hashCode method on the object, and convert the result to hexadecimal (base 16 characters), concatenated with the name of its class. If you see something like `Student@148ae300` on your output screen, you are probably missing a toString() method in your Student class.

9.2.2 Output to stderr

In order to send plain text to `stderr`, we simply use `err` instead of `out` in the call to the println method. This form is intended to be used for error messages, such as:

```
System.err.println ("Number of credits should not be negative");
```

9.2.3 Exercises

1. Which IDE do you use? When executing a program does it show `stdout` and stderr in separate windows?
2. Write a simple java method to write some text to `stdout` and some to `stderr`. Compile and test this method from the system command line. Are the output lines of `stdout` separate from the output lines of `stderr`?

   Hint:

   (a) Open a terminal window (or cmd window).
   (b) Move to the directory in which you wish to work.
   (c) Define a class named `Test` in a text file named `Test.java`. It should have a method:

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

(d) Compile your class by issuing the command:
`javac Test.java`

(e) If there are no compilation errors, run the main method in the class by issuing the command:
`java Test`

3. Only Strings can be printed to `stdout` or `stderr`. Explain how it is possible to print something that is not a String:
`System.out.println (new Student ("jim", "123"));`

9.3 Input from stdin

Input is the process of transferring information from an external source such as disk, keyboard, or USB port into the computer's RAM as a program executes. We can use the standard input file, `stdin`, which is normally directed to the user's keyboard. There are several ways of accomplishing this; one of the easiest and most powerful tools for input is the `Scanner` class which is in the `java.util` package. Do not let the huge API for this class scare you away; it has many powerful features, but we will be using a few of the easier features.

In order to use Scanner it must be imported at the beginning of the class:
`import java.util.Scanner` or `import java.util.*`
When instantiating a Scanner object, you can provide it with the source of the input. In this section we wish to read from `stdin`, i.e. the user's keyboard. Consequently we identify this as `System.in`:

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

We can now apply any of the methods in the `Scanner` class to this object, which we have named `scan`. The two methods from `Scanner` which we introduce now are:

- `public boolean hasNextInt();` – This method will return true only if an int has been entered on the input stream (System.in in our case).
- `public int nextInt();` – This method will return an int, if possible, from the input stream.

As an example, we could implement a method in our `Student` class to get the number of credits from the user's keyboard:

```
/**  Get the number of credits for the given Student
 *   from stdin.
 *   @return The number of credits read from stdin.
 */
public int getCredits (Student st)
{  int result = 0;

   // Prompt user for input
   System.out.println ("Enter the number of credits for " + st);

   if (scan.hasNextInt())
       result = nextInt();
   else
       System.err.println ("Invalid number entered, 0 is assumed");
   return result;
}
```

Note that this method first prompts the user to enter something on the keyboard. If you forget this prompt, the user will not know that the computer is waiting for input; the user and computer will sit there waiting for each other to do something. Users need to be prompted for action, and they need to be told what the program is expecting to be entered.

The call to `hasNextInt()` is recommended, so that if the user enters something which is not a valid int, the program will not crash with an IOException.

9.3.1 Exercises

1. Which method in the Scanner class can be used to read the next valid double precsion number from the input stream?

2. Which method in the Scanner class can be used to determine whether there is a valid double precsion number at the front of the input stream?

3. Experiment to see what happens when a program tries to get a double precision number from the input stream, and the front of the input stream is not a valid double precision number.

4. Write a method to prompt the user to enter three whole numbers at the keyboard. Your method should return the average of the three numbers.

```
/** @return the average of three whole numbers entered
 *  on the user's keyboard */
public double average3 ()
```

9.4 Data Files

Large quantities of data can be stored permanently on external or internal storage devices such as disks and USB flash memory. In this section we will see how to input data from storage, and how to put out data to storage, from an executing program. Before a data file can be accessed, it must be *opened*. Once it has been opened, it is possible to read data from the file, append data to the file, or overwrite existing data with new data to the file.

9.4.1 Opening a data file

Before any input or output can occur, a data file must be opened. This is the point where the program requests a service from the operating system:

- I will tell you whether I will be using it for input or for output.
- If using it for input, can this file be found in the folder where I think it exists?
- If using it for output, can this file be created in a particular folder?
- Do I have permission to use this file?

A file can opened by instantiating an appropriate class from the Java class library; this will typically be an instance of the File class for input, or the FileWriter class for output; the constructor is provided with the name of the file to be opened:

```
new File("myInputFile.txt");
new FileWriter("myOutputFile.txt");
```

More complete examples are given in the following sections.

9.4.2 Input from Data Files

Input is the process of transferring information from an external source, such as a disk or USB flash memory, into RAM for an executing program. We will use the same class that we used for keyboard input: `Scanner`. To open a file for input, we will instantiate a `Scanner` object using an instance of the `File` class. `File` is in the `java.io` package so it will need to be imported:

`import java.io.File;` or `import java.io.*;`

We will also need to import the Scanner class:

`import java.util.Scanner` or `import java.util.*;`

We can now open a file for input by instantiating Scanner:

```
Scanner scan = new Scanner (new File("myFile.txt"));
```

Notice that we are careful to provide `Scanner` with a `File`. If we had omitted the `new File` and simply provided `Scanner` with the file name as a String, we would have been calling a different constructor in the `Scanner` class – Scanners can also read from Strings, and that is not what we are interested in doing here.

As shown above, however, the compiler will not accept this. If you look at the API for the constructor in the `File` class you will see that it *throws* IOException. The API goes on to explain that if the file cannot be found or cannot be opened for some other reason, an `IOException` will be thrown. Recall from chapter **??** (Figure 8.4) that `IOException` is a *checked* exception, which means that the client method (i.e. the calling method) must do one of the following:

- Handle the Exception with a try/catch statement.

- Declare that this method *also* throws IOException (i.e. pass the buck).

If you omit both of the above, the compiler will remind you of your choices:

`Unreported Exception java.io.IOException; must be caught or declared to be thrown`

Once the file has been opened for input, we can use the `Scanner` object to read data just as if we were reading from the user's keyboard. In addition to `hasNextInt()` and `nextInt()`, the `Scanner` class has methods which will read an entire line as a String: `hasNextLine()` and `nextLine()`.

When finished reading data from a file, we should close it. If we forget to close it, the system will close it when the program terminates; however we should close it anyway. This is a good habit to get into; good citizens always close their files when finished:

`scan.close();`

We can now write code which will read all the lines in a text file and print them on `stdout`:

```
import java.util.*;             // Scanner
import java.io.*;               // File
...

String line;
// Open the file roster.txt for input
try {
       Scanner scan = new Scanner (new File ("roster.txt"));

       while (scan.hasNextLine())
       {  line = scan.nextLine();   // Read a line from the file
          System.out.println (line);
       }

       scan.close();                  // Good citizens close files
     }
catch (IOException ioe)
    {  System.err.println ("File roster.text not found"); }
```

Notice that we include in the try block not only the call to the File constructor, but everything which depends on its successful termination as well.

9.4.3 Output to data files

There are many ways of writing output to a data file. We will discuss one of the simpler techniques. Though it is possible to write binary data to a file, we will be writing plain text. We will need the `FileWriter` class which is in the `java.io` package:

`import java.io.FileWriter;` or `import java.io.*;`

Though it is possible to append data to an existing file, we will simply write to a file, assuming that if it does not exist, it will be created, and if it does exist it will be overwritten. To open the file for output, instantiate the `FileWriter` class, by giving it the name of the file as a String:

```
FileWriter writer = new FileWriter("myFile.txt");
```

Like the `File` class, the `FileWriter` constructor also throws IOException, so we enclose its instantiation in a try block.

The example above presumes that the file to be opened is in the same directory (i.e. the same folder) as the Java source file. If it is located somewhere else, a full path description of its location is needed:

```
FileWriter writer = new FileWriter("c:myStuff/myFile.txt");
```

After opening the file, we can write Strings to it using the `write (String)` method:

```
writer.write("any string...");
```

Each time this method is executed a String will be written to the file. Note that if you want the Strings stored on separate lines, you must include a newline character, `'\n'` at the end:

```
writer.write("any string...\n");
```

Finally, when finished writing data to the file, it should be closed. Most systems will automatically close a file when the program terminates, but you should close it anyway. This is a good habit to get into; all good citizens close their files when finished using them:

```
writer.close (); // good citizens close files
```

The following code segment will open a file named `roster.txt` for output, and write data for two students to that file:

```
import java.io.*;
...
try {
       FileWriter writer = new FileWriter ("roster.txt");
       writer.write ("joe\n");            // name
       writer.write ("234-54-9498\n");    // ssn
       writer.write ("3.5\n");            // GPA

       writer.write ("jim\n");            // name
       writer.write ("432-45-8949\n");    // ssn
       writer.write ("0.0\n");            // GPA

       writer.close();                    // good citizens close files
    }
catch (IOException ioe)
    {  System.err.println ("File not found:  roster.txt"); }
```

9.4.4 Exercises

1. Under what circumstances might an attempt to open a file result in the throwing of an Exception?

2. Define a method named `createFile` with one parameter which will accept input from stdin and store the lines in a file with the given name.

   ```
   /** Open the given file for output.  Obtain lines of text from stdin,
    *  and write them to the given file.
    *  @param filename Name (and path) of the text file to be created.
    */
       public void createFile (String filename)
   ```

   When testing your solution, use ctrl-d (Unix) or ctrl-z (Windows) to terminate the input.

3. Define a method named `copyFile` with two parameters which will copy all the lines from a text file into a new text file.

   ```
   /** Open source file for input and open target file for output.
    *  Copy all lines of source file to target file and close.
    *  Pre: Source file is a text file.
    *  @param sourceFilename Name (and path) of the text file to be copied.
    *  @param targetFilename Name (and path) of the text file to be created.
    */
       public void copyFile (String sourceFilename, String targetFilename)
   ```

   Test your program by using it to copy the source file to a new file.

4.

9.5 Running an Application from the Command Line

As we develop software in Java, we generally use an Interctive Development Environment (IDE), such as BlueJ, Eclipse, or NetBeans. The IDE provides a convenient way to edit, compile, test, debug, etc. Many IDEs will provide information on a project's classes as well as a nice-looking API. Some IDEs, such as BlueJ, allow the developer to execute an individual class or instance method when testing a particular class (BlueJ also allows direct execution of a single Java statement with a CodePad feature).

9.5.1 Compile and Test from the Command Line

However, a Java application can also be invoked from the Unix or DOS command line. Follow these steps to create your source file(s), compile, and test your application:

1. There should be a source file (.java) for each class in your project. The name of the file should be the same as the name of its class. These files will normally all be in the same directory (folder), and they should be plain text files, created and edited with a line editor such as vi, emacs, edlin, or notepad.
2. One of the classes in your project, the *main* class, should have a `main` method which starts the application:

   ```
   public static void main (String [] args) { ... }
   ```

   This is the method which will be called when execution begins, and is explained in the next section.
3. When ready to compile these source files, they can be compiled individually, or all at the same time:
   - `javac ClassName.java`
   - `javac *.java`
4. If there are error messages from the compiler, use your editor to correct the errors, save, and recompile.
5. If there are no error messages from the compiler, you will notice that there is a .class file in the project's directory for each class in the project which has been compiled. These files contain *byte code* which can be directly executed by the Java RunTime Environment. To execute, you must invoke the Java runtime environment from the command line:

   ```
   java MainClassName parm1 parm2 parm3 ...
   ```

   The parameters, parm1, parm2, parm3, ... are not required, but they should correspond to the array of Strings (args) in the declaration of the main method. Note that there is no .java suffix (or any suffix) on the class name.

9.5.2 public static void main (String [ ] args)

Here we explain why the main method must be declared as shown in the previous section. We dissect this declaration and explain each part below:

- `public` – Since this method will be invoked from outside the class, access must be public.

- `static` – The main method must be a class method; it cannot be an instance method because there are as yet no instances of that class. The runtime environment is assuming it will be a class method and will invoke it as:

  ```
  MainClassName.main (...)
  ```

- `void` – The runtime environment does not make use of a returned value, and requires that the main method be a `void` method.

- `main` – Since the runtime environment is calling a method named `main`, that is what the name of your method must be.

- `String` – The runtime environment calls your main method with one actual parameter; thus your main method must have one formal parameter.

- `[ ]` – The parameter is an array of Strings.

- `args` – The parameter name can be any valid Java name, but most people use `args` which is short for *arguments*. The term ‘argument’ is synonymous with ‘actual parameter’ and is carried over from older languages such as C++ which have functions.

The command line arguments correspond to the items in the array of Strings which is the main method’s formal parameter. If the runtime environment is invoked as shown below:

```
   java MyClass sam joe bill
```

then when the main method starts up, the value of its parameter will be:

```
args.length = 3
args[0] = "sam"
args[1] = "joe"
args[2] = "bill"
```

If you supply too many arguments on the command line, your program simply ignores the extra arguments. However, if you enter too few arguments on the command line, you could get an `IndexOutOfBoundsException`.

9.5.3 Exercises

1. Choose one of the methods you defined in the exercises of the prevous section, and modify its class so that it can be executed from the command line of a terminal window.

2. Define a command named Alphabetic which will determine whether its three arguments are in alphbetic order. It should print either “in order” or “not in order”. For example:

```
> java Alphabetic alpha gamma beta
not in order
> java Alphabetic alpha beta gamma
in order
```

3.

4.

9.6 Projects

1. A classic data processing task involves the *merging* of two or more ordered data files. The output file contains all the data of the input files and is also ordered.

   In this project we wish to merge two text files, each of which contains a single word on each line. The words are in alphabetic ordeer. We wish to create a file containing all the words in both files (duplicates are ok) in alphabetic order.

   Use the project `fileMerger` from the code repository. Define a class named `Merger` in which you will define a method with three parameters named `merge`. The parameters are the names of the two input files and the name of the output file to be created.

   ```
   /**
   * Pre: file1 and file2 are text files in alphabetic order.
   * @param result is the name of the file produced by merging
   *  file1 with file2.
   * The result is also in alphabetic order.
   */
   public void merge (String file1, String file2, String result)
   ```

   Test your solution by using the data files provided in the code repository, `file1.txt` and `file2.txt`.

2. Java (and many other programming languages) have two kinds of comments:

   - Single-line comments: Begin with // and extend to the end of the line
   - Multi-line comments: Begin with /* and end with */

   In a new project create a class named `Comments`. In this class define two methods:

   - A method named `countSingleLineComments` which will count all the comments beginning with // in a java source file.

```
/** @return number of single-line comments in the
*  given java source file
*/
public int countSingleLineComments (String filename)
```

- A method named `countMultiLineComments` which will count all the comments beginning with /* and ending with */ in a java source file.

  ```
  /** @return number of multi-line comments in the
   *  given java source file
   */
  public int countMultiLineComments (String filename)
  ```

Be careful:

- The following counts as two multi-line comments:
  `/* hi */ /* there */`
- The following counts as one single-line comment:
  `// hi there /* today */`
- The following counts as one multi-line comment:

  ```
  /*  This is a // multiline
      comment
  */
  ```
- The following counts as one multi-line comment:

  ```
  /*  This is a /* multiline
      comment
  ****/
  ```

Hint: Define one method which returns both values as a `List<Integer>` of size 2. Your method will behave like a *state machine* as it scans the characters of the source file:

- It will initially be in a *default* state (not inside a comment).
- When it sees a single slash, it will be in the *slash* state.
- When it sees // it will be in *single-line* state, and will remain in that state until it reaches the end of the line, at which point it will return to the default state.
- When it sees /* it will be in *multi-line* state, and will remain in that state until it reaches */ at which point it will return to the default state.

This state machine is shown in Figure 9.2. The circles represent states, and the labels on the arcs represent input characters (n represents a newline character, and A represents any character other than /, *, or newline). As each input symbol is scanned, the state of the machine changes according to the arrow. The machine starts out in the default state (DEF).

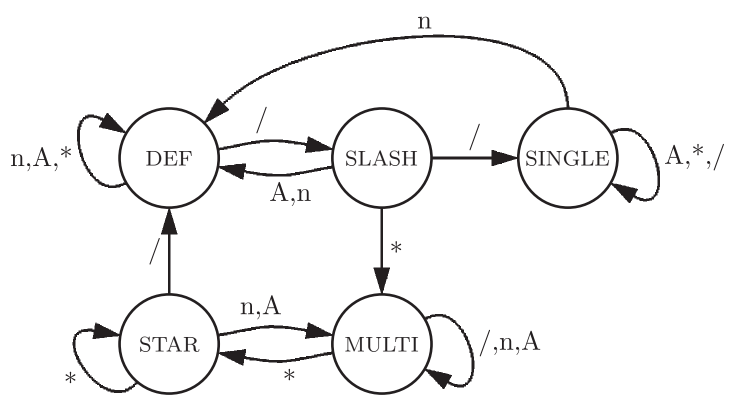


Figure 9.2: State machine to process java comments. `n` represents a newline character. `A` represents any input character except newline, *, and /. Start at the default state (DEF).

Chapter 10

Graphical User Interfaces

In the early days of computing (1945-1965) large mainframe computers executed programs in *batch* mode. Programs (and data) were submitted via punched cards or paper tape. At some later time, paper output would be available to the user. There was no interaction with the user as the program executed.

This was followed (1965-1980) by time-sharing systems which allowed users to provide input, and read output, via electric typing machines (often teletype machines, or IBM Selectric typewriters). Users were now able to interact with programs at execution time. Users could manage files and launch programs by typing commands to the operating system at the *console*. Such *command line* user interfaces are still in use today (Unix, DOS, e.g.).

In the mid 1970's researchers at the XEROX Palo Alto Research Center (PARC) experimented with a new user interface in which a pointing device (a mouse) was used to select icons on a monitor. XEROX did not pursue this effort, but the Apple corporation had recently developed micro-computers to compete with the IBM PC. Apple took PARC's idea, and produced the Macintosh computer with a *graphical user interface* (GUI) using a desk-top metaphor:

- Images of folders represented directories
- Copying of files was done by dragging items with the mouse
- Programs were launched by clicking on filenames with the mouse
- Files were deleted by dragging them to a trash can

Thus the GUI revolution was born; this was clearly a better way for the average user to communicate with an operating system. This user interface is sufficiently intuitive that novice users spent little, if any, time reading manuals. Apple's leading competitor in software, Microsoft, was compelled to follow suit or be left in the dust; a GUI for DOS, called Windows, was the result.

Today, though we still have command-line interfaces for operating systems, console applications are rare. Any software which expects user interaction will have a graphical user interface. In this chapter we expose some of the Java classes which can be used to provide an application with a GUI.

10.1 Packages java.awt and javax.swing

Related classes can be grouped together in a *package*. There are primarily two such packages which can be used to build a graphical user interface:

- `java.awt` – This is the original package used for building a GUI. Some of its classes have been replaced by a better version in javax.swing.
- `javax.swing` – This package includes newer versions of some of the classes in java.awt, and other classes not found in java.awt.

Since many of the classes in java.awt are still considered usable, and some have been replaced by newer versions in javax.swing, we will need to import from both of these packages. The easiest way to do this is:

```
import java.awt.*;      // all the old classes
import javax.swing.*;   // all the newer classes
```

We do this at the risk of introducing name conflicts with classes imported from java.util or other sources (more on that later). A quick perusal of the java.awt package API shows that we have classes for some common GUI components, such as `Button`, `TextField`, `CheckBox`, `Frame`, and `Color`. These components will allow the user to communicate with an application; think of a Frame as being like a window frame, in which our GUI components will reside.

Looking at the javax.swing package, we see similar classes: `JButton`, `JTextField`, `JCheckBox`, and `JFrame`. However, there is no `JColor` class. The designers of javax.swing felt that the Button, TextField, and Frame classes needed to be rebuilt from scratch, but there was no need to make changes to the Color class, as depicted in Figure 10.1. In general, classes in javax.swing which replace older classes from java.awt begin with a J. Conceivably, we could build a GUI using java.awt only; however, we wish to be more up-to-date, so we will be using swing classes whenever possible.

10.1.1 Exercises

1. Which of the following classes from java.awt have been updated in javax.swing?

   BorderLayout, Menu, Label, Insets, Applet

2. (a) Why did Apple file a lawsuit against MicroSoft and Hewlett-Packard in 1988?

   (b) Which company filed a lawsuit against Apple at the same time and for the same reason?

   (c) What were the outcomes of those lawsuits?

Class in awt	Updated version in swing
Button	JButton
TextField	JTextField
Frame	JFrame
CheckBox	JCheckBox
Color	——
——	ImageIcon

Figure 10.1: Classes in package java.awt and updated versions, if present, in package javax.swing

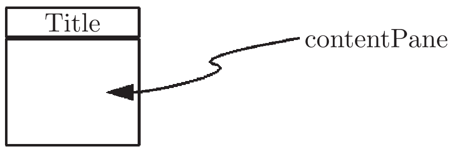


Figure 10.2: A simple frame with a title and contentPane

10.2 Starting out: Frame and ContentPane

A GUI will generally consist of at least one *Frame*. A Frame is the basic structure from which a window may be constructed. Since there is an updated version, `JFrame`, in javax.swing, we will use the updated version. Looking at the constructor for JFrame, we see that it can have a String as its parameter; this is the title of the JFrame. A frame also has a *contentPane*. This is the part of the frame which can store the components (buttons, textfields, etc) of the frame. The contentPane is a *Container*, which is simply a general class which contains zero or more components. A simple frame, with title and contentPane, are shown in Figure 10.2. The appearance of the frame can vary, depending on the host platform (MacOS vs Windows vs Android, etc.).

To expose the various elements of a GUI we will use our Student class and develop a simplified information system for a typical university. All interaction with the user of this informaion system will take place through the GUI. Below we show the inital structure of the GUI:

```
public class UniversityGUI
{
   private JFrame frame;

   public UniversityGUI()
   {
     frame = new JFrame ("State U");   // title
     makeFrame();
   }
   // initialize the frame
   private void makeFrame()
   {
      frame.setVisible(true);         // default is false
    }
}
```

At this point our class has one field, `frame`. We instantiate it in the constructor, then call `makeFrame()` to initialize the frame (this is done in a private helper method rather than in the constructor itself because we will be adding more to it later). Note that frames are initially invisible, and we must set the visibility to `true` if we wish the user to see the frame.

If you try this yourself, you will see a small frame, perhaps in the upper left corner of your screen; it may not display the full title. To remedy this we can set the size of the frame with the `setSize(int width, int height)` method, for example:

```
   frame.setSize(200,100);
```

This will set the width of the frame to 200 pixels and the height of the frame to 100 pixels. A *pixel* is a picture element; it is the fundamental (atomic, or indivisibile) unit of a graphics display. To understand the meaning of 'pixel' examine a photograph in your newspaper very closely. You will see that it is composed of many small dots; each dot can be considered a pixel.

We now wish to access the frame's contentPane, and add components to the contentPane. An example of a component would be a *label* which merely displays some text (or picture) on the frame. In order to work with the contentPane we will use an instance variable, `contentPane`, a Container which can be obtained from the frame:

```
   contentPane = frame.getContentPane();
```

In our `makeFrame()` method we can now add one or more components to the contentPane:

```
   contentPane.add(new JLabel ("I am a label"));
```

Our GUI class now looks like this:

```
public class UniversityGUI
{
   private JFrame frame;
   private Container contentPane;

   public UniversityGUI()
   {
     frame = new JFrame ("State U");            // title
     contentPane = frame.getContentPane();
     makeFrame();
   }

   // initialize the frame
   private void makeFrame()
   {
      contentPane.add (new JLabel ("I am a label"));
      frame.setSize(200,100);
      frame.setVisible(true);                // default is false
    }
}
```

10.2.1 Exercises

1. Point out the syntax errors, or possible run-time errors, if any, in the following statements:
   (a) `Frame frame = new JFrame("State U");`
   (b) `JFrame frame = new JFrame();`
   (c)
   ```
   JFrame frame = new JFrame ("Sate U");
   Container contentPane;
   contentPane.add (new JLabel("label"));
   contentPane = frame.getContentPane();
   ```
2. (a) In a new project define a class named `GUI` which creates a GUI for which the title of the frame is ”Exercise”. Add a textfield to this frame. The initial width of the frame should be 100 pixels and the initial height of the frame should be 200 pixels. Instantiate the GUI to see that it displays properly.
   (b) Experiment to see what happens if you add a button, in addition to the textfield.

10.3 Adding components to a container

JFrame!adding components Component, in a JFrame

10.3.1 Designing the GUI

We now wish to give our information system some functionality; we wish to admit students to our university. The user should have the capability of:

- Admitting students to the university
- Displaying all students currently admitted
- Searching (by name or by ssn) for a particular student

We propose using a GUI which looks like the one in Figure 10.3. The frame for this GUI has the following features:

- The title ’State U’

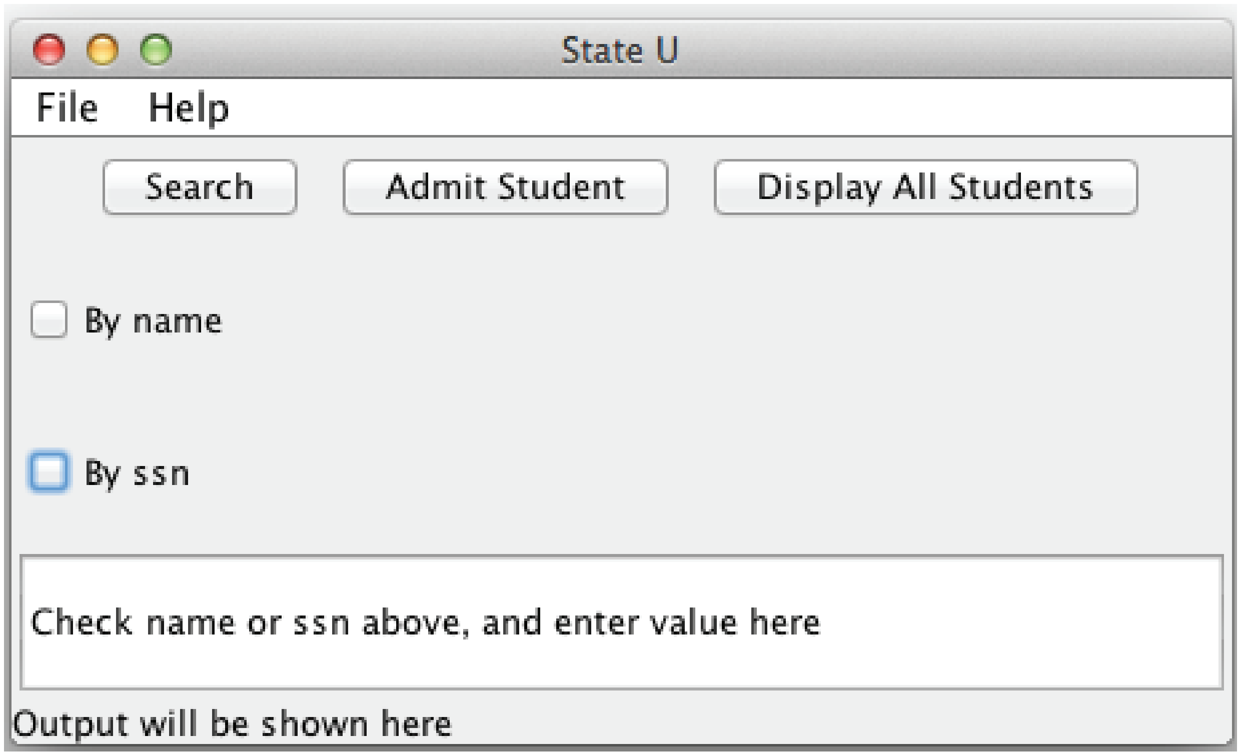

State U
File
Help
Search
Admit Student
Display All Students
By name
By ssn
Check name or ssn above, and enter value here
Output will be shown here

Figure 10.3: Proposed GUI for the university information system

- Three buttons:
  - Search – search for a particular student
  - Admit Student – Allow the user to enter the new student's name and ssn
  - Display All Students
- Check boxes to search by name, or by ssn
- A text field to allow the user to enter a search string (name or ssn)
- A label showing where the output of a search or a display will be shown.

10.3.2 Adding components

Since the contentPane is a Container, we can add components to it. We will now remove the label from the contentPane, and add a few buttons, as shown at the top of Figure 10.3. We can create a button, with text often called a 'caption', showing its purpose:

```
   JButton someButton = new JButton("Caption");
```
We then add the button to the contentPane:
```
   contentPane.add(someButton);
```
For our university GUI, the `makeFrame()` method would now be:

```
   // initialize the frame
   private void makeFrame()
   {
      JButton searchButton = new JButton("Search");
      contentPane.add (searchButton);
      JButton admitButton = new JButton("Admit Student");
      contentPane.add (admitButton);
      JButton displayButton = new JButton("Display All Students");
      contentPane.add (displayButton);

      frame.setSize(200,100);
      frame.setVisible(true);                // default is false
    }
}
```

When you instantiate UniversityGUI, you will see the frame with the correct title, but unfortunately it will show only one button – `Display`. To understand what is happening, we will need to learn about layout managers.

10.3.3 Exercises

1. Give some examples of other kinds of components, in addition to JButton and JLabel.

2. Is a Container an example of a component? (i.e. is Container a subclass of Component?)

10.4 Layout managers

We have seen that components can be added to containers to produce a GUI. A contentPane is an example of a container, and a button is an example of a component. When several components are added to a container, those components must be visually positioned within the container in some way. Hopefully, they will be positioned in such a way that the user will find it easy to understand the meaning and purpose of each component; it would be easy to confuse the user if the components were situated at random positions within the container. It is possible to specify an exact position within the container for each component; but suppose we later add or delete components? What would happen when the user resizes the container? Our GUI would have to recalculate the position of each component to maintain a good appearance for the container.

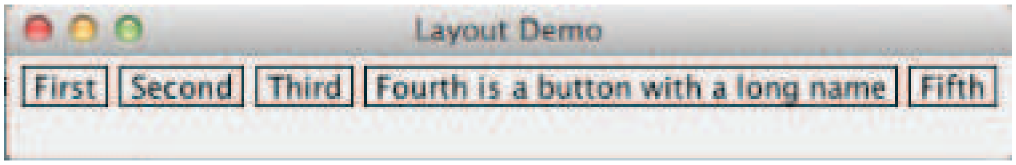


Figure 10.4: Frame with five buttons, using flow layout

There is a better solution to the problem of positioning components in a container: the *layout manager*. `LayoutManager` is an interface in java.awt, and every container has a layout manager. The layout manager will position components in the container in a fairly 'intelligent' way. The layout manager will also reposition, or resize, the components at appropriate times (e.g. the container's size changes). There are several classes which implement the `LayoutManager` interface; each has its own algorithm for positioning components. If you don't like the look of your container, you can use a different layout manager which may give it a better appearance. In this section we will examine some of the more commonly used layout managers.

10.4.1 Flow Layout

The easiest layout manager to use is called *Flow Layout*. A `FlowLayout` is the default layout manager for many kinds of containers, including `JPanel`. When you create a `JPanel`, its layout manager will be `FlowLayout`:

```
    JPanel buttonPanel = new JPanel(); // layout mgr is flow layout
```

To see how the components are positioned by a FlowLayout, we have developed a simple GUI with five buttons. Each button has a caption, but the fourth button has a much longer caption, which makes the default size of that button significantly larger than the other buttons. The GUI initially appears as shown in Figure 10.4. Note that the buttons are positioned in the order in which they were added to the container (i.e. the contentPane). When the window is resized, the buttons are automatically shifted so as to have a 'nice' appearance, as shown in Figure 10.5. With flow layout, the components are arranged horizontally as long as they fit in the container; but if they do not fit, they will be moved vertically to available space. The components seem to flow to available space in the container, hence the name `FlowLayout`.

10.4.2 Grid Layout

A *Grid Layout* will position the components in a rectangular array, or grid, with rows and columns (think of a spreadsheet, or a checkerboard). When components are added, the columns of the first row are filled with components before moving to the second row; i.e. the container is filled in *row-major* order, not in *column-major* order. When instantiating a GridLayout, you can specify the number of rows and columns it will have. Then you can set the layout

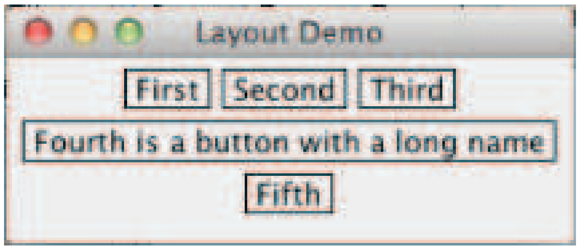


Figure 10.5: Frame with five buttons, using flow layout, after resizing the frame

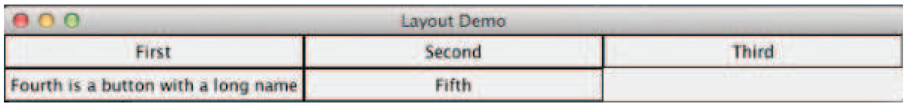


Figure 10.6: Frame with five buttons, using grid layout, 2 rows and 3 columns

manager when instantiating the container:

```
    // 2 rows and 3 columns
    LayoutManager gridMgr = new GridLayout(2,3);
    JPanel inputPanel = new JPanel(gridMgr);
```

The first parameter in the constructor for `GridLayout` is the number of rows, and the second parameter is the number of columns in each row. An example of a frame with grid layout is shown in Figure 10.6. This frame has five buttons (same as the example for flow layout).

If you wish the components to be arranged in a horizontal grid with one row and a variable number of columns, specify 0 as the number of columns:

```
    // One row
    LayoutManager gridMgr = new GridLayout(1,0);
```

In this case the number of columns will increase as components are added to the container. Alternatively, to arrange the components in a vertical column, use 0 for the number of rows:

```
    // One column
    LayoutManager gridMgr = new GridLayout(0,1);
```

In general, if the number of rows (columns) is 0, the grid will have as many rows (columns) as are needed to accommodate the components which have been added.

As you experiment with Grid Layout, you will notice a few things:

- The components are arranged in a rectangular grid
- If the number of components added to the grid is less than the product of the number of rows and the number of columns, unused grid positions are empty.
- As you resize the container, the components also resize to fill their respective grid positions (this may be undesirable, and we will address this issue

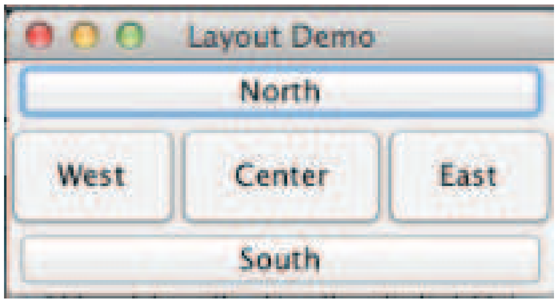


Figure 10.7: Frame with five buttons, using border layout

below)

- There can be at most one component in each position of the grid. If you add a second component to a position, it replaces the first component.

10.4.3 Border Layout

Border Layout is a very commonly used layout manager. It allows you to arrange the container into five regions. A single component may be placed in any of those regions. The five regions are: NORTH, SOUTH, EAST, WEST, and CENTER, as shown in Figure 10.7.

To create a container with a BorderLayout manager:

```
LayoutManager borderMgr = new BorderLayout();
JPanel outPanel = new JPanel(borderMgr);
```

Then to add a component to a particular region, use a class constant from the `BorderLayout` class to specify the region:

```
outPanel.add(new JLabel("hi"), BorderLayout.NORTH);
outPanel.add(new JButton("there"), BorderLayout.CENTER);
```

When using a Border Layout:

- There can be only one component in each region (as with GridLayout). If you add a second componenet to a region, it will replace the first component.

- Each component will expand to fill the entire region as the container is resized (as with Grid Layout).

- If no component is added to a particular region, that region will not be shown at all. The other regions will expand to fill up the entire container.

- The default region is CENTER. If you do not specify a region, the component will be placed in the center region.

- BorderLayout is the default layout manager for the contentPane. This explains why our UniversityGUI in the previous section did not show both buttons in the contentPane. The layout manager was BorderLayout, by default, and both buttons were added to the center region. The displayButton replaced the admitButton.

10.4.4 Nested containers and summary of layout managers

The layout manager for a particular container can be changed at execution time, using the `setLayout(LayoutManager)` method. For example:

```
contentPane.setLayout(new GridLayout(4,3));
```

changes the layout manager for the contentPane to GridLayout.

We have been using the words 'container' and 'component' in a somewhat general sense without giving precise definitions of these words. `Container` and `Component` are both classes in java.awt. Moreover, `Container` is a subclass of `Component`. This means that every `Container` *is-a* `Component`. This provides for the nesting of containers. You can put components into a container, but since every component *is-a* container, you can put a container into a container. This recursive notion is very common in computer science (think of the folders on your computer, which may contain other folders, which in turn may contain other folders, ...). When we nest containers inside other containers, each of those containers may have its own layout manager; we will make use of this property in our UniversityGUI class.

We mentioned that only one component may be placed into a position when using GridLayout, and only one component may be placed into a region when using BorderLayout. If you wish to put more than one component into a position or region, consider adding a container to that position or region. Then several components can be added to the nested container, as shown in Figure 10.8.

This diagram shows a contentPane with BorderLayout. Its five regions contain:

- NORTH – A container with Flow layout
- SOUTH – A container with Border layout. Its five regions contain:
  - NORTH – Nothing
  - SOUTH – Nothing
  - EAST – A component which is not a container, perhaps a button or label, not shown here for lack of space
  - WEST – A component which is not a container, perhaps a button or label, not shown here for lack of space
  - CENTER – A component which is not a container, perhaps a button or label, not shown here for lack of space
- EAST – Nothing

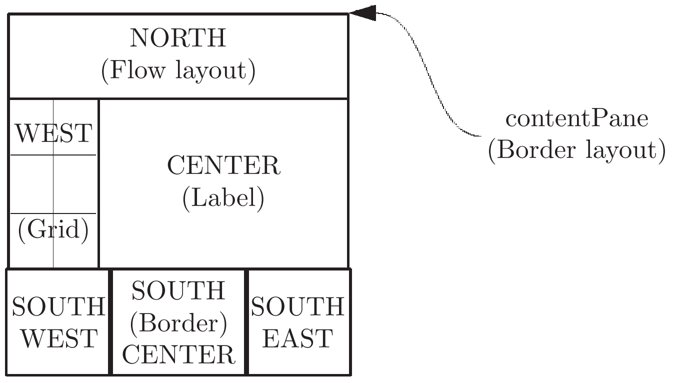


Figure 10.8: A contentPane with BorderLayout, in which containers have been placed in the north, west, and south regions – the container in the south region also uses BorderLayout

- WEST – A container with Grid layout, 3 rows and 2 columns. Its six positions could be components which are not containers, perhaps buttons or labels, not shown here for lack of space
- CENTER – A label

In summary, layout managers may not always perform in an ideal way, but most programmers feel the advantages of using layout managers far outweigh the disadvantages. There are other layout managers in java.awt which we have not mentioned; if your GUI doesn't look just right, you are probably using the wrong layout manager.

10.4.5 University Information System - version 1

Version 1 of our University Information System is in the project university-v1 in the repositry for this chapter. At this point there is only one class in the project: `UniversityGUI` which will define and place all the components in the graphical user interface for our information system.

The frame is instantiated in the constructor, with the title "State U". The constructor also obtains a reference to the contentPane and calls a helper method `makeFrame()` to do all the work involved in placing components into the contentPane.

In the `makeFrame()` method we note that the default layout manager for the contentPane is BorderLayout. We make use of only three regions in the BorderLayout:

- North - contains a JPanel called `buttonPanel` with FlowLayout (default) into which we place three buttons:
  - searchButton, to search for a Student

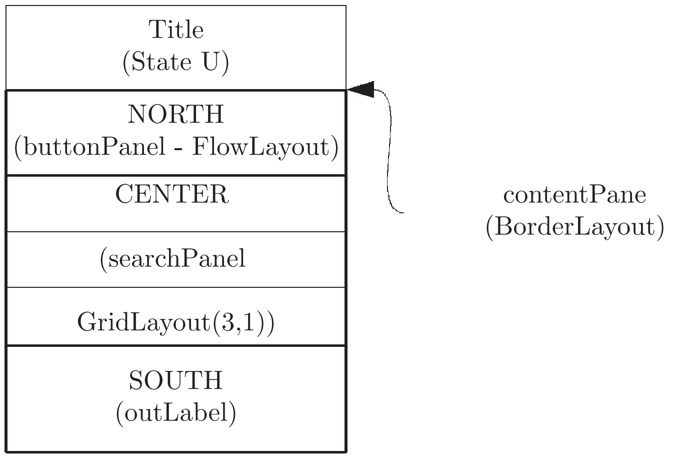


Figure 10.9: Diagram of the GUI for the University Information System, version 1

  - admitButton, to admit a new Student to the University
  - displayButton, to display all students in the University

- Center - contains a JPanel called `searchPanel` which contains check boxes to select a search by name or by ssn. It also contains a text field in which the user may enter a name or ssn for search purposes.
- South - A label which will display output: either search results or a display of all students.

Figure 10.9 shows a diagram of the layout of our graphical user interface at this point.

Students often ask about the sequence in which components are added to the contentPane, and the sequence in which we specify attributes (Should the frame be made visible before or after adding all the components?) In most cases it really doesn't matter; we are simply building a structure and filling in references. As an example, we show a hypothetical object diagram for our frame in Figure 10.10. We call it a hypothetical diagram because we are just guessing at the names of the private fields in many of these classes (as we did with Sets, Maps, etc). To fit this diagram on the page we are not showing some of the referenced objects (and we are probably omitting many fields as well) but Figure 10.10 can be helpful in understanding the internal structure of a GUI.

10.4.6 Exercises

1. Give the name of the LayoutManager with the following properties:
   (a) Components are repositioned vertically and horizontally as the container is resized.

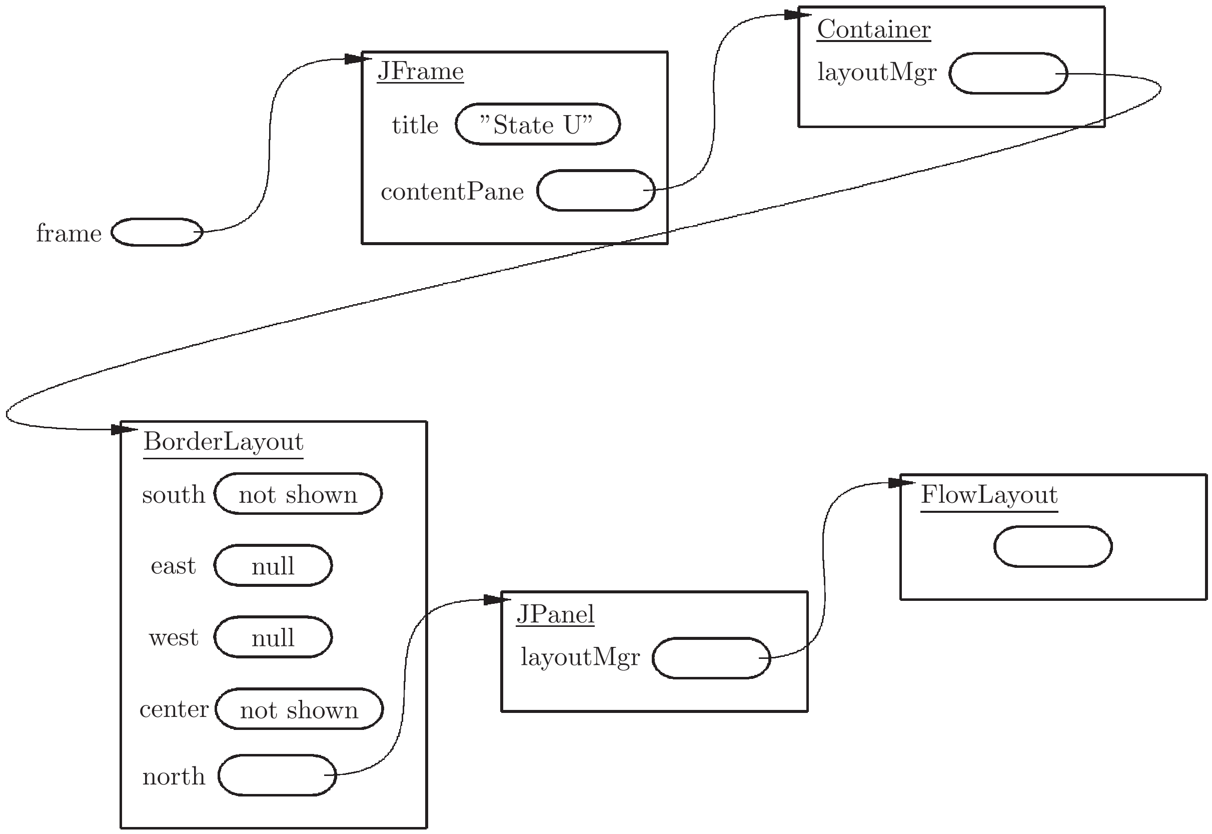


Figure 10.10: A hypothetical object diagram showing the value of the variable `frame` after the `makeFrame()` method has terminated

   (b) Is capable of storing at most five components.

   (c) Will arrange components into rows and columns.

   (d) Has regions named North, South, East, West, and Center.

2. Identify the apparent error, if any, in each of the following:

   (a)
   ```
   JPanel myPanel = new JPanel();
   myPanel.setLayout (new BorderLayout());
   myPanel.add (new JButton ("Click"), BorderLayout.NORTH);
   myPanel.add (new Label ("Hi"), BorderLayout.SOUTH);
   myPanel.add (new Label ("There"), BorderLayout.NORTH);
   ```

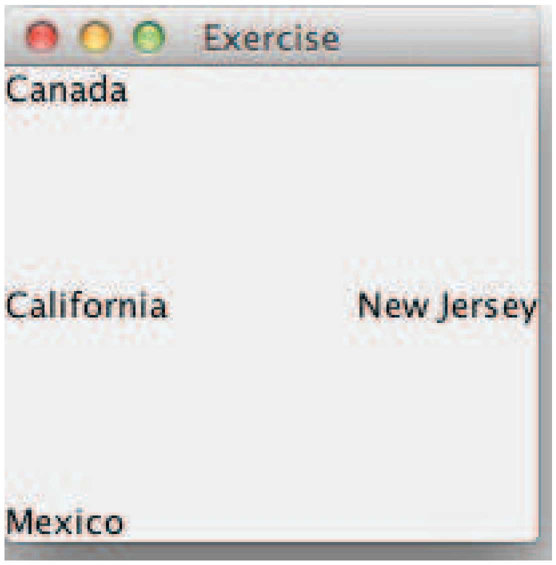
   (b)
   ```
   JPanel myPanel = new JPanel();
   myPanel.setLayout (new GridLayout(3,2));
   myPanel.add (new JButton ("Click"), BorderLayout.NORTH);
   ```

   (c)
   ```
   JPanel myPanel = new JPanel();
   myPanel.setLayout (new GridLayout(0,2));
   myPanel.add (new JButton ("Click"));
   ```

3. How many rows and columns of components will there be in `myPanel` after the code shown below has executed?

   ```
   JPanel myPanel = new JPanel();
   myPanel.setLayout (new GridLayout(0,2));
   myPanel.add (new JButton ("Click"));
   myPanel.add (new JButton ("Here"));
   myPanel.add (new JButton ("Now"));
   ```

4. Define a class which will produce a GUI as shown below when instantiated.



10.5 Actions and Listeners

We now have a GUI for our University Information System. The only problem is that it does not do anything. If you click on a button, nothing happens. We need to use Actions and Listeners; thus we will introduce *event-driven* programming. Up to this point, all appications that we hve developed started up from a `main` method, or from an IDE, and ran to completion (perhaps pausing for input from stdin). With GUIs the sequence of events is much different. Once the GUI has been initialized, it waits for a user action. Examples of actions are:

- The user clicks on a button
- The user selects an item from a menu
- The user moves the mouse
- The user types on the keyboard
- The user provides some other form of interaction with the computer

When any of these actions occur, our application can be programmed to handle them in an appropriate way. Alternatvely, we may wish our application to ignore certain actions (such as a mouse movement). Run time computations occur as a result of a particular action or event, thus the phrase 'event-driven' is used to describe this kind of program.

In the package `java.awt.event` there are several kinds of *listeners*. Listeners are objects which detect a particular event, and are then capable of handling the event appropriately. Each listener is an interface in the package `java.awt.event`; the listener must be implemented in order to handle actions. This means we must import needed classes from this package:

```
import java.awt.event.*;
```

Note that `import java.awt.*;` will not give us the classes from `java.awt.event` because the * matches class names only , not package names, in a package, which is essentially just a folder, or directory. (The . in java.awt.event is like a slash - forward or backward - in a unix or Windows directory path)

Examples of listeners from java.awt.event are:

- `ActionListener` - listen for an action such as a button click or a menu selection
- `MouseListener` - listen for a mouse movement, mouse button down, mouse button up, etc.
- `KeyListener` - listen for a keyboard strike
- `TextListener` - listen for a change to a text field

The most useful listener is `ActionListener`, and this is the one we will be using most. As we look at the API for the `ActionListener` interface, we see that it has only one method: `actionPerformed (ActionEvent)`. This means that any class which implements `ActionListener` must define the `ActionPerformed` method. When an action occurs, any listener which is listening for that action will automatically call the `actionPerformed` method. It is our responsibility to define this method to handle the action before returning control to the Java runtime environment. The program is event-driven: nothing happens until an event causes a listener to respond.

The parameter for the `actionPerformed` method is of type `ActionEvent`. This parameter provides us with everything that is known about the event, as shown in the API for `ActionEvent`. These include, but are not limited to:

- Text from the component which cause the event (such as a button's caption)
- Time that the event occurred
- A reference to the component which caused the event; returned by the method `getActionCmd()` (this will be used to identify which component caused the event; e.g. which button was clicked)

Any component for which we need to handle events, must be *registered* with an event listener. This can be done with the the method
`addActionListener(ActionListener)`.

One last item needs to be addressed before we try to apply all this. How do we instantiate listeners? The easiest way to do this is to make our GUI an ActionListener. We can do this by declaring that it implements ActionListener. Then the GUI object is itself an `ActionListener`, and can be used as the actual parameter in the call to `addActionListener`. For example:

```
public class MyGUI implements ActionListener
{  private JButton myButton = new JButton ("click me");

   ...

   private void makeFrame()
   {  myButton.addActionListener (this);  // this MyGUI object
      ...
   }

   public void actionPerformed (ActionEvent evt)
   {  if (evt.getActionCmd() == myButton)
          // code to handle the button click
   }
}
```

In the `makeFrame` method we register an ActionListener with a button. The usage of the key word `this` should be explained. The key word `this` stores a reference to the object on which the method was called. In this case it refers to `this` MyGUI, which *is an* ActionListener. This usage of `this` is the same as that which was described previously in chapter 3.

Note in the `actionPerformed` method that we obtain a reference to the component that caused the action (returned by `getActionCmd()`) and compare it with the reference in the field `myButton`. In this case the comparison is `==` and not `.equals(Object)` because we are comparing references rather than the objects to which those references refer.

10.5.1 University Information System - version 2

We are now ready to build a GUI for our information system. We wish it to be capable of admitting students, displaying admitted students, and searching for a particular student by name or by ssn.

At this point we should clarify the purpose of a user interface; it should be used only to communicate with the user. All computations and processing of data should be done separately in what is often referred to as an *engine* or *kernel*. There should be a clear separation of the user interface and the engine. If this is done properly, we should be able to remove the graphical user interface for our information system, and plug in a command-line user interface, without making any changes to the engine.

To further clarify, we now define the engine, and we call it `UniversityInfoSys`. This class will have a set of students. Version 2 will be able to do the following:

- Add a student to the set
- provide a reference to the set of students
- search for a student by name or by ssn

Our `UniversityInfoSys` class is shown in the project `university-v2`. In this class we note that

- There is one field, a set of students.
- The constructor instantiates the set, it is now an empty set. The constructor also instantiates the GUI. This will cause the GUI frame to be initialized and to pop up on the monitor.
- The `addStudent` method adds one student to the set of students.
- The `getStudents` method is an accessor method which returns a reference to the set of students who have been admitted.
- The `searchByName` method will search for all students who have a given name, and return those students as a set.
- The `searchBySSN` method will search for the one student who has the given SSN.

10.5.2 Exercises

1. For each of the following interfaces in the package `java.awt.event` show all methods which must be defined in any class which implements the interface.
   (a) MouseListener
   (b) KeyListenerr
   (c) TextListener
2. Show the code necessary to print the caption of a button named `goButton` to stdout when the button is clicked. Assume the button has been registered with `this ActionListener`.
3. In the project `university-v2` in this chapter's repository the program will throw a NullPointerException and crash if the user cancels input for the name and/or ssn when attempting to admit a student. Correct this problem so that the student is not admitted, and an appropriate message is displayed on the output label.
4. In the project `university-v2` in this chapter's repository if the user attempts to enter two students with the same ssn, the first one is expelled and replaced by the second one. Modify this project so that the first student is retained, and an appropriate message is displayed on the output label.

10.6 Menus

One useful way of communicating commands to a GUI is through the use of *Menus*. A menu is normally a drop-down list of items, one of which can be selected with the mouse. Menus typically allow users to open or save to a disk file, terminate an application, edit data, etc. You can define menus to perform any action you wish. We will include two menus in the GUI in our information system: a File menu and a Help menu.

10.6.1 Adding menus to the frame

If you wish to include menus in your GUI, your frame must have a *menu bar* (there can be only one menu bar). You may then add several menus (e.g. File, Help) to the menu bar. Each menu may in turn have several menu items (e.g. Save, Save As, Open, Quit). A diagram of the menu bar for our University GUI is shown in Figure 10.11.

Menus are included in version 3 of our university information system. We will have a File menu which allows us to save our data to a disk file, and to retrieve our data from a disk file. We will also have a Help menu for the clueless users. The menus are set up in a separate method: `setMenus()`.

10.6.1.1 Setting the MenuBar

If your frame is to make use of menus, it must have a *menu bar* (We will use `JMenuBar` from `javax.swing` to be up-to-date). Think of this as a container, appearing as a rectangle as shown in Figure 10.11, for all the menus. A frame cannot have more than one MenuBar. The method used to include a JMenuBar is `setJMenuBar(JMenuBar)` as shown below:

```
JMenuBar menuBar = new JMenuBar();
frame.setJMenuBar (menuBar);
```

Note that this method is *not* `addJMenuBar`; the word 'add' would imply that a frame can have more than one menu bar.

10.6.1.2 Adding Menus to the MenuBar

One or more *menus* may be added to the menu bar. When a menu is created, it is provided with a String, which is the text shown on the menu. The menu can then be added to the menu bar with the method `add(JMenu)` as shown below:

```
JMenu fileMenu = new JMenu("File");
menuBar.add (fileMenu);
```

10.6.1.3 Adding the MenuItems to a Menu

Each menu may have 0 or more *menu items*. A menu item may be selected by the user to generate an event (e.g. open a file, quit, edit text, etc.). Again,

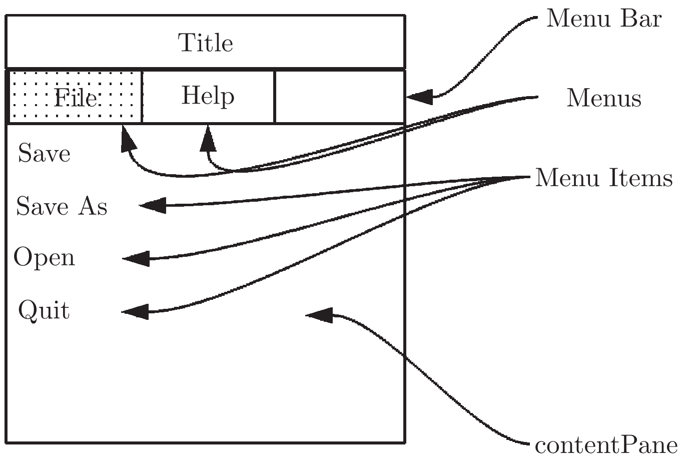


Figure 10.11: A simple frame with a title, menu bar, and contentPane

we will be using JMenuItem rather than MenuItem. When the menu item is created, it is provided with a String, which is the text shown on the menu item. To add a menu item to a menu, use the method `add(JMenuItem)`. A menu item which is supposed to save our data to disk can then be added to the File menu as shown below:

```
JMenuItem saveItem = new JMenuItem("Save");
fileMenu.add (saveItem);
```

The menu items in our GUI are declared as fields rather than as local variables because we may wish to refer to them from other methods in this class.

10.6.1.4 Other features of menus

Menu items may be *disabled* (or greyed-out) by using the method `setEnabled(boolean)`. For example:

```
   saveItem.setEnabled(false); // disable save
   saveItem.setEnabled(true); // enable save
```

The disabling of a menu item is desirable when its selection would not make sense, particularly if the user can cause an error by selecting the item. A good GUI will protect the novice user from making careless mistakes.

The java class library has a number of other features involving menus:

- Menu items may themselves contain other menu items
- menus can pop up at places in the frame other than the menu bar
- The menus on a menu bar, and the menu items on a menu can be changed as the program executes, to ensure that the user is always seeing options which are currently of interest

- Even the menu bar may be changed at run time, to ensure that the user is always seeing options which are currently of interest (at any one time there can be no more than one menu bar)

10.6.2 Listening for menu selection

At this point our GUI has menus, but they do not do anything; the user can make selections, but nothing happens as a result of the selection. Our GUI needs to *listen* for these actions and handle them appropriately.

The handling of a menu item selection is exactly the same as the handling of a button click; it is merely an action. If there is an action listener registerd with the menu item, the method `actionPerformed(ActionEvent)` is called automatically. Just as we did with buttons, we can use the `addActionListener(ActionListener)` to register an action listener with a particular menu item. In our example, the GUI class implements the interface `ActionListener`, so we can register the Save menu item as follows:

```
saveItem.addActionListener(this);
```

Having done this, when the user selects the Save menu item, the method `actionPerformed(ActionEvent)` is called automatically.

To handle this event in the `actionPerformed` method, we compare the reference provided by the parameter, `evt`, with a reference to the Save menu item. If they are equal (same reference), we know that `actionPerformed` was called as a result of the Save menu item selection. This is done as shown below:

```
public void actionPerformed (ActionEvent evt)
{   ...
    if (evt.getSource() == saveItem)
       infoSys.saveFile();
    ...  // handle other menu items here
}
```

Note that the actual saving to disk is *not* done in the user interface; it is done with a call to `saveFile()`, a method in the information system engine. In general all computation and processing should be done in the engine (or in other classes; we could have a separate class dedicated to input/output of data). A user interface should be used only for communication with the user.

10.6.3 Menus for the University Information System - version 3

The GUI for version 3 of our university information system has a menu bar with two menus:

- File menu - has 4 menu items:
  - Save menu item: save the set of students in the current disk file. Prompt for a file name if necessary.

  - Save As menu item: prompt the user for a new file name to be used as the current disk file, and save the set of students. Provide an error message if the file cannot be opened for output.
  - Open menu item: read a set of students from the current file; provide an error message if the file does not exist or cannot be opened for input.
  - Quit menu item: terminate the application.

- Help menu - intended to provide assistance for the novice user. Implementation of the menu items contained here is left as an exercise for the reader.

After we set up the frame and add the menus, we should tell the frame to resize itself appropriately to accommodate all the components which have been added, as well as the menubar. This is done with the `pack()` method:
`frame.pack();` // resize the frame
The engine for this project, UniversityInfoSys, is expanded to include methods for saving to and retrieving from disk. The class has a String, `fileName`, which stores the name of the file currently storing a permanent copy of our data. This file is in the same folder as the project, by default, though we could specify a path to a different folder. The file is a plain text file in which each student name and each student ssn is on a separate line. We will not go into the details of these methods here because they are not relevant to the subject of this chapter, but they apply the concepts of input and output as covered in chapter 9.

10.6.4 Exercises

1. Point out the error, if any, in each of the following:

   (a)
   ```
   JMenu menu = new JMenu ("File");
   frame.setJMenu (menu);
   ```
   (b)
   ```
   JMenuBar menuBar = new JMenuBar ();
   frame.addJMenuBar (menuBar);
   ```
   (c)
   ```
   JMenu menu = new JMenu ("File");
   JMenuItem menuItem = new JMenuItem ("Quit");
   menu.addItem (menuItem);
   ```
   (d) Assume the GUI class implements `ActionListener`
   ```
   JMenu menu = new JMenu ("File");
   JMenuItem menuItem = new JMenuItem ("Quit");
   menu.add (menuItem);
   menuItem.addActionListener (menu);
   ```
   (e) Assume there is a button named `goButton` and a method named `go()`.

```
public void actionPerformed (ActionEvent evt)
{  if (evt.getSource().equals (goButton))
        go();
}
```

2. (a) Which method in the `ActionEvent` class can be used to find out the exact time that the event occurred?
   (b) What are the units of time used in that method's result?
3. The project university-v3 in this chapter's repository includes a Help method which does not do anything. Improve the Help menu so that the user will understand the meanings of the three buttons. (Do NOT print to stdout nor to stderr. All communication with the user must go through the GUI.)

   Hint: See the API for `JOptionPane` in the javax.swing package.
4. Add a menu to the menubar which will ...

10.7 Projects

1. Build an application with a GUI which will test a student on integer arithmetic skills. Include the operations:
   - Addition
   - Subtraction
   - Multiplication
   - (Integer) quotient on division
   - (Integer) remainder on division

   The student should be rewarded for attempting more difficult operations and for quick responses. Include a menu to set the level of difficulty. An example of this program can be executed from the project `games` in this chapter's repository. Instantiate the Arithmetic class to start it up.
2. Build a java application that works like the one in the project `chase` in the code repository for this chapter. To execute it, simply instantiate the `Chase` class.

   Hints:
   - See the API for the `MouseListener` interface in the `java.awt.events` package.
   - Use a List of buttons, all of which are not visible, except for one button.
   - Use a GridLayout for the contentPane. Add all the buttons to the contentPane, and register each one with a MouseListener.

- In the `MouseEntered` method, make the selected button invisible, and choose some other button to become visible. (This can be done with a random number generator, or with clever use of the mod % operator)

Chapter 11

Abstract Data Types

In chapter 2 we discussed the various data types provided in java. Some examples of primitive data types are: int, float, double, char, boolean. In addition any class may be regarded as a data type - we can declare a variable that stores a reference to an object of a defined class.

In this chapter we examine classes which are designed to be used in the same way that primitive data types are normally used. In this way we can use software to improve, or enhance, the properties of the primitive data types provided by the hardware. We call these classes *abstract data types*.

Some programming languages, such as C++, provide features which make abstract data types even more attractive. One such feature, called *operator overloading* allows the programmer to define the semantics of primitive operators to include programmer-defined abstract data types. Unfortunately java does not allow operator overloading. Nevertheless we can define very useful abstract data types, but the user must understand the proper syntax for their use.

11.1 The Rational ADT

In mathematics a *rational* number is one that can be expressed as the ratio of two whole numbers. Some examples of rational numbers are: 3/2, 2/1, 4/2, 0/4, 5/-3.

In this section we develop an ADT for rational numbers. This ADT is motivated by some of the shortcomings of the floating point types.

11.1.1 Some problems with float and double

Java provides two floating point types: `float` and `double`. These types presumably reflect data types of the underlying hardware. I.e. most computers have a 32-bit floating point numeric format (which java calls `float`) and a 64-bit floating point numeric format (which java calls `double`).

These floating point types can exhibit unexpected behavior (which we will investigate further in the next section). Here are a few of the problems we might encounter:

- 1.0/3.0 produces the result 0.3333333333333333 which is close but not perfect. This repeating decimal cannot be represented exactly in base 10 (nor in its internal representation on the computer).
- $0.1 + 0.1 + 0.1 == 0.3$ is false. If you don't believe this, try it. The reason has to do with the fact that 0.1 cannot be represented exactly on a binary machine, for the same reason that 1.0/3.0 cannot be represented exactly in decimal.
- 1.0E200 + 1.0 == 1.2E200 is true. If you don't believe this, try it. This problem results from the limited precision available in the floating point formats. The Rational ADT will not address this problem, but we will address it in a later section.

Smart programmers will generally avoid comparing floats for equality to avoid some of these unexpected results. In cases where it is necessary to compare floats for equality, it is better to determine whether the two values differ by more than a very small tolerance. For example, if one wishes to compare the floats x and y for equality we wish to avoid:

```
   if (x == y)
```

Instead we will use a small tolerance, such as:

```
   float epsilon = 1.0e-7
```

and instead compare x and y as follows:

```
   if (Math.abs (x - y) < epsilon)
```

In this statement we determine whether the absolute value of the difference between x and y is smaller than epsilon; if so, we conclude that the values of x and y are sufficiently close that they can be considered equal.

11.1.2 Defining the Rational ADT

Some of the problems described above can be alleviated with a *Rational* ADT. Our ADT will store two ints to represent a rational number: a numerator and a denominator (the denominator must not be zero). In this way we can store exact values in cases where floats are not perfect. A numerator value of 1 and a denominator value of 3 is a perfect representation of one third.

Our ADT should include operations; we will define addition, subtraction, multiplication, and division of rationals to provide the user with a complete and useful ADT. We'll start with the class definition shown below:

```
/** Rational ADT
  * A Rational has a numerator and a denominator.
  * Rationals can be added, subtracted, multiplied, and divided
  */
public class Rational
```

```
{   private int num;              // numerator
    private int denom;            // denominator, cannot be 0

    /** @param d must not be 0
      */
    public Rational (int n, int d)
    {  num = n;
       denom = d;
    }

}
```

This means that we can create Rational objects such as:

```
Rational r;
r = new Rational (1,3);           //   1/3
r = new Rational (-7,2);          //   -7/2 = -3 1/2
r = new Rational (2,6);           //   2/6 = 1/3
r = new Rational (17,1);          //   17
```

Note that there can be more than one representation for the same rational number: $1/3 = 2/6$
If at some point we wish to compare rational numbers for equality, or we exceed the precision of ints, this could be a problem, which we will address here. The constructor should call a method to simplify this Rational, or reduce it to lowest terms.

```
    /** @param d must not be 0
      * Zero is always represented as 0/1
      */
    public Rational (int n, int d)
    {  num = n;
       denom = d;
       if (n == 0)
          d = 1;            //   normal form for 0
       else
          simplify();
    }
```

In this way Rationals should always be simplified when created (and we should avoid setting the numerator or denominator of an existing Rational). In other words, do this: `r = new Rational (n,d);` not this:

```
r = new Rational();
r.num = n;
r.denom = d;
```

The `simplify` method will utilize the fact that we can divide two numbers by their greatest common divisor to eliminate the factors which they have in common. For example, gcd(70,21) is 7 because: $70 = 2 \cdot 5 \cdot 7$ and $21 = 3 \cdot 7$. To simplify 70/21 we divide both numerator and denominator by 7 to obtain 10/3. The `simplify` method is shown below:

```
/**  Simplify this Rational, eliminating common factors
 *   from the numerator and denominator.
 */
private void simplify()
{   int g = gcd(num, denom);   // greatest common divisor of num and denom
    num = num / g;
    denom = denom / g;
}
```

To define the gcd method, we use a classic algorithm known as the *Euclidean* algorithm.

```
/** @return the greatest common divisor of a and b.
  * @param a must greater than or equal to 0
  * @param b must greater than 0
  */
private int gcd (int a, int b)
{  int r = a % b;
   while (r > 0)
       {  a = b;
          b = r;
       }
   return b;
}
```

Note that our `simplify` method works only for non-negative numbers. (Improvement is left as an exercise.)

We can now proceed to define some of the arithmetic operations for our Rational ADT. With all of our operations, we'll assume that the left operand is *this* Rational, and the right operand is the parameter. If we had operator overloading and the client wanted to multiply the Rationals r1 and r2, it would simply use: `r1*r2`. But since we do not have operator overloading it will be done this way:

`r1.mult(r2)`

Multiplication is the easiest so we'll start with that. Recall that when multiplying rational numbers we simply multiply the numerators and multiply the denominators: $(a/b) * (c/d) = (ac)/(bd)$

```
/** @return The product, this Rational multiplied by the parameter r
 */
public Rational mult (Rational r)
{  return new Rational (num * r.num, denom * r.denom); }
```

We next implement addition of Rationals. Recall that we can add the numerators only if the denominators are equal. To ensure the denominators are equal, we can multiply each operand by a number equal to 1:

$a/b + c/d = a/b * d/d + c/d * b/b = ad/bd + bc/bd = (ad + bc)/bd$

We define the addition method using this formula.

```
/** @return The sum, this Rational plus the parameter r
 */
public Rational add (Rational r)
{  return new Rational ((num*r.denom + denom*r.num)/
                        denom * r.denom); }
```

Subtraction and division will be left as exercises for the student. Note that we can now compute $0.1 + 0.1 + 0.1$ with perfect accuracy:

```
Rational tenth, result;
tenth = new Rational (1,10);            //  1/10
result = tenth.add(tenth);              //  2/10
result = result.add(tenth);             //  3/10
```

To print out a Rational we should have a `toString()` method:

```
/**  @return this Rational as a String
 *   Show numerator over denominator
 */
public String toString()
{  return num + "/" + denom;  }
```

To determine whether two Rationals are equal:

```
/** @return true only if this Rational represents
 *  the same number as obj.
 *  Pre: this Rational and obj have both been simplified.
 */
public boolean equals (Object obj)
{  if (obj==null | (! (obj instanceof Rational))
      return false;
   Rational r = (Rational) obj;
   // numerators and denominators must be equal
   return num == r.num && denom == r.denom;
}
```

If the user wishes to create a HashSet or HashMap of Rationals, we should supply a hashCode method, which agrees with the equals method: two Rationals which are equal should have the same hashCode.

11.1.3 Exercises

1. Define a division method for the Rational class:

   ```
   /** @return the result of dividing this Rational by
       the parameter, r.
       @param r is not zero.
     */
   public Rational divide (Rational r)
   ```

   Hint: $a/b \div c/d = ad \div bc$

2. Define a subtraction method for the Rational class:

   ```
   /** @return the result of subtracting
       the parameter, r, from this Rational.
     */
   public Rational subtract (Rational r)
   ```

3. Improve the `simplify` method called by the constructor, to allow for negative values for numerator and/or denominator.

4. Define additional constructors in the Rational class to allow convenient ways of constructing Rational numbers which are whole, and Rational numbers representing zero:

   ```
   Rational r = new Rational (17);            //    17/1
   Rational p = new Rational ();              //    0/1
   ```

   Hint: Eliminate duplicated code in your constructors by calling another constructor using the keyword *this*. A call to another constructor must be the first statement in your constructor.

5. Define a copy constructor for the Rational class. It should have one parameter, the Rational being copied.

6. Define a method in some class, to return the purchase price (including tax) on a purchase. Assume the retail price and the tax rate are both Rationals. For example, a tax rate of 7/100 is a rate of 7%. You may use the copy constructor defined in the previous exercise.

   ```
   /** @return the total purchase price, including tax.
     * @param retail The retail price, in dollars.
     * @param taxRate The tax rate
     */
   public Rational getCost (Rational retail, Rational taxRate)
   ```

7. Define a method in the Rational class which will round this Rational up to a given whole fraction. For example, if fraction is 100, it will return the Rational rounded up to the next higher one hundredth. Test your method in conjunction with your getCost method to obtain a purchase price rounded up to a penny (fraction = 100).

```
/** @return this Rational rounded up to the next higher
 *  fraction.
 *  If fraction is 100, it will round up to the
 *  one hundredth.  Can be used in financial applications
 *  to round to the (higher) penny,
 *  Pre: this Rational is not negative.
 */
public Rational roundUp (int fraction)
```

8. Show how to create Rational objects representing each of the following numbers.
   (a) 7
   (b) 4.1
   (c) -12.04

9. Define a toString() method for Rationals. The easiest way to do this is simply to include the character '/' between the numerator and denominator. An example could be `"13/2"`.

```
/** @return A String representation of this
 *  Rational.
 */
public String toString()
```

10. Improve your toString() method to return a representation in *composite* form, such as 2 1/3 rather than 7/3. Hint: Watch for negative values; in normal form, it is the numerator which will be negative.

11. Define an `equals (Object)` method for Rationals, so that the client can have a Set of Rationals. You may assume that this Rational and the parameter are both in normal form.

```
/** @return true only if the parameter obj
 *  is a Rational and is equal to this Rational.
 */
public boolean equals (Object obj)
```

12. Define a hashCode method for Rationals, so that the client can have a HashSet of Rationals. You may assume this Rational is in normal form.

```
    /** @return A hashCode for this Rational.
     *  Pre: this Rational is in normal form.
     */
    public int hashCode ()
```

13. We would like the Rational class to implement the `Comparable` interface, so that the client can have a TreeSet of Rationals. Change the class definition to read as follows:

    `public class Rational implements Comparable<Rational>`
    This means that a Rational can be compared with any other Rational for order as well as equality (i.e. greater than, less than). The compiler will require you to include a compareTo method:

```
    /** @return a negative value if this Rational is less than
     *  the parameter r, a positive value if this Rational
     *  is greater than the parameter r, and 0 if they
     *  are equal.
     *  Pre: Both this Rational and the parameter r
     *  are in normal form.
     */
    public int compareTo (Rational r)
```

11.2 MyFloat

In the previous section we examined some anomolies of floating point arithemetic. These resulted from the fact that certain values (such as 0.1) cannot be represented exactly on a binary machine, and from the limited precision of the floating point hardware. In order to gain a better understanding of floating point types, we will implement our own floating point ADT in this section. In doing so, we will use only whole numbers (i.e. ints). The presumption here is that floating point values (floats and doubles) do not exist, and that we are building this data type *from scratch.* In doing so, we will see many similarities with the Rational ADT of the previous section, and will use what we learned there.

We will call our ADT *MyFloat*, to distinguish it from the existing wrapper class in the java.lang package, Float. We begin by pointing out that every MyFloat can be represented by two whole numbers, and we call these the *mantissa* and the *exponent.* These whole numbers correspond to the two parts of a java constant when written in scientific notation, as in the following examples:

```
125e0 = 125.0
12e3 = 12000.0
12e-3 = 0.012
0e0 = 0.0
```

The mantissa of a MyFloat corresponds to the part of a floating point number before the `e`, and the exponent of a MyFloat corresponds to the part of a floating point number after the `e`. The exponent is always assumed to be an exponent of 10. In this way, any floating point value can be represented with two whole numbers.

11.2.1 Constructor for MyFloat

We begin our class definition as follows:

```
/** Every MyFloat has a mantissa and an exponent (of 10).
 *  MyFloat operations are addition, subtraction,
 *  multiplication, and division.
 */
public class MyFloat
{    int mant;
     int exp;        // exponent of 10

     public MyFloat (int m, int e)
     {  mant = m;
        exp  = e;
     }
}
```

As we noticed with the Rational ADT, MyFloat also has the property that there can be more than one representation for any number:
103e0 = 1030e-1 = 10300e-2 = 103000e-3 ... = 103.0
602e21 = 6020e20 = 60200e19 = 602000e18 ... = 6.02×10^{23}

We can eliminate trailing zeros in the mantissa:

```
/** Eliminate trailing zeros in the mantissa of this
  * MyFloat.  Adjust the exponent accordingly
  */
private void simplify ()
{  if (mant == 0)
      { exp = 0;              // Handle zero as a special case
        return;
      }
   while (mant % 10 == 0)     // Low order digit is zero?
      { mant = mant / 10;
        exp++;
      }
}
```

In the constructor we can now include a call to the `simplify()` method. As with the Rational ADT, we can be sure that all MyFloats are in normal form if created with our constructor:

```
/** Every MyFloat has a mantissa and an exponent (of 10).
 *  MyFloat operations are addition, subtraction,
 *  multiplication, and division.
 *  Constructed MyFloat will be in normal form.
 */
public class MyFloat
{    int mant;
     int exp;        // exponent of 10

     public MyFloat (int m, int e)
     {  mant = m;
        exp  = e;
        simplify();   // Put into normal form
     }
}
```

11.2.2 Arithmetic operations for MyFloat

We will now define the same four arithmetic operations for MyFloat which were defined for the Rational ADT: Addition, Subtraction, Multiplication, and Division. Again, we will be assuming that the left operand is `this` Myfloat, and the right operand is the parameter. Thus, if x and y are MyFloat objects, they can be multiplied to produce a product:

```
   product = x.mult(y);
```

We'll start with multiplication because that is the easiest operation. When multiplying two MyFloats we multiply the mantissas and add the exponents, as shown in the following examples:

`123e3 * 2e4 is 246e7`, i.e. `123000.0 * 20000.0 = 2460000000.0`
`3e-3 * 2e1 is 6e-2`, i.e. `0.003 * 20.0 = 0.06`
`123e0 * 1e-4 is 123e-4`, i.e. `123.0 * 0.0001 = 0.0123`
`5e0 * 6e0 is 3e1`, i.e. `5.0 * 6.0 = 30.0`

In the last example, note that the result has been simplified to normal form (i.e. it has been *normalized*).

Our multiply operation can be defined as shown below:

```
/** @return the product of this MyFloat multiplied
  * by the parameter, f.
  */
public MyFloat mult (MyFloat f)
{  int mantissa = this.mant * f.mant;
   int exponent = this.exp + f.exp;
   return new MyFloat (mantissa, exponent);
}
```

Note that we have taken care to do this so that the result has been simplified to normal form (i.e. the `simplify()` method is called from the constructor).

Division is similar to multiplication, but is not as easy. The following examples show that to divide MyFloats, we divide the mantissas and subtract the exponents:
`12e4 / 3e1 = 4e3`, i.e. `120000.0 / 30.0 = 4000.0`
`4e0 / 1e3 = 4e-3`, i.e. `4.0 / 1000.0 = 0.004`

These examples are straightforward and easy because the mantissa of the left operand is a divisor of the mantissa of the left operand in both cases. The next two examples expose the case where division of the mantissas will not produce the desired result
`1e0 / 4e3 = 25e-5`, i.e. `1.0 / 4000.0 = 0.00025`
`1e0 / 3e0 = 333333333e-8`, i.e. `1.0 / 3.0 = 0.33333333`

Recall that integer division produces an integer quotient; for example 1/4 is 0, thus dividing the mantissas will not give the desired result. To correct this problem in the first example above, we must *adjust* the left operand by making the mantissa larger, and the exponent smaller, to form an alternate representation of the same number.
`1e0 = 10e-1 = 100e-2`
We can now perform the division without loss of precsion:
`100e-2 / 4e3 = 25e-5`

In the case of the second example above, we need to adjust the left operand even more. Notice that the larger we make the mantissa of the left operand, the more precision we will get in the result:
`1e0 / 3e0 = 0e0`, i.e. `1.0 / 3.0 = 0.0`, not correct.
`10e-1 / 3e0 = 3e-1`, i.e. `1.0 / 3.0 = 0.3`, a little better.
`100e-2 / 3e0 = 33e0`, i.e. `1.0 / 3.0 = 0.33`, even better.
`1000e-3 / 3e0 = 333e0`, i.e. `1.0 / 3.0 = 0.333`, even better.

The solution is to make the mantissa of the left operand as big as possible (without exceeding the bounds of the int primitive data type). This value can be obtained from the wrapper class for ints: `Integer.MAX_VALUE`. This is done in the private helper method, `adjust`, shown below, leading to the following definition of the divide operation:

```
/** @return the quotient when this MyFloat is divided
  * by the parameter, f.
  * @param f Must not be a representation of zero
  */
public MyFloat divide (MyFloat f)
{  MyFloat temp = new MyFloat (mant, exp); // temp copy of this
   temp.adjust (f);                           // adjust with respect to f
   return new MyFloat (temp.mant / f.mant, temp.exp - f.exp);
}

/** Adjust this MyFloat with respect to the parameter f,
  * to provide maximum precision when dividing
  */
private void adjust (MyFloat f)
{   while (mant % f.mant != 0 && mant < Integer.MAX_VALUE/10)
      {  mant = mant * 10;
         exp--;
      }
}
```

In the adjust method above, we wish to stop the loop when the mantissa of the parameter, f, divides the mantissa of this MyFloat without a remainder. We also wish to stop the loop before overflowing the capacity of an int. Also note that we made a temporary copy of this MyFloat in the divide method; we do not wish to make any alterations to either operand when dividing.

The `adjust` method shown above presumes that the mantissas of this MyFloat is positive. We need to allow for the fact that it could be negative. To do this we will make use of the absolute value method provided in the Math class: Math.abs(int).

```
/** Adjust this MyFloat with respect to the parameter f,
  * to provide maximum precision when dividing
  * @param f must not be a representation of 0
  */
private void adjust (MyFloat f)
{   while (mant % f.mant != 0 && Math.abs(mant) < Integer.MAX_VALUE/10)
      {  mant = mant * 10;
         exp--;
      }
}
```

We are now ready for addition and subtraction. In order to add or subtract the operands must have the same exponent. For example, to add `12e3 + 3e4`, i.e. `12000.0 + 30000.0` we cannot simply add the two mantissas (this would produce a mantissa of 15, clearly not correct). Instead we must adjust one of the operands so that the exponents are equal. We choose to adjust the operand with the larger exponent, 3e4 in this case: `3e4 = 30e3`. We can now perform the addition by adding the mantissas and using the exponent of either operand (they are equal) for the exponent of the result:

`12e3 + 30e3 = 42e3`, i.e. `12000 + 30000 = 42000`.

Our addition method for MyFloat is shown below:

```
%%%%%%%%%%%%%% TEST THIS
/** @return the sum when this MyFloat
```

```
  * and the parameter, f, are added.
  */
public MyFloat add (MyFloat f)
{  MyFloat temp = new MyFloat (mant, exp);          // temp copy of this
   MyFloat tempF = new MyFloat (f.mant, f.exp);     // temp copy of f
   temp.adjustAdd (tempF);                          // adjust either this or f
   return new MyFloat (temp.mant + tempF.mant, temp.exp);
}

/** Adjust either this MyFloat or f
  * so that they have the same exponent.
  */
private void adjustAdd (MyFloat f)
{  while (exp > f.exp)
      {   exp--;
          mant = mant*10;
      }
   while (f.exp > exp)
      {   f.exp--;
          f.mant = f.mant*10;
      }
}
```

The subtract method will be almost identical to the add method.

11.2.3 Exercises

1. Show how to create MyFloat objects representing each of the following numbers.
    - 7
    - -12.04
    - 0.00032
    - 6.02x10 sup 23
    - 6.02e23
2. What is the value of 12e23 * 3e10 / 4e12 ?
3. Define a toString() method for MyFloats. The easiest way to do this is simply to include the character 'e'or 'E' between the mantissa and the exponent, producing a result such as `"32e-4"` or `"1e3"`.
4. Improve your toString() method so as to produce more conventional results such as `"0.0032"` instead of `"32e-4"` and `"1000.0"` instead of `"1e3"`. You'll need to decide on an arbitrary criterion to produce output with the "e" notation.

```
/** @return A String representation of this
 *  MyFloat.
 */
public String toString()
```

Hint: Convert the mantissa to a String, then handle each of the following cases separately, in the order shown (LIMIT is a constant (perhaps 6) representing the number of zeroes to be inserted before resorting to the 'e' notation):

Case	mant	exp	result
$exp == 0$	123	0	"123.0"
$exp > LIMIT$	123	12	"1.23e14"
$exp > 0$	123	3	"123000.0"
$-exp > mant.len + LIMIT$	12345	-10	"1.2345e-6"
$-exp > mant.len$	123	-4	"0.0123"
$default$	123	-2	"1.23"

5. Define a subtract method for MyFloats:

```
/** @return the difference produced when
 *  the parameter f is subtracted from
 *  this MyFloat
 */
public MyFloat subtract (MyFloat f)
```

6. Define additional constructors in the MyFloat class to allow convenient ways of constructing MyFloats which which have an exponent of 0, and MyFloats representing zero:

```
MyFloat r = new MyFloat (17);            //    17e0
MyFloat p = new MyFloat ();              //    0e0
```

Hint: Eliminate duplicated code in your constructors by calling another constructor using the keyword *this.* A call to another constructor must be the first statement in your constructor.

7. Define an `equals (Object)` method for MyFloats, so that the client can have a Set of MyFloats. You may assume that this MyFloat and the parameter are both in normal form.

```
/** @return true only if the parameter obj
 *  is a MyFloat and is equal to this MyFloat.
 *  Pre: Both this MyFloat, and the parameter,
 *  are in normal form.
 */
public boolean equals (Object obj)
```

8. Define a hashCode method for MyFloats, so that the client can have a HashSet of MyFloats. You may assume this MyFloat is in normal form.

   ```
   /** @return A hashCode for this MyFloat.
    *  Pre: this MyFloat is in normal form.
    */
   public int hashCode ()
   ```

9. We would like the MyFloat class to implement the `Comparable` interface, so that the client can have a TreeSet of MyFloats. Change the class definition to read as follows:

   `public class MyFloat implements Comparable<MyFloat>`
   This means that a MyFloat can be compared with any other MyFloat for order as well as equality (i.e. greater than, less than). The compiler will require you to include a compareTo method:

   ```
   /** @return a negative value if this MyFloat is less than
    *  the parameter f, a positive value if this MyFloat
    *  is greater than the parameter f, and 0 if they
    *  are equal.
    *  Pre: Both this MyFloat and the parameter f
    *  are in normal form.
    */
   public int compareTo (MyFloat f)
   ```

10. Try the following operation in Java (if using BlueJ, you can use the code-pad feature).

    3e200 + 1e0

    - Explain why the result is not perfectly accurate.
    - Try this same operation with MyFloats. If an error occurs, correct it so that a reasonable result is produced.

11. Try the following operation in Java (if using BlueJ, you can use the code-pad feature).

    123456789e0 * 123456789e0

    - Explain why the result is not perfectly accurate.
    - Try the same operation with MyFloats. If an error occurs, correct it so that a reasonable result is produced.

11.3 BigNumber

Our third example of a useful ADT is one which allow arithmetic with unlimited precision. A java `int` is represented by 32 bits, and thus has limits (see Integer.MAX_VALUE and Integer.MIN_VALUE). When the result of an arithmetic operation exceeds these limits (known as *overflow*), no Exception is thrown; the program continues to execute with values that are almost certainly not valid. Even if we use the primitive type `long` (64 bits) instead, there are still limits to the values which can be computed and stored.

In this section we will define an ADT named *BigNumber* which will be used to represent whole numbers that are not limited in size. [1]. Numbers with unlimited precision have many important applications in today's world, most notably in the areas of cryptography and computer security. BigNumbers are keeping the internet alive today.

Our strategy is to represent a whole number as a list of decimal digits. Each digit will be in the range [0..9]. For example, the number 423,502 will be represented as the list [4,2,3,5,0,2]. With BigNumbers we will need to find a suitable representation for negative numbers, and here we will follow the example of computer chip design; we will use a representation similar to two's complement.

11.3.1 Constructing BigNumbers

Our BigNumber class will need only one field: the List of digits. We need to decide what kind of List this should be in the Constructor. Recall that ArrayLists are designed to be efficient when accessing specific elements directly, but not efficient when the size of the List is often changed. As far as we can see now, with BigNumbers we will not need to access digits in the middle of the List, but instead will process the digits in sequentially, from low order digit to high order digit. For this reason we choose to implement our List of digits with a LinkedList, noting that if we made the wrong choice it should be easy to change it to an ArrayList.

```
/** A BigNumber represents a whole number with unlimited
 *  precision.
 */
public  class BigNumber
{   private List<Integer> digits
        = new LinkedList <Integer> ();

    /** Default constructor is used in the add
     *  method.
```

[1]Of course every computer has a finite memory, hence there will be limits on the size of a BigNumber, yet our software will be designed so as to impose no limit on the size of a BigNumber, and the client can potentially allow for larger numbers by adding more memory to the computer

```
     */
    BigNumber ()
    {   }

    /** Construct a BigNumber from a String
     *  of numeric characters.
     *  @param num is a String of numeric
     *  characters [0..9]
     */
    public BigNumber (String num)
    {  // Process the digits from low-order to high-order
       for (int i=num.length()-1; i>=0; i--)
           digits.add (num.charAt(i));
    }

}
```

The client can now create very large numbers such as:

```
   BigNumber huge = new BigNumber ("2348099039393920349820439834");
```

Note that the constructor will put the low-order digit (the one at the right, as normally viewed, into position 0 of the List. If you are accustomed to seeing position 0 at the left end of a List, the digits of a BigNumber will appear to be in reverse order.

11.3.2 Adding BigNumbers

We now wish to be able to add BigNumbers. We will do this by adding corresponding digits, beginning at the low-order digit, and working toward the high-order digit, adding in a carry (0 or 1) at each position.

In our add method, we use three loops (not nested). The first loop continues as long as there are more digits in both operands. The second loop continues as long as there are more digits in this BigNumber, and the third loop continues as long as there are more digits in the parameter. At least one of these loops will repeat 0 times. We use `carry` to carry a 1 into the next 'column'. When all loops terminate, we can add the carry at the high order position of the result. At this point we are assuming BigNumbers are not negative.

```
/** @return The sum of this BigNumber and the
 *  parameter, b.
 */
public BigNumber add (BigNumber b)
{   int carry = 0;
    int sum;
    BigNumber result = new BigNumber();
    Iterator<Integer> ittyThis = digits.iterator();
    Iterator<Integer> ittyB = b.digits.iterator();

    while (ittyThis.hasNext() && ittyB.hasNext())
    {  sum = ittyThis.next() + ittyB.next() + carry;
       carry = sum / 10;
       result.add (sum % 10);
    }
    // This loop may execute 0 times
    while (ittyThis.hasNext())
    {  sum = ittyThis.next() + carry;
       carry = sum / 10;
       result.add (sum % 10);
    }
    // This loop may execute 0 times
    while (ittyB.hasNext())
    {  sum = ittyB.next() + carry;
       carry = sum / 10;
       result.add (sum % 10);
    }
    if (carry == 1)
       result.add (carry);
    return result;
}
```

11.3.3 Subtracting BigNumbers

When subtracting, it is possible to obtain a negative result; consequently we need to think about how to represent negative BigNumbers. Here are a few possibilities:

1. Sign and Magnitude - Store the magnitude of the BigNumber as we are doing currently; also store a boolean which indicates whether the BigNumber is negative.

2. Nines Complement - Negate a BigNumber by subtracting each digit from 9 (similar to ones complement for binary numbers). For example, -04096 would be stored as 95903. Then when adding we would need to add 1 to the result, only if one of the operands is negative.

3. Tens Complement - Similar to two's complement for binary numbers, -1 would be represented by all 9's (9999), -2 would be 9998, -3 would be 9997, etc.

We will use tens complement representation for a few reasons:

- This will simplify subtraction: $a - b = a + (-b)$. All we need now is a negate method, to find -b (though we will need to modify our add method).

- Tens complement is analogous to twos complement for binary numbers. Hence, tens complement will reinforce your understanding of twos complement representation which is used in virtually all digital hardware.

In tens complement representation, the number is negative if and only if the high order digit is greater than 4. To determine the value of a negative number, we can negate it. Here are three algorithms for negating a number in Tens Complement:

- Subtract it from 0. For example, to negate 345:

```
    000          Extend the zeroes
   -345
-------
    655  =  -345
```

- Find the nines complement, and add 1. For example, to negate 345:

```
    999
    345      Nines complement, subtract each digit from nine
   ----
    654
   +  1      Add 1
   ----
    655 =   -345
```

- Scan the digits right to left:

  1. copy zeros
  2. subtract first non-zero digit from ten
  3. subtract all remaining digits from nine

  For example, to negate 3405000:

```
        3405000
            000       copy zeros
           5          subtract first non-zero digit from ten
        659           subtract remaining digits from nine
--------------
        6595000  =  -3405000
```

  If it seems strange to you that 6595000 should be equal to -3405000, remember that 6595000 represents a negative number because its high order digit is greater than 4. Try adding 6595000 + 3404000 (and discard the carry out of the high order digit). The result should be 0, proving that they are complements of each other.

  We prefer this algorithm, and recommend it be used in our BigNumber class.

Here are some other examples of tens complements:

```
-280 = 720
-0509 = 9491
-0500 = 500 = 9500 = 99500
```

Note that a non-negative number may have leading zeros, and a negative number may have leading nines; thus there are multiple represenations of the same number (as with our other abstract data types Rational and MyFloat).

Number	Tens Comp	Alternates	
+3	3	03	003
+10930	10930	010930	0010930
-3	7	97	997
-8	92	992	9992
+499	499	0499	00499
-499	501	9501	99501
+500	0500	005000	000500
-500	500	9500	99500
-501	599	9599	99599

A normal form for BigNumber objects is described as:

- A non-negative BigNumber has a minimum number of leading zeros.
- A negative BigNumber has a minimum number of leading nines.

Before we can implement subtraction of BigNumbers we will need to define the *negate* method. We will use the third negate algorithm described above.

```
 /**@return this BigNumber negated, tens complement */
 private BigNumber negate()
 {   BigNumber result = new BigNumber();
     Iterator <Integer> itty = digits.iterator();
     int d = itty.next();
     while (d==0 && itty.hasNext())      // copy zeros
         {   result.digits.add (0);
             d = itty.next();
         }
     result.digits.add ( (10-d) % 10);            // copy 10 - digit
                                            // ensure digits is not empty.
     while (itty.hasNext())
         {   result.digits.add (9-itty.next());  }    // copy 9 - digit
     return result;
}
```

Now that we have a negate method, the subtract method is easy: $a - b = a + (-b)$.

```
/**  @return the result of subtracting the parameter, b,
 *   from this BigNumber.
 */
public BigNumber subtract (BigNumber b)
// a-b = a+(-b)
{
   return this.add (b.negate());
}
```

However, we will need to make a few minor changes to our add method, now that we are working with negative numbers (this is left as an exercise for the student):

- In the loops which accomodate the fact that one of the operands may have fewer digits than the other operand, we assumed leading zeros for the shorter operand. Now we will have to determine whether the shorter operand is negative; if so, assume leading nines. This can be done easily using a fill digit:

  `int fill=0;`

  If the shorter operand is negative, change the fill digit to 9. Add in the fill digit on each iteration.
- When adding two non-negative operands, the result should be non-negative. If this is not the case (i.e. *overflow* has occurred), append a leading zero. For example, when adding the positive numbers 402+350 you will get 752, which is negative because the high order digit is greater than 4; append a leading zero to produce 0752.
- When adding two negative operands, the result should be negative. If this is not the case (i.e. *overflow* has occurred), append a leading nine. For example, when adding the negative numbers 702+650 you will get 352, which is non-negative because the high order digit is less than 5; append a leading nine to produce 9352.
- A private helper method to determine whether a BigNumber is negative may be helpful; it will merely compare the hight order digit with 5.

Having defined addition and subtraction, how can we define multiplication, division (and mod) using what we have already built? We leave these as exercises for the student; here are some hints:

- Multiplication could be done by repeated addition: $4 * 7 = 7 + 7 + 7 + 7$.
- Remember, both operands are BigNumbers. This means that a loop is needed, and it will be controlled by decrementing the left operand until zero is reached. In the body of the loop the right operand is added into the result.

- If the left operand is negative, work with its negation in order to control the loop.

This is a fairly slow algorithm for multiplication of BigNumbers. A faster algorithm is called *shift and add*:
The multiplier is this BigNumber, and the multiplicand is the parameter.

1. Define a private helper method to multiply this BigNumber by a decimal digit returning the product.

   ```
   /** @return the product of this BigNumber multiplied by
    *  the parameter, i.
    *  @param i is in the range 0..9
    */
   private BigNumber multByDigit (int i)
   ```

2. Use a ListIterator to iterate over the digits in this BigNumber from high order to low order digit.
3. Multiply the parameter by the high order digit, storing the product in a local BigNumber, result.
4. Continue to iterate over the digits in this BigNumber using your ListIterator (using methods hasPrevious() and previous()):
   (a) Shift the result (simply insert a 0 at position 0)
   (b) Multiply the parameter by the next (i.e. previous) digit of this BigNumber, storing the result in a temporary BigNumber.
   (c) Add the temporary BigNumber into the result.

A slow algorithm for division of BigNumbers is fairly easy. Division should in general produce two results: a quotient and a remainder, since we are working with whole numbers. We should design our software so that the client can obtain both of these values without repeating the division. To do this, the divide method should return a List of BigNumbers; the first element in the list is the quotient and the second element is the remainder.

```
/** @return Both the quotient and the remainder when this
 *  BigNumber is divided by the parameter b
 */
public List<BigNumber> divide (BigNumber b)
```

The algorithm for division (this BigNumber is the dividend, and the parameter is the divisor):

1. Copy the dividend to a local BigNumber, dividendTmp.

2. Loop while the dividendTmp is greater than the divisor (use a BigNumber counter, to count the number of times the loop repeats):

   (a) Subtract the divisor from the dividendTmp storing the result in dividendTmp.

3. When the loop terminates, the loop counter is the quotient, add it to the result list.

4. The dividendTmp is the remainder, add it to the resultlist.

Now that you have a BigNumber class it could be tested as shown below (commment out the lines which have not yet been implemented):

```
    public static void main()
    {   Scanner scanner = new Scanner (System.in);      // read from stdin
        BigNumber x,y;
        String input;
        System.out.println ("Enter a Big Number, or Enter to terminate");
        while (scanner.hasNextLine())
            {
// read a big number from keyboard
                input = scanner.nextLine();
                if (input.length() == 0)
                    return;
                x = new BigNumber (input);

// read a big number from keyboard
                System.out.println ("Enter another Big Number");
                input = scanner.nextLine();
                if (input.length() == 0)
                    return;
                y = new BigNumber (input);
                System.out.println ("x+y: " + x.add (y));
                System.out.println ("y+x: " + y.add (x));
                System.out.println ("x-y: " + x.subtract (y));
                System.out.println ("y-x: " + y.subtract (x));
                System.out.println ("Enter a Big Number, or Enter to terminate");
            }
        }
```

11.3.4 Exercises

1. Complete the following table using tens complement representation.

   Hint: When viewing a number in tens complement representation, the first decision to be made is whether the number is positive or negative.

Number	Tens Comp Using 4 digits, if possible	Tens Comp Using as many digits as needed, but no more
0	0000	0
2		
4		
5		
-4	9996	6
-5		
-6		
25		
92		
-25		
-92		
499		
500		
501		
-499		
-500		
-501		
4999		
5000		
-5000		
-5001		
-394920	Not Possible	

2. Complete each of the following operations, assuming tens complement representation (use as many digits as necessary for the result):

(a)

```
  0003 = +3
+ 0097 = +97
------------
```

(b)

```
  4004 = +4004
+ 3097 = +3097
--------------
```

(c)

```
  0003 = +3
- 0097 = +97
------------
```

(d)

```
  9999 = -1
+ 0100 = +100
------------
```

```
  9999 = -1
+ 9998 = -2
------------
```

3. Include a toString() method in your BigNumber class:

   ```
   /** @return this BigNumber as a String. */
   public String toString()
   ```

4. (a) Revise the toString method so that it will handle negative values, in tens complement representation.

   Hint: If the number is negative, negate it, produce the result string, and append a '-' at the beginning.

   (b) Revise the add method so that it works with tens complement values. Use the main method given at the end of this section to test your solution.

   Hint: Use a *fill digit* for the operand that has fewer digits. The fill digit should be 9 if the operand is negative, and 0 otherwise. Discard the carry out of the high order digit. Add a 0 (or 9) at the high order digit if necessary to ensure the correct sign of the result.

   (c) Implement the subtract method. Test subraction using the main method.

   Hint: a - b = a + (-b)

   (d) There are several tens complement representations for the same number:

   ```
   +17 = 17 = 017 = 0017 = 00017 ....
   +93 = 093 = 0093 = 00093 ....
   -19 = 81 = 981 = 9981 = 99981 ....
   -82 = 918 = 9918 = 99918 = ....
   ```

   We can define a *normal form* by choosing the representation which has no unnecessary leading zeros (or nines for negative numbers). In the examples given above, the normal form of each number is shown first. Define a method which will normalize this BigNumber, and test with the main method. All newly created BigNumbers should be normalized.

```
/** Put this BigNumber into normal form.
 *  Eliminate unnecessary leading zeros or nines
 */
private void normalize()
```

(e) Revise the toString() method, if necessary, so that it does not print unnecessary leading zeros.

5. Implement multiplication using repeated addition: 5*3 = 3+3+3+3+3

   The method signature should be:

   ```
   /** @return the product of multiplying this BigNumber by b. */
   public BigNumber multiply (BigNumber b)
   ```

   Hint: Check the signs of the operands first, if they are different you know the sign of the result should be negative. Then negate each operand which is negative, and do the multiplication with non-negative values. Then negate the result if necessary.

6. Improve your multiply method to use a shift and add algorithm as described in this section.

7. Implement division using repeated subtraction. Division should produce two BigNumbers as results: a quotient and a remainder. The signature is:

   ```
   /** @return the quotient and remainder (in that order)
        when this BigNumber is divided by b.
       @throws DivideByZeroException if b is 0.
    */
   public List<BigNumber> divide (BigNumber b)
   ```

   If the dividend and divisor have different signs, the quotient should be negative. The sign of the remainder should be the same as the sign of the dividend.[2]

8. Make improvements to your divide method so that it is faster; this is similar to the shift and add algorithm for multiplication, but instead it will be shift and *subtract*.

[2]Caution: Various platforms do not agree on the correct sign for the remainder when one or both operands is negative. The sign proposed here agrees with the Java virtual machine.

Chapter 12

Algorithms: Sorting and Searching

This chapter contains an introductory discussion of sorting and searching algorithms. This subject is covered more extensively in Data Structures textbooks. We include it here because sorting and searching are included in the College Board's Advanced Placement exam for Computer Science.

An *algorithm* is a well-defined series of steps leading to the solution of a given problem. An algorithm must terminate with a correct solution. Note that the concept of an algorithm is independent of a particular implementation, or programming language used, for that algorithm. For example, we discussed the *sequential search* algorithm in chapter7. The sequential search method could have been written in C++, Python, or any other programming language. Also, it could have been written in a very different way in Java; it could have been a recursive method instead of using a loop. These would all be implementations of the *same algorithm*.

There are some problems which become very complex as the size of the input increases. A classic example is the *traveling salesman* problem: Given a map showing cities and roads connecting the cities, find a shortest path which enables the salesman to visit every city exactly once. There are algorithms to solve this problem, and if the number of cities is 10, they work quite well. However, if there are 1000 cities, your program will take too long to execute. For problems such as this we may choose to use a method which is not guaranteed to produce a correct solution, but which executes quickly enough that we will see it terminate. This kind of solution is called a *heuristic*. It is important to understand that a heuristic is not an algorithm; for problems like the traveling salesman problem a heuristic can be more useful than an algorithm.

12.1 Searching: Binary Search

In chapter 5 we discussed the problem of searching a list for a given target value. This was the `indexOf(Object)` method in the list classes (for the API, see the List interface in the `java.util` package of the java class library). This method performs a *sequential* search for the given object (i.e. the target). It begins at the first element of the list, comparing with the target, and proceeding to each element of the list until it either finds an element which is equal to the target (in which case it returns its index in the list), or reaches the end of the list, in which case it returns -1. If it finds the target, it will therefore return the index of the *first* occurrence of that value (there may be duplicate values in a list).

In this section we discuss an improvement to the sequential search algorithm. If the list being searched is sorted in ascending (or descending) order we can use an algorithm known as *binary search* to locate a given target value.

Imagine that you have an old (paper) telephone book for a nearby city. If you needed to find out who lives at "332 N. Main St", you would have to begin by looking at every entry on page 1, and continue through every entry in the phone book until you either find that address, or reach the end of the book, in which case you would conclude that the address "332 N. Main St." is not in the phone book.

However, if you need to find "Smithson, John" in that phone book, it would be much faster. If the first entry you look at is "Potter, James", then you know that if "Smithson, John" is in the book, it would have to come after "Potter, James". Thus you immediately exclude from consideration those entries coming before "Potter, James". This is an informal description of the binary search algorithm. After each comparison, half of the values are eliminated from consideration; we can ignore them, speeding up the search considerably.

The reason that looking up a name is so much faster than looking up an address is that the phone book is *sorted* by name, but it is *not* sorted by address. Thus the binary search algorithm applies only when the data are sorted prior to beginning the search.

The binary search algorithm can be expressed as follows:
Search a given list for a given target value, given the starting and ending indices of the list being searched

1. If the starting index is greater than the ending index, terminate. The target is not in the list.

2. Calculate the index of the midpoint, by averaging the start and end indices:
   `mid = (start+end)/2`[1]

3. If the target is equal to the value at position `mid`, the target has been found; return `mid`, the position of the target.

[1]Note that if start+end is odd, the result is rounded down to an int. E.g. 15/2 = 7. This works fine as the midpoint.

4. If the target is smaller than the value at position `mid`, then the target must be in the left portion of the list (if it is in the list). Search the left portion of the list (excluding the mid point) using the binary search algorithm, from `start` to `mid-1`.

5. If the target is greater than the value at position `mid`, then the target must be in the right portion of the list (if it is in the list). Search the right portion of the list (excluding the mid point) using the binary search algorithm, from `mid+1` to `end`.

Note that this algorithm is expressed recursively. The base cases are steps 1-3. The recursive cases, steps 4 and 5, reduce the size of the input; thus it satisfies the desired properties of recursion.[2]

The binary search algorithm is diagrammed in Fig 12.1, in which the sorted list is [-7,-4,2,3,5,7,8,8,11,11,12,13,17,17,22] and the target is 11. On the first iteration, `start=0` and `end=14`. The midpoint is calculated as `mid = (0+14)/2 = 7`. Since the value at position 7 is 8, and the target, 11, is greater than 8, the target must be in the right half of the list; search the portion of the list from `start=8` to `end=14`, using the same binary search algorithm.

At this point the midpoint is calculated as `mid = (8+14)/2 = 11`. The value at position 11 is 13, which is less than the target. Thus if the target is in the list, its position must be in the range [8..10]. Thus on the next iteration `start=8` and `end=10`.

When searching this part of the list, `mid = (8+10)/2 = 9`. At this point the algorithm finds the target,11, at the midpoint, and terminates by returning its position, 9.

Note that the algorithm did not find the position of the *first* occurrence of the target. It is sufficient to return the position of *any* occurrence of the target.

Each time this algorithm invokes the binary search algorithm recursively, the start and end indices get closer together. If the target is not in the list, then the start index becomes greater than the end index, and the algorithm terminates with the base case in step 1. This case is diagrammed in Fig 12.2, in which the target, 4, is not in the list. When `start`=4 and `end`=3, the algorithm determines that the target 4 is not in the list.

A java method for the binary search algorithm is shown in Fig 12.3. The list to be searched is defined as a field in the class, and is assumed to be initialized.[3] We use a recursive helper method, `binSrch` to search the portion of the list from position `start` to position `end` for the target. There are two base cases in `binSrch`:

- `start > end`: The target is not in the list.

- The value at the midpoint is equal to the target. The target has been found; return its position: `mid`.

[2]This algorithm could have been expressed just as easily with a loop; we use recursion simply to gain some additional experience with recursion.

[3]The list should be an ArrayList, for efficiency.

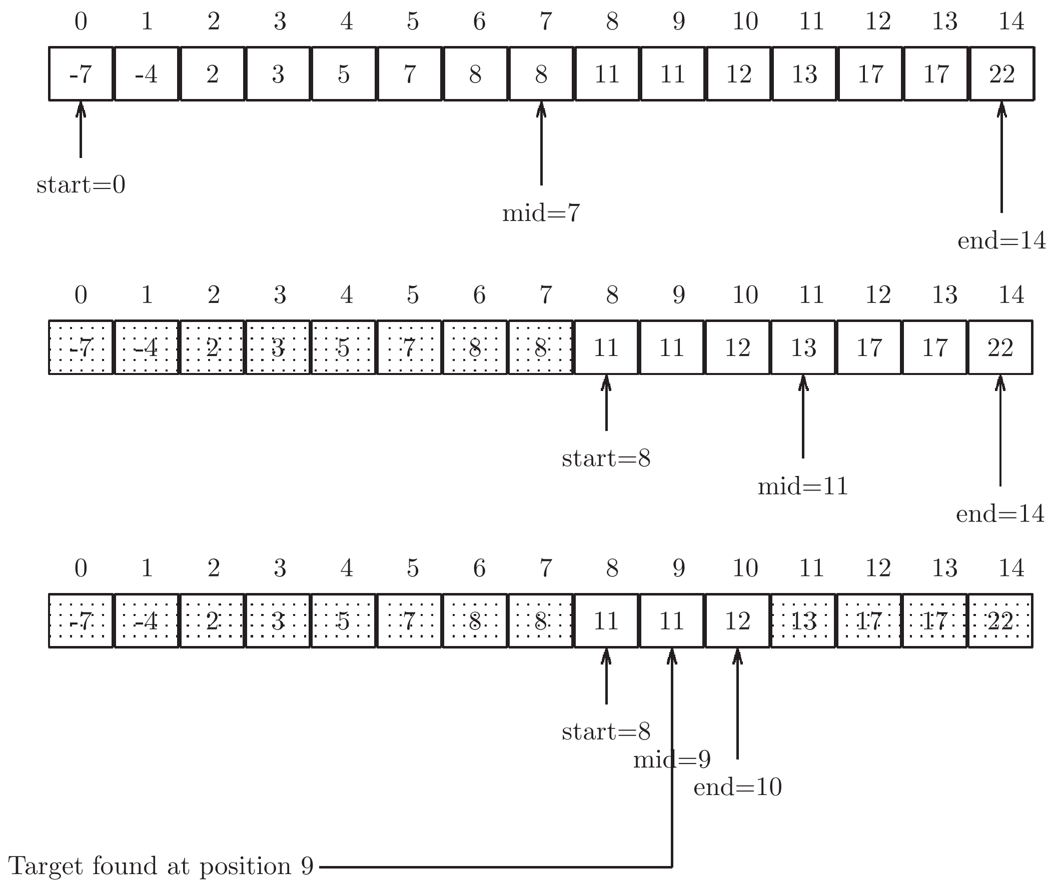
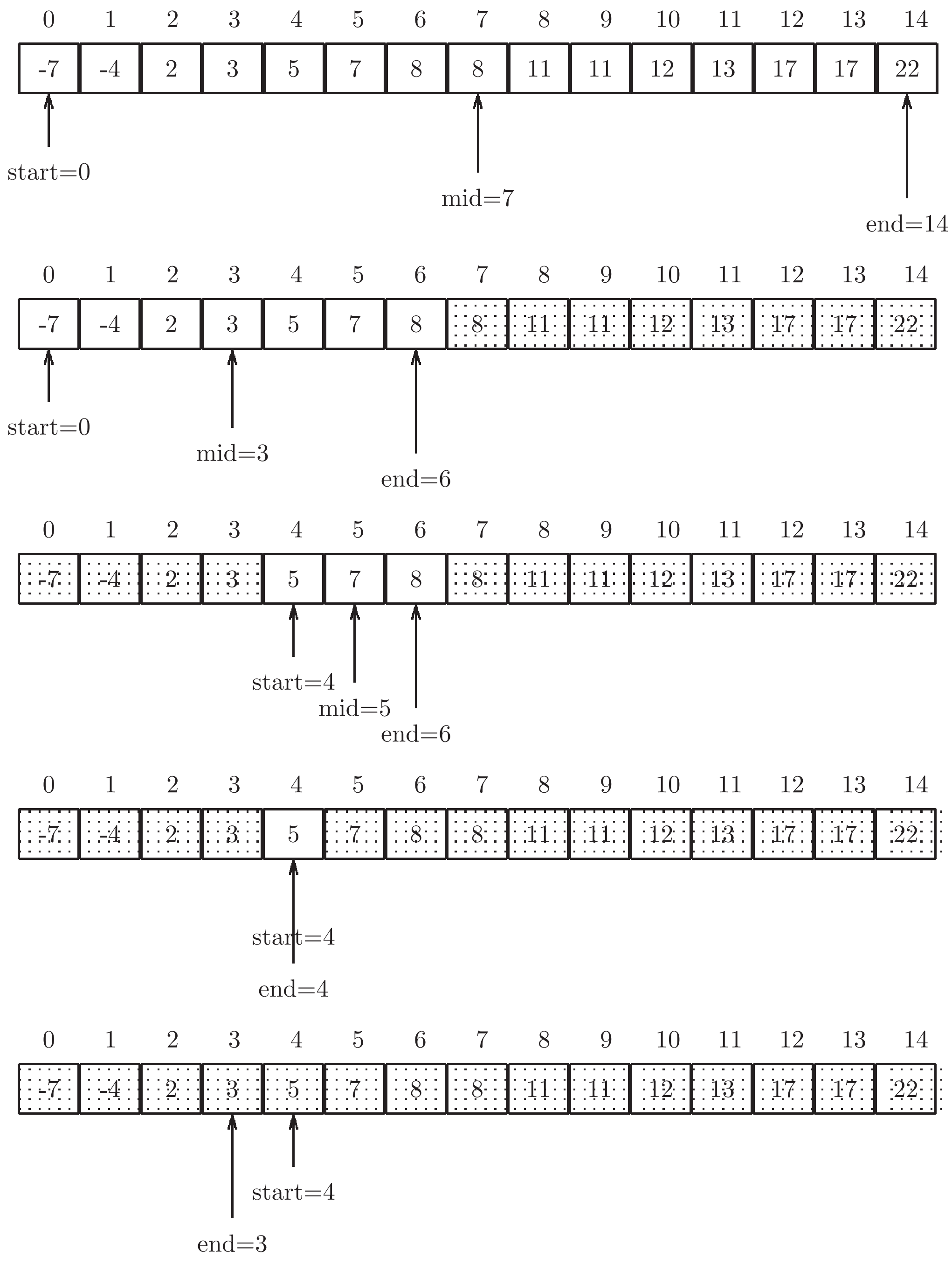


Figure 12.1: Binary Search algorithm, on a list of size 15. The target is 11, found at position 9. There is no need to search the shaded regions.



Target not found

Figure 12.2: Binary Search algorithm, on a list of size 15. The target, 4, is not found in the list.

```
List<Integer> list;     // Should be an ArrayList
   ...
public int search (int target)
   {    return binSrch (0, list.size()-1, target);   }

   /** @return A position of the target in the
    * portion of the list from start .. end, or
    *  -1 if not found,
    */
private int binSrch (int start, int end, E target)
   {    if (start > end)
            return -1;
        int mid = (start + end) / 2;
        if (target == list.get(mid))
            return mid;
        if (target < list.get(mid))
            return binSrch (start, mid-1, target);
        // target > list.get(mid)
        return binSrch (mid+1, end, target);
    }
```

Figure 12.3: Method to search a sorted list of Integers for a given target value, using the binary search algorithm.

There are two possible recursive cases in `binSrch`:

- The target is smaller than the value at the midpoint; search the list from positions `start` thru `mid-1`, inclusive.
- The target is larger than the value at the midpoint; search the list from positions `mid+1` thru `end`, inclusive.

.

12.1.1 Exercises

1. Given the List of Fig 12.1 show the values assigned to the variable `mid` for each of the following target values:

   (a) `target = 8`

   (b) `target = 13`

   (c) `target = 2`

   (d) `target = 15`

   (e) `target = -400`

2. Show a diagram similar to Fig 12.1 for each of the targets given in the previous exercise (and the list given in that figure).

3. Show a binary search method similar to Fig 12.3 in which we are searching a sorted *array* of ints, rather than a sorted List of Integers.

4. Show a binary search method similar to Fig 12.3 which uses a loop rather than a recursive helper method.

5. How many iterations (or calls to `binSrch`) would occur when searching for a target that is not in the given list if:

   (a) The size of the list is 7.

   (b) The size of the list is 15.

   (c) The size of the list is 1023.

   (d) The size of the list is 2^{k-1} for some integer k.

   (e) The size of the list is n.

6. (a) Rewrite the search method in Fig 12.3 to search a List of Strings for a given target String.
   Hints:

   - A String, s1, is smaller than another String, s2, iff s1 precedes s2 alphabetically.
   - See the compareTo method in the String class.

   (b) If you are familiar with generic types in java, and typed classes, rewrite the search method in Fig 12.3 assuming that it is in a class with generic type `E`. The `E` represents any class which implements the `Comparable` interface.

   ```
   public class Search<E extends Comparable>
   ```

12.2 Sorting a list

12.2.1 Rationale for Sorting

In this section we discuss an important problem known as the *sorting* problem: given a list of values which can be compared[4] for order, arrange the list in ascending (or descending) order.

Once a list has been sorted, it can be searched quickly with the Binary Search algorithm. Sorting a long list can take a lot of time, but it is probably worth it if the list is to be searched for many different values.

[4]Comparing two values to determine which is larger, or whether they are equal

12.2.2 Selection Sort Algorithm

There are many algorithms which can solve the sorting problem. We examine one of the easiest to understand and implement in this section. It is called *selection sort*. We describe the algorithm informally below, where `size` represents the size of the list.:

- Scan the list from left to right (`ndx` = 0,1,2,3,...`size`-2)
- Find the position, `p`, of the smallest value in the list beginning at position `ndx`, i.e. search the sublist in positions [ndx..size-1] for the smallest value.
- Swap (i.e. exchange) the values at positions `p` and `ndx`.
- Increment `ndx` and repeat from step 2 until `ndx` = `size`-1.

Fig 12.4 shows how this algorithm is applied to a List of size 5. The list is shown as the value of `ndx` ranges from 0 through 3. When the value of `ndx` is 0, the position of the smallest value to the right (-1) is position 3. Thus the algorithm swaps positions 0 and 3. Note that when the value of `ndx` reaches 1, the position of the smallest value starting from position 1, is also 1. That means that the algorithm swaps position 1 with itself - an unnecessary, but harmless, operation.[5] Note also that for a list of size 5, there are only 4 iterations. After the first 4 iterations, the first 4 values are correctly placed, and the remaining value must be in its correct position.

A Java method to sort a given List using the selection sort algorithm is shown in Fig 12.5. It uses a private helper method, `posSmallest(int start)` to return the position of the smallest value beginning at the given start position.

.

12.2.3 Insertion Sort Algorithm

The *insertion sort* algorithm is similar to the selection sort algorithm, in that it also requires n-1 passes over the input list, for a list of size n. On each pass, p=1..n-1, the algorithm will move the value in position p to the left as many places as needed so that it is inserted in the correct position. To do this, values to the left of position p must be shifted to their righ-hand neighbors, in order to make room for the inserted value.[6] Fig 12.6 shows how the algorithm sorts a List of size 5.

When p=2, the value at position 2 is 19, which is then removed and inserted at position 2 (essentially a no-operation). On that iteration the list is not changed. As with selection sort, we note that after i iterations the first i values of the list are correctly sorted. If n is the size of the list, then after n-1 iterations,

[5]Students often suggest checking for this condition, to avoid an unnecessary swap; however we feel that for random data it will not save a significant amount of time.

[6]We could remove the value at position p, and insert it at the correct spot, but the remove operation has a 'hidden loop' which we prefer to avoid, for reasons of run-time efficiency, and also to facilitate adapting the algorithm to operate on an array rather than an ArrayList.

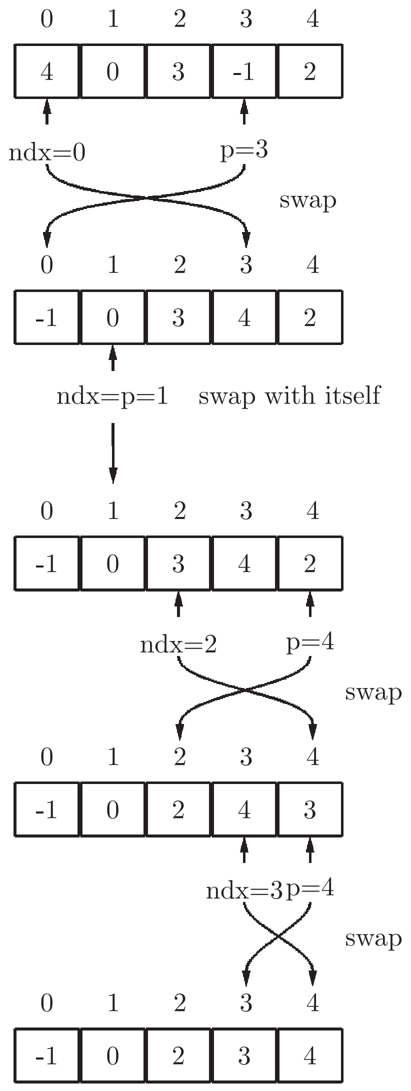


Figure 12.4: Selection sort algorithm, on a list of size 5

```
    /** Post: The values in the List will be arranged in ascending
              order
     */
    public void selectionSort (List <Integer> list)
    {   for (int i=0; i<list.size()-1; i++)
           swap (list, posSmallest (list, i), i);
    }

    /** @return the position of the smallest value,
     *   beginning at the given start position
     */
    private int posSmallest (List<Integer> list, int start)
    {   int smallestPos = start;
        for (int i=start+1; i<list.size(); i++)
            if (list.get(i) < list.get(smallestPos))
                smallestPos = i;
        return smallestPos;
    }

    /** Exchange values at positions i and j
     */
    private void swap (List <Integer> list, int i, int j)
    {   E temp;
        temp = list.get(i);
        list.set (i, list.get (j));
        list.set (j, temp);
    }

}
```

Figure 12.5: Java method implementing the selection sort algorithm, applied to a List of Integers

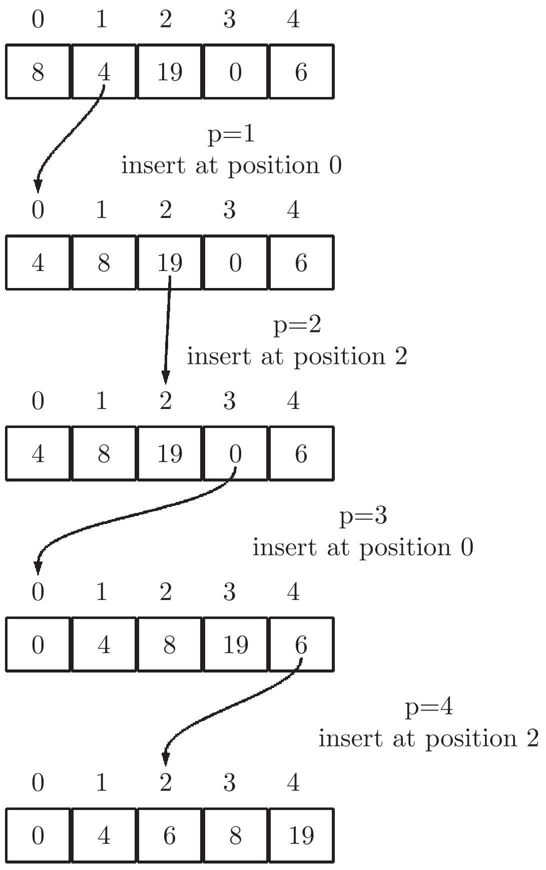


Figure 12.6: Insertion sort algorithm, on a list of size 5

```
/** Post: The values in the List will be arranged in ascending order
 */
    public void sort (List <Integer> list)
    {   int j, tmp;

        for (int p=1; p<list.size(); p++)
           {  tmp = list.get(p);
              for (j = p; j>0 && tmp < list.get(j-1); j--)
                list.set (j, list.get(j-1));         // shift right
              list.set(j, tmp);
           }
    }
```

Figure 12.7: Java method implementing the insertion sort algorithm, applied to a List of Integers

the first n-1 values are correctly sorted, so the last value must be in its correct spot; the loop repeats only n-1 times.

The code for the insertion sort algorithm is shown in Fig 12.7. Note that the inner `for` loop shifts values in the List to their right-hand neighbor:
`list.set (j, list.get(j-1));`
This allows insertion of the value from position p, stored in `tmp`, to be inserted at the correct position.

This example sorts a List of Integers, though it can be readily adapted to sort a List of any type which is Comparable.[7]

12.2.4 Merge Sort Algorithm

The Merge Sort algorithm makes use of one of the oldest techniques in computer science: that of merging two sorted lists. This was done decades ago when large datasets were stored on magnetic tape. If two tapes contained data in ascending order, they could be merged to a single third tape with a simple merge algorithm.[8]

12.2.4.1 Merge Algorithm

The *merge algorithm* takes 2 sorted lists as input; here we assume they are sorted in ascending order. The algorithm produces a sorted list from the values of the two given lists (refer to Fig 12.8 as we describe the algorithm). The two given lists are [3,5] and [2,3,6,7].

1. We compare the first value of the first list with the first value of the second list. The smaller value is added to the result list, and we move to the next

[7]With a generic type that extends Comparable, we can sort any List.

[8]Access to the data on a magnetic tape is sequential, similar to the access to data in a linked list.

value of that input list. In Fig 12.8 since $2 < 3$, we copy 2 to the result and move to the next position of the second list (value is 6).

2. Since $3 < 6$, we copy 3 to the result and move to the next position of the first list (value is 5).

3. Since $5 < 6$, we copy 5 to the result and move to the next position of the first list (reaching the end of the first list).

4. Since we have reached the end of the first list, we copy the remaining values of the second list (6,7) to the result.

The final result is the list [2,3,5,6,7] Fig 12.8 shows how the algorithm merges a sorted list of size 2 with a sorted list of size 3, to produce a sorted list of size 5.

A java method which implements the merge algorithm is shown in Fig 12.9. This method uses two Iterators, one for each input List. Thus it will run efficiently whether the input is an ArrayList or LinkedList. In the loop it chooses the smaller of the two values selected from list1 and list2, and adds it to the result List, after which it obtains the next value from that List. Note that a null value indicates that the end of the corresponding List has been reached. At that point the loop terminates, and the remaining values of the other list, if any, are copied to the result list.

This method uses a few helper methods, which are shown in Fig 12.10:

- `getNext(Iterator<Integer> it)`: Use the given iterator to obtain the next value from one of the input lists, or null if there is none.[9]

- `copyRemainingValues(int value, Iterator<Integer> it)`: Use the given iterator to copy all the remaining values to the result list.

12.2.4.2 Merge-in-Place Algorithm

Our goal is to expose a sorting algorthm which uses the merge algorithm that we have just seen. However, in order for it to be useful (and efficient) in our sorting algoirthm we need to modify the merge algorithm. We will consider the two lists to be merged as residing in the *same* list. We will need to provide the start position of the second list. For example, the lists [3,5] and [2,6,7] can be located in the same list as [3,5,2,6,7] with the start position of the second list at position 2. The result of our merge-in-place algorithm will be the list [2,3,5,6,7]. The Merge-in-Place is depicted with a diagram in Fig 12.11.

A Java method which performs the Merge-in-Place is shown in Fig 12.12. It has one parameter, the starting position of the second list to merged. The inner loop is used to shift values of the first list to the right to make room for the insertion of a value from the second list.

[9]This redesign of the next() method from the Iterator interface leads to a substantially cleaner solution.

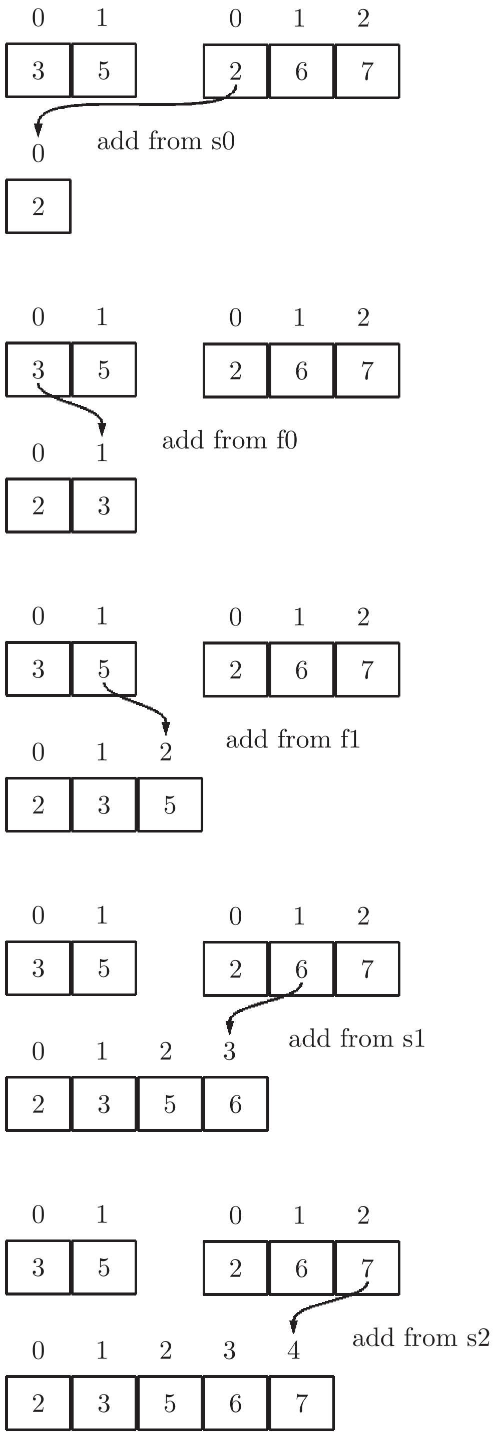


Figure 12.8: Merging two sorted lists into one sorted list. f = first list, s = second list.

```
    List<Integer> result = new ArrayList<Integer>();
    Iterator<Integer> it1, it2;
    Integer value1 = null, value2 = null;

 /**  @return a sorted ArrayList consisting of all values from
  *   the two given lists.
  *   @param first and second are both sorted in ascending order.
  */
public List<Integer> merge (List<Integer> first, List<Integer> second)
{   it1 = first.iterator();
    it2 = second.iterator();
    value1 = getNext(it1); value2 = getNext(it2);
    while (value1!=null && value2!=null)
     {  if (value1.compareTo(value2) < 0)    // add smaller value
            {  result.add(value1);           // to the result.
               value1 =getNext(it1);
            }
         else
            {   result.add(value2);
                value2 = getNext(it2);
            }
     }
   copyRemainingValues(value1,it1);     // one of these will
   copyRemainingValues(value2,it2);     // do nothing.
   return result;
 }
```

Figure 12.9: Method to merge two sorted lists into one sorted list.

```
   /** Copy the given value to the result list if not null, then copy
     *  the remaining values using the given iterator.
     */
    private void copyRemainingValues(Integer value, Iterator<Integer> it)
   {   while (value!=null)
            {  result.add(value );
               value  = getNext(it);
            }
    }

    /** @return the next value using the given Iterator, or
     *  null if there is none.
     */
    private Integer getNext(Iterator<Integer> it)
     {  if (it.hasNext())
              return it.next();
           return null;
     }
}
```

Figure 12.10: Helper methods for the method which merges two sorted lists into one sorted list (Fig 12.9).

12.2.4.3 MergeSort Algorithm

We now have all the tools we need to implement a sort algorithm known as MergeSort. Given an ArrayList of values which can be compared for larger-vs-smaller,[10] we wish to sort the values in ascending order. Here is the MergeSort algorithm:

1. If the size of the list is 0 or 1, terminate; the list is sorted.
2. If the size of the list is 2, swap the two values, if necessary, so that the smaller is to the left of the larger, and terminate.[11]
3. Find the midpoint of the list. We now have a left part (which includes the midpoint) and a right part (which excludes the midpoint).
4. Sort the left part, using the MergeSort algorithm.
5. Sort the right part, using the MergeSort algorithm.
6. Merge-in-place the two parts, using the Merge-in-Place algorithm described above.

[10]In Java, any class which implements the `Comparable` interface must have a `compareTo(Object)` method, which enables the client to compare not only for equality, but for ordering, i.e. smaller or larger.

[11]Step 2 is actually not needed; we include it to simplify the diagrams.

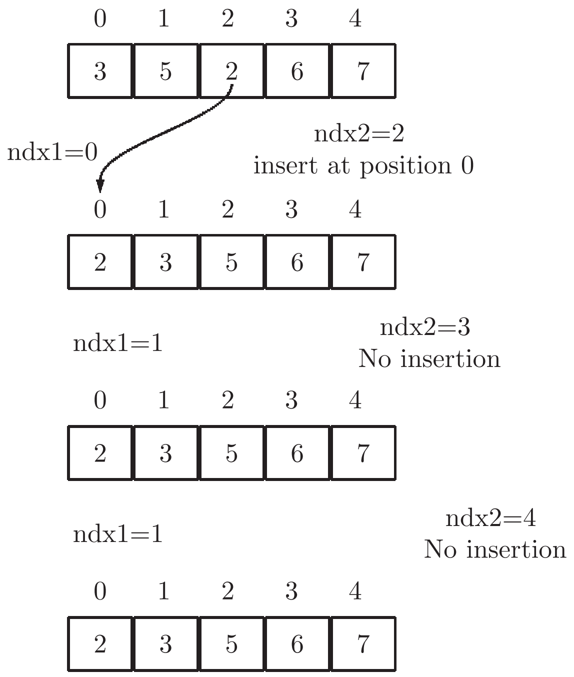


Figure 12.11: Merge in place. The two input lists are [3,5] and [2,6,7], contained in the same list. The second input list begins at position 2. Result is the merged list.

```
List<Integer> list;   //  list is a field
                      // initialized here ...
     /** Merge in place two lists stored in the same list.
      * @param start2 Position of first value in the second list.
      */
 public void mergeInPlace (int start2)
 {  int end = start2 - 1;              // end of first list
    int ndx = 0;                       // position in first list

     while (ndx<=end && end < list.size()-1)
       {   if (list.get(ndx) <=  list.get(start2))
                ndx++;
           else
             {  int value = list.get(start2);
                for (int i=start2; i>ndx; i--)
                     list.set(i, list.get(i-1));         // shift for insert
                list.set(ndx, value);          // insert value from second
                                               // list.
                ndx++;
                start2++;
                end++;
             }
       }
 }
```

Figure 12.12: Method to merge two sorted lists, both in one list, in place, using only one listt.

Notice that the description of the MergeSort algorithm involves a directive to use the MergeSort algorithm in steps 4 and 5. Thus it is a recursive algorithm (see chapter 3). This recursive method satisfies the two basic properties of recursion:

- There is a base case, which involves no recursive call (steps 1 and 2).
- In each recursive call, the size of the input is somehow reduced. In this case we are invoking the MergeSort algorithm on *half* of the given list (steps 4 and 5).

A diagram of this algorithm is shown in Fig 12.13. In that diagram we wish to sort the list [20,18,19,14]. To do that we first calculate the midpoint using the starting and ending positions. Thus the midpoint,
`mid = (0+3)/2 = 1`. The left half of the list is [20,18], and the right half is [19,14]. There is then a recursive call to MergeSort, using `start=0, end=1` to sort the left half. When that completes the left half is [18,20]. The same procedure is then applied to the right half, [19,14], which results in [14,19]. The complete list is now [18,20,14,19], which is merged using position 2 as the starting position of the second list. This merge results in [14,18,19,20], and the list is now sorted in ascending order.

A Java method for the MergeSort algorithm is shown in Fig 12.14. Note that since `mergeInPlace` is called from `msort`, and since `msort` is working on a *part* of the list, `mergeInPlace` will *also* need to know on what part of the list it is working. Therefore, `mergeInPlace` will need a second parameter, `end`, which is the index of the last value in the part of the list being considered.

This completes our discussion of the MergeSort algorithm. For long lists it is significantly faster than both SelectionSort and InsertionSort. However, a detailed analysis and explanation of the efficiency of MergeSort is beyond the scope of this book.

12.2.5 Exercises

1. Show which positions are swapped on each iteration of the main loop in the SelectionSort algorithm when sorting the list shown below:
   $[5, 4, 2, 4, 3, 1]$
2. Show a diagram similar to Fig 12.4 showing how the SelectionSort algorithm sorts the list shown below:
   $[5, 4, 2, 4, 3, 1]$
3. In the SelectionSort algorithm shown in Fig 12.5 there is a private helper method, `posSmallest(List,int)`. That method has a loop in which it compares two elements in the List being sorted. How many comparisons, total, are made if the size of the List, n, is:
   (a) n = 4
   (b) n = 5

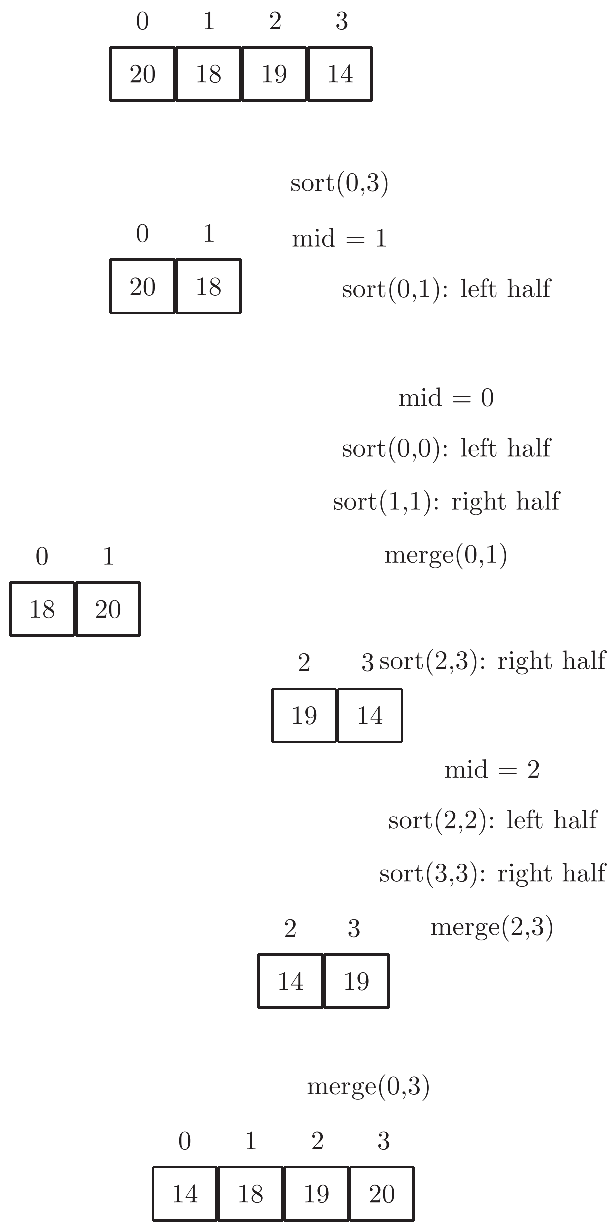


Figure 12.13: Merge sort algorithm, on a list of size 4

```
    // sort increasing, using mergeSort algorithm
    public void sort (List <E> list)
    {   this.list = list;
        msort (0, list.size()-1);
    }

   // Recursive helper method
   // Sort the portion of the list beginning at start
   // and ending at end.
    private void msort (int start, int end)
    {   if (end-start < 1)                    //  base case, size = 1
            return;
        if (end-start < 2)                    //  base case, size = 2
            if (list.get(start).compareTo(list.get(end)) > 0)
            {   swap (start,end);
                return;
            }
        int m = (start+end) / 2;              // midpoint
        msort(start,m);                       // sort left half
        msort(m+1,end);                       // sort right half
        mergeInPlace(start,end);              // merge the two halves
    }

private void mergeInPlace (int start, int end)
    {  int m = (start+end)/2;
       int ndx1 = start, ndx2 = m+1;

       while (ndx1<=m && m<end )
          {   if (list.get(ndx1).compareTo(list.get(ndx2)) <= 0)
                ndx1++;
              else
                {   E value = list.get(ndx2);
                    for (int i=ndx2; i>ndx1; i--)
                        list.set(i, list.get(i-1));         // shift for insert
                    list.set(ndx1, value);
                    ndx1++;
                    ndx2++;
                    m++;

                }
          }

     }
```

Figure 12.14: Method to sort a list using the MergeSort algorithm; mergeInPlace now requires two parameters.

   (c) n = 6
   (d) n = 1000
   (e) n

4. Show an implementation of the SelectionSort algorithm which operates on an array of ints rather than a List of Integers.

5. Show an implementation of the SelectionSort algorithm which operates on a List of Strings rather than a List of Integers.

6. If you are familiar with generic types in java, and typed classes, rewrite the sort method in Fig 12.5 assuming that it is in a class with generic type `E`. The `E` represents any class which implements the `Comparable` interface.
   `public class Sort<E extends Comparable>`

7. Refer to the InsertionSort algorithm. Given the List shown below, show the position at which each value at position p is inserted as p is incremented from 1 through 8:
   $[9, 12, 4, 6, 9, 2, 0, 8, 1]$

8. Show a diagram, similar to Fig 12.6, showing how the List given below is sorted by the InsertionSort algorithm:
   $[9, 12, 4, 6, 9, 2, 0, 8, 1]$

9. In the InsertionSort algorithm shown in Fig 12.7 there is a a call to the `set` method on the List being sorted:
   `list.set(j,list.get(j-1)).`
   in the inner loop. How many times is that `set` method called for a List (initially in descending order) of size:
   (a) 3
   (b) 4
   (c) 5
   (d) 1000
   (e) n

10. Show an implementation of the InsertionSort algorithm which operates on an array of ints rather than a List of Integers.

11. Show an implementation of the InsertionSort algorithm which operates on a List of Strings rather than a List of Integers.

12. If you are familiar with generic types in java, and typed classes, rewrite the sort method in Fig 12.7 assuming that it is in a class with generic type `E`. The `E` represents any class which implements the `Comparable` interface.
    `public class Sort<E extends Comparable>`

13. The private helper method, `msort` in the MergeSort algorithm shown in Fig 12.14 is a recursive method, with parameters `start` and `end`. Show the parameter values and the value computed for the local variable `m` on each call to msort for the List [9,2,8,3,0,7,5]

14. The private helper method, `msort` in the MergeSort algorithm shown in Fig 12.14 is a recursive method. How many times is that method called for a List of size:

    (a) 7

    (b) 15

    (c) 31

    (d) n

15. Show an implementation of the MergeSort algorithm which operates on an array of ints rather than a List of Integers.

16. Show an implementation of the MergeSort algorithm which operates on a List of Strings rather than a List of Integers.

17. If you are familiar with generic types in java, and typed classes, rewrite the sort method in Fig 12.14 assuming that it is in a class with generic type `E`. The `E` represents any class which implements the `Comparable` interface.
    ```
    public class Sort<E extends Comparable>
    ```

Glossary

! - Logical NOT operator

& - Bitwise AND operator

&& - Logical AND operator

| - Bitwise OR operator

|| - Logical OR operator

˜ - Bitwise NOT operator

abstract - Having no evident details; non-concrete

abstract class - A class which has one or more abstract methods, and cannot be instantiated

abstract data type - A collection of data with associated operations; ADT

abstract method - A method which has no body and must be defined in a subclass

abstraction - The process of separting ideas from specific instances of those ideas

access mode - A specification of which classes can refer to a particular class, method or variable: public, [default], protected, and private

accessor method - A method with the purpose of obtaining the value of a particular field

Action - A class which is capable of generating an event in a GUI

ActionListener - An interface for the handling of actions, such as a button click or menu selection, in a GUI

actionPerformed - A method in the ActionListener interface which handles actions

actual parameter - A parameter value to be passed to a called method

ADT - Abstract data type

algorithm - A well-defined sequence of steps to solve a given problem, which terminates with a correct solution

AND - A boolean operation which results in `true` only if both operands are `true`

ArrayList - A List which can efficiently get and set a value at a particular position, or index

ASCII - American Standard Code for Information Interchange

American Standard Code for Information Interchange - An 8-bit numeric code for each character; a subset of Unicode

API - Application Program Interface

application program interface - The information needed to use an entity, such as a class, package, or program; API

ArithmeicException - An Exception indicating that a non-valid arithmetic operation, such as a division by zero, has occurred at run time

array - A homogeneous collection of values which is mapped directly to the computer's main memory

assertion - A statement of the program's current state, at run time, for purposes of verification

assignment - The binding of a data value with a variable

awt - Abstract window toolkit; package used for graphics applications

BigNumber ADT - An ADT for whole numbers with unlimited magnitude

binary search - A search algorithm in which the number of values which need to be examined is proporitional to the the logarithm of the size of the collection being searched; a fast search algorithm

BinaryTree - A Tree in which each value has two children

bit - A binary 0 or 1; a binary digit

BlueJ - An IDE used to develop java software

BorderLayout - A LayoutManager with five regions: NORTH, SOUTH, EAST, WEST, and CENTER

Byte - Wrapper class for the primitive type `byte`

byte - A data type for whole numbers using 8-bit two's complement representation

byte - 8 bits

boolean - A data type with only two possible values: `true` and `false`

Boolean - Wrapper class for the primitive type `boolean`

catch - The process of handling a thrown Exception

central processing unit - That portion of the computer's hardware which is capable of performing calculations and making decisions; CPU

char - A data type for characters, such as those on the keyboard, using ASCII

Character - Wrapper class for the primitive type `char`

checked Exception - An Exception which must either be caught (in a try/catch statement) or declared to be throwbn

class - A template defining the composition of a data object

ClassCastException - An Exception indicating that a reference is not being casted correctly at run time, to a subclass or subtype

class method - A (static) method which applies to a class

class variable - A (static) variable shared by all objects of a class

client - Software needing services from other software

close (a file) - Discontinue use of, or relinquish access to, a file which has been opened

collection - An object consisting of a variable number of objects

command line - A user interface in which words are typed on a keyboard for input

comment - A programmer-supplied description, ignored by the compiler

compile-time error - An error in a program which is detected by the compiler

compiler - A program which translates a program written in a high-level language to an equivalent program in machine language

Component (graphics) - A graphical entity, or Container, which may be included in a Container

compound statement - A block of statements enclosed in curly braces

concrete - Having exposed implementation details; not abstract

ConcurrentModificationException - An Exception indicating that a value in a collection is being changed as an iteration through the collection is occurring

console application - A program in which the user interacts with a keyboard and text display

constant - A data value supplied by the programmer

Container - An awt class which enables storage of multiple components and/or containers

contentPane - A Container for the components of a JFrame

control structure - A programming construct enabling an altered flow of execution

constructor - A method used to initialize the fields of an object when it is created

CPU - Central processing unit

data file - Information stored on a secondary storage device, such as disk or flash memory

De Morgan's Laws - Boolean identities: The negation of a conjunction is the same as the disjunction of the negations; The negation of a disjunction is the same as the conjunction of the negations

declaration - A definition of a variable or method

[**default**] **access** - Access is permitted from any class in the same package

double - A data type for numbers using 64-bit floating point representation

Double - Wrapper class for the primitive type `double`

do while statement - An iteration structure which does not specify the number of times the loop body is to be executed; a post-test loop

duplicated code - Program code which is duplicated verbatim in several places in a program

dynamic method lookup - The process of determining which of several functions having the same name is being called, at run time

Event - A state caused by an external action such as mouse move or keyboard entry

Exception - A class which is used to manage errors or other unexpected occurrences at run time

expression - A variable, a constant, or an operation on two expressions

extends - Specification of a subclass relationship

extremum problem - The problem of finding a minimum or maximum value in a collection of values

field - A data value belonging to an object (non-static)

FileReader - A class in the java.io package enabling input from a data file

FileWriter - A class in the java.io package enabling output to a data file

final - Cannot be changed as the program executes

finally - A clause in a try/catch statement which allows the handling of an Exception when no specified catches apply

file - Data stored on a secondary storage device, such as disk or flash memory

float - A data type for numbers using 32-bit floating pointrepresentation

Float - Wrapper class for the primitive type `float`

floating point - A approximate data representation for numbers which need not be whole numbers, and which may be very large, or very close to 0

FlowLayout - A LayoutManager which arranges components from one row to the next in available space

for statement - An iteration structure defining the number of times the loop body is to be repeated

for-each statement - An iteration structure associated with a collection

formal parameter - A parameter in a method declaration

Frame - See JFrame

free format - A lexical property: white space is ignored by the compiler

generic type - A variable type to be filled in at compile time

get - The operation of obtaining a value from a collection or Map

graphical user interface - A user interface in which icons, and other images on a display, and mouse or touchpad are used to interact with a program

GridLayout - A LayoutManager in which components are arranged in rows and columns

GridWorld - Case study developed by the Educational Testing Service for the Computer Science Advanced Placement course

GUI - Graphical User Interface

has-a - Composition relationshp; field within a class

HashMap - An implementation of the Map interface using a HashTable

HashSet - A Set implemented with a HashTable

HashTable - A structure storing many values enabling quick access

heuristic - A series of steps which attempts to solve a given problem but which may terminate with an incorrect, or approximate, solution

high-level language - A language such as Java which enables humans to develop software; a programmming language

IDE - interactive development environment

if statement - A one-way selection structure

if-else statement - A two-way selection structure

implements - Specification of a (java) interface to be implemented

IndexOutOfBoundsException - An Exception indicating a non-valid position being accessed in a collection

inheritance - The establishment of a class hierarcy resulting from a subclass-superclass relationship

initialization - The assignment of a value to a variable when it is first declared

input - Data supplied by the user of a program

instance (of a class) - An object

instance method - A (non-static) method which applies to an object

instance variable - A variable owned by a particular object

interactive development environment - software used to edit, compile, and test programs being developed

instantiate - To create an object, or instance, of a particular class

int - A data type for whole numbers using 32-bit two's complement representation

Integer - Wrapper class for the primitive type `int`

interface - An adapting layer between two or more entities (see applicaton program interface, java interface, and user interface

IO - Input and output

IOException - An Exception involving input and/or output; checked

is-a relationship - Subclass relationship

iteration structure - A selection structure permitting repeated exection of a statement; a loop

Iterator - A class which enables access to each of the elements of a Collection; also enables selective removal of values from a Collection

java - A high level programming language developed by Sun Microsystems, later acquired by the Oracle corporation; a command to execute a compiled java program

javac - A command to compile a java program

java interface - Specification of operations on data

JFrame - A swing class defining an application's extent and components

KeyListener - A class which can handle keyboard events in a GUI

LayoutManager - Class in awt which automatically arranges the components in a Container

LinkedList - A List which can change size efficiently (insert and/or remove values)

List - A Collection in which order is maintained, and duplicate values are permitted

Listener - A class which is capable of reacting to an event in a GUI

long - A data type for whole numbers using 64-bit two's complement representation

Long - Wrapper class for the primitive type `long`

loop - An iteration structure

loop body - The statement to be executed repeatedly in a loop

machine language - The language of binary coded instructions which can executed by the CPU

main - Name of the starting method for a program

Map - An interface defining an ADT for quick access to values using associated (unique) keys

method - A programmer-defined operation to be performed on an object or class

method abstraction - The ability to see the important aspects of a method, without being exposed to its underlying detailed code, which may involve calls to other methods

MouseListner - A class which can listen for Mouse events, such as movement, click buttonDown, in a GUI

multiline comment - A comment initiated with /* and ended with */

multiple inheritance - The ability, or property, that a class may have more than one superclass

mutator method - A method with the purpose of changing the value of a particular field

MyFloat ADT - An ADT (designed for this textbook) which mimics floating point data types and is capable of doing arithmetic

nested loop - A loop defined to be entirely in the loop body of another loop

NOT - A boolean operation which results in the logical complement of its operand

NullPointerException - An Exception caused by the dereferencing of a null reference

object - Data values in memory with a predetermined structure; an instance of a class

object diagram - A drawn diagram depicting the fields of an object, and their current values

one-way selection - A selection structure with only one possible choice of execution paths; an `if` statement

open (a file) - Determine correct access to a file and prepare for input and/or output

operation - A calculation on one or two data values, producing a new data value, with possible side effects

OR - A boolean operation which results in `false` only if both operands are `false`

output - Data produced by a program for a user

overriding methods - The process of redefining a method from a superclass

package - A group of associated classes

parameter - A variable used to send information to a method

pixel - One of the small dots making up an image; a picture element

post-test loop - An iteration structure in which the loop body is executed once before the termination condition is tested

polymorphism - The capability of exhibiting different behaviors at run time, generally enabled by inheritance

pre-test loop - An iteration structure in which the termination condition is tested before the first execution of the loop body

primitive type - A data type included in the java programming language

program - A sequence of binary coded instructions in the computer's memory

programming language - A language such as Java which enables humans to develop software; a high-level language

protected - Access is permitted from any subclass or any class in the same package

private - Access is not permitted from any other class

program - A sequence of binary coded instructions stored in the computer's memory

public - Access is permitted from any class

public static void main - Specification of the starting method for a program

put - The operation of adding a value to a HashTable or Map

Rational ADT - An ADT which can do arithmetic with non-whole numbers

recursive method - A method which calls itself

reference - The memory location of a data object

reference type - A data type defined by a class

return type - The type of data to be returned by a method

return statement - A statement intended to terminate the execution of a method, with a possible value to be sent to the calling method

run time - The execution of a program, as opposed to the compilation

run-time error - An error in a machine language error, detected when the program is executing

RunTimeException - An Exception which the programmer may ignore; unchecked

Scanner - A class in the java.util package used for input, pattern recognition, etc.

scope (of a variable) - The range of statements over which a variable has meaning

search - The problem of finding a given target value in a collection of values

selection structure - A programming construct enabling a program to take one of a few possible execution paths

sequential search - A search algorithm which examines all elements of a collection until the desired value is found, or determined not to be in the collection

server - Software providing data or computation for other software

Set - A Collection in which ordering of the values is not required, and in which there are no duplicates

set - The operaion of changing a value in a collection

short - A data type for whole numbers using 16-bit two's complement representation

Short - Wrapper class for the primitive type `short`

short circuit evaluation - An optimization of a selection structure resulting from the evaluation of a single operand

side effect - A change in a program's state, or output, resulting from an operation

signature - The part of a method defining the access mode, return type, method name, and parameter list

single line comment - A comment initiated with // and ended at line-end

sorting - The process of arranging the values in a collection in ascending (or descending) order

statement - An assignment operation, or method call, followed by a semicolon, a selection statement, a loop statement, or a compound statement

static - Describing a field or method which applies to a class, not an object

static field - A data value belonging to a class; a class variable

static method - A method invoked on a class; a class method

stderr - Standard error file; defaults to the user's console display

stdin - Standard input file; defaults to the user's keyboard

stdout - Standard output file; defaults to the user's console display

String - A class in the java.lang package containing operations on strings

string - Data consisting of a sequence of characters

swing - An updated package for graphics; in javax

TextListener - A class which can handle text entry from the keyboard in a GUI

throw - A statement which interrupts execution to indicate that an Exception has occurred at run time

toString() - A standard method used to convert a data object to a String representation

Tree - A storage structure in which each value is associated with other values, called the 'children'

TreeMap - An implementation of the Map interface using a BinaryTree

TreeSet - A Set implemented with a BinaryTree, in which natural ordering of the values is maintained

try - A statement which enables a thrown Exception to be handled at run time

type - classification of data, such as int, char, String, ...

two's complement - A binary representation system for negative, as well as positive, whole numbers

two-way selection - A selection structure with a choice of two possible execution paths; an `if - else` statement

type conversion - The transformation of a data value to a different type

user interface - Hardware and/or software used for human interaction with a device or program

variable - A name representing a memory storage location for a primitive value or a reference to a data object

visibility - Accessibility in a class

void method - No value is to be returned

while statement - An iteratioon structure which does not specify the number of times the loop body is to be executed; a pre-test loop

wrapper class - A predefined class in the java.lang package whose objects store only the value of a particular primitive type

Index