Theoretical aspects of the syntax and semantics of the Java language.

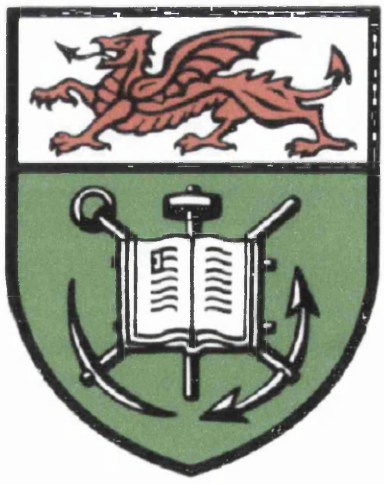
Morris, David Edward Ronald

Theoretical Aspects of the Syntax and Semantics of the Java Language

by

David Edward Ronald Morris

Master of Philosophy Thesis submitted to the
University of Wales, Swansea.



Department of Computer Science,
University of Wales, Swansea.

August, 2006.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

This thesis investigates two theoretical aspects of the formal definition of programming languages, using case studies in Java.

First, we define modular grammars which can be used to decompose large grammars. Modular grammars allow the modular definition of formal languages. They provide concepts of component and architecture for grammars and languages. We show that this modular method can be used to define a modern practical language like Java.

Second, we describe recent general work on the definition of interfaces and interface definition languages (IDLs). In Rees, Stephenson and Tucker [2003], there is an analysis of the idea of interfaces and an algebraic model of a general IDL. We apply these ideas to analyzing aspects of interfaces in Java.

The thesis is comprised of five chapters together with an appendix. Chapter 1 consists of an introduction to the thesis. The second chapter reports on object-oriented programming and the Java programming language with particular emphasis on a mathematical theory of its definition. Chapter 3 deals with a modular decomposition of Java syntax and grammars.

In Chapter 4, we expound a theory of the modular definitions of interfaces within any programming language. One important feature of the general account is the process of flattening the hierarchical structure produced by modularity.

In Chapter 5, we attempt to implement the results of research into the Interface Definition Language discussed in Chapter 4. We define *'Little Java'*, a subset of the programming language Java, and endeavour to provide a series of translations from *'Little Java'* to an abstract object-oriented interface definition language OO-IDL and thence to an interface definition language AS-IDL for abstract data types.

In the Appendix, we review the history of the Java language.

Declaration

This work has not, previously, been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

The thesis is my own work. Other sources have been acknowledged together with explicit footnotes and citations giving explicit reference.

Signed:

Date:

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Explicit references have been made in acknowledgement of the many sources of contributive academic writings. A bibliography has been appended.

Signed:

Date:

Statement 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan. The title and summary is to be made available to any organization or institution.

Signed:

Date:

Acknowledgements

In September 1930, for the first time, Adolf Hitler's Nazi regime was recognised as a political party by the people of Germany. One month later, with no apparent connection, historically or politically, I was born. Perhaps the timing was unfortunate but I subsequently spent the duration of the Second World War in our local Grammar School and in 1946 was eager to venture forth into the brave new world of a post war Britain. I entered into the field of telecommunications and spent six years learning skills, which I am glad to say, have never left me. After qualifying as an engineer, I suddenly became aware that my future was to be found elsewhere, in teaching. I applied for a place at a Teacher Training College and, after two years, moved on to complete a Diploma year at Cardiff College of Physical Education.

It may seem presumptuous but the next thirty years were the happiest years of my life. I was privileged to teach children and to learn from them in the process. On returning to Wales, after a teaching spell in England and with two young children, my wife and I decided that we would build our own home in Baglan. Four and a half years later we moved into our new but slightly unfinished home and we are still here, forty three years on. Our family, now four, have flown the nest and left us with a large and beautiful house with a large and not so beautiful Council Tax.

In 1978, at the age of forty eight, I was seconded on a two year Computer Science Diploma course at Swansea University. Having successfully completed the Diploma, I returned to Cwrt Sart Comprehensive School as Head of Computer Studies. It should be noted that Swansea University Computer Science Department was the first educational body to train teachers in this field and it meant, most significantly, that we were all pioneers of our time.

Due to ill health and, on reflection after a good innings, I retired at the grand old age of fifty four. I wondered where the years had flown. With all that time on my hands my wife and I started a computer business specialising in bespoke software and after ten years, we retired again.

I would like to mention two people who, probably, were responsible for me embarking on the latest adventure of my life. Mr. and Mrs. Edward Moseley, both graduates, convinced me that my idea of applying to the University of Wales Swansea to undertake a Computer Science course was worthwhile. I would like to thank them for their gentle persuasion and friendly advice.

I applied to the College in August 1997 and, some months later, I was called to an interview where I was told I could commence tuition on a higher degree course in the September. During that time I was fortunate to have had the opportunity to read certain topics prior to starting the syllabus in January 1998. In addition, there was time for me to acquaint myself, once again, with the Unix operating system and the new network in the Computer Science Laboratory. I say new because the last time I was at the University was some twenty years earlier and equipment and operating procedures had been updated since then.

My discussions with Dr Chen and Dr Stephenson were most enlightening and I was aware that the course content was something that would have to be decided at some time in the near future. It was with this in mind that I commenced my studies in the Spring Term of 1998.

Unfortunately, I was to find my course of study at the University interrupted by a serious illness. What two doctors thought was a digestive problem proved to be angina and I was, eventually, admitted to hospital as an emergency patient. This was towards the end of February and within the space of two weeks I was to undergo major heart 'bypass' surgery. As a result, I was unable to do any kind of work for the next four months and, in consultation with Dr. Chen, it was decided that I restart the course again in spring, 1999.

I had discussed the course material with Dr. K. Stephenson for the coming year and the timetable had been formulated on the basis of projected academic requirements and relevant lecture material deemed most suitable.

In my time of study at the University of Wales Swansea I have shared the company of students, graduates, post graduates, lecturers, professors and staff. They have all made

some contribution, knowingly or unknowingly, to my life. Each and everyone has shown a willingness and desire to help and nurture the academic needs of a rather senior fellow student; my thanks to you all.

Of course, every student has a supervisor, and I must note a gesture of special appreciation to two academics who took on the task of guiding me through the labyrinths of the Theory of Programming Languages.

- Dr. K. Stephenson whose patience and understanding was immeasurable and,

- Professor J. V. Tucker who took up the reins when Dr. Stephenson moved to a new position. I am grateful to him for his invaluable help, advice and consideration, in particular, his support and encouragement during the correction, preparation and eventual completion of the thesis.

I would, also, pay tribute to the administrative and technical staff for their guidance on matters of overwhelming importance such as keeping all systems in working order, including my own. A special mention for the secretaries, Mrs. Susan Fenn and Mrs. Jill Edwards. Jill, as secretary to Professor John Tucker, has helped me enormously and has shown great patience and understanding in maintaining that communicative link between myself and my supervisor, J.V.T.

Professor Min Chen and Professor Faron Moller, over the years, have been supportive on matters of University protocol and procedure and I thank them. My thanks also, to Dr. Phil Grant and Dr. John Sharpe, the sole survivors of the earlier Computer Science Department teaching staff, who were my tutors on the first West Glamorgan Teachers' Diploma Course in 1979-81.

My fellow students, throughout my time at Swansea University, have always been helpful and understanding and I regard them as very dear friends. I am very aware of their contribution to my own studies, to my well being and to my family. In particular, I would like to single out Alfie Abdul-Rahman, for her inspiration and help, and Simon Walton for his unfailing assistance.

I would like to thank my examiners, Dr. Grant Malcolm and Professor Peter Mosses, for their invaluable contribution to this thesis. The remarks and suggestions proffered by them, on reading my submitted thesis, were supportive and constructive and I have endeavoured to complete their list of suggestions and corrections as accurately as possible. I also thank Dr. Neal Harman for useful material on Java.

My final acknowledgement is to my wife Bette for her care and support throughout my years at The University of Wales Swansea. It is significant to say that her encouragement and endeavor has been unwavering despite having to endure a crippling illness. I am deeply appreciative of her love and loyalty.

So, in my seventy sixth year, I submit a thesis. All those people I have mentioned have contributed to this event and I humbly thank them.

Contents

Table of Figures

Chapter 1

Introduction

"It must be remembered that there is nothing more difficult to plan, more doubtful of success, nor more dangerous to manage, than the creation of a new system. For the initiator has the enmity of all who would profit the preservation of the old institutions and merely lukewarm defenders of those who would gain by the new ones."

Machiavelli.

In general, the thesis investigates some theoretical aspects of the formal definition, or specification, of programming languages. We consider aspects of syntax and semantics and focus on the object-oriented language Java as a case study of the methods. In software terms the overall theme is modularity and hierarchical structure, and the architecture of languages and programmes it determines. Theoretically, the overall theme is the radical simplification of programming languages in order to use simple theoretical tools.

For syntax we will consider modular grammars and Backus–Naur Forms and use them to give a decomposition of Java syntax. More specifically, we examine examples and case studies in Java together with a modular construction of a subset of the Java Language (Version 1.1).

For semantics we will consider the mathematical modelling of the concept of interfaces and its semantics. We abstract from a programming language the definition of an Interface Definition Language. (Rees *et al* [2003]). We define sets of interfaces, termed repositories, and examine their structure to develop a notion of system architecture and give a formal specification.

The idea of modular grammars for the specification of languages has been proposed in Stephenson and Tucker [2006]. There, it is used to define small to medium size

examples of languages. We will develop modular grammars and apply them to a large real world programming language to see if the idea scales up.

The general idea of interface has been analysed in Rees, Stephenson and Tucker [2003]. There, interfaces have modularity through being able to extend or import interfaces when constructing a new interface. The paper gives algebraic specifications of libraries of interfaces and a process called flattening which assembles an equivalent interface by substituting the interface components. We will give an account of this theory and investigate its application to a large real world programming language to see what the ideas reveal and to explore their scope and limits.

Thus, the thesis investigates some theoretical concepts and tools - modular grammars, abstract interfaces, flattening, etc – that have been used on small illustrative examples and develops them in order to apply them to large real world languages.

In Chapter 2, we examine the main ideas about object-oriented language and identify the concepts of class, inheritance and library. We compare languages such as Eiffel, Corba, Java and Remote Method Invocation (RMI). We then move on to define the syntax of the Java Language with further examples, illustrations and comparisons.

In Chapter 3, we discuss the decomposition of the Java language and make a detailed analysis of the structures that constitute the language. We describe modular grammars and the modular decomposition of the language. We illustrate a modular decomposition of the syntax of Java. The theories used for this structured approach are based on modular, context free, grammars in Backus–Naur Form.

In Chapter 4, we define the general mathematical modelling of any interface definition language as given in Rees, Stephenson and Tucker [2003]. Interfaces are elements of sets called repositories and are given an algebraic structure to define system architecture. We discuss flattening transformations that eliminate, or reduce, the hierarchical structure, by a form of structural induction, and illustrate this with algebraic definitions. We show how a dependency trail represents the data dependencies of an interface and consider the properties of these interfaces. We extend the original interface by adding the import definition. The imported interface dependency is made

redundant when the import is removed by the transformation technique known as flattening.

In Chapter 5, we abstract from the concrete syntax of Java and we define a subset '*Little Java*' and endeavour to provide an adequate and appropriate interface definition language to analyse and support its semantic modelling. We have simplified the language and made it as sound as possible without compromising the full Java language. In order to do this we employ certain restrictions and omit certain advanced features (for example: overloading, threads and exceptions).

In the Appendix we summarise the history of Java.

Chapter 2

Object-oriented Languages and Java

In this chapter we discuss some of the basic ideas about object-oriented programming as they appear in Java. The thesis demonstrates certain features of software architecture. What is software architecture? The following examples give selected background information on the topic. We include some of the definitions and associated discussions of software architecture that can be found on the CMU Software Engineering Institute website.

· **Dahl and Nygaard [1966]**

Ole-Johan Dahl and Kristen Nygaard are responsible for the origin, design and development of the programming language Simula at the Norwegian Computing Centre in Oslo between 1962 and 1967. Although the language never proved to be popular it has been highly influential on modern programming methodology. Simula introduced important object-oriented programming concepts such as objects, classes, inheritance and dynamic binding.

· **Garlan and Shaw [1996]**

Mary Shaw and David Garlan suggest that software architecture is a level of design concerned with issues... "*beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organisation and global control structure; protocols for communication, synchronisation, and data access; assignment of functionality to design elements; distribution; composition of design elements; scaling and performance; and selection among design alternatives.*"

· **Bass, *et al* [1998]**

Writing about a method to evaluate architectures with respect to the quality attributes they instil in a system. Bass and his colleagues write that... "*The architectural design of*

a system can be described from (at least) three perspectives -- functional partitioning of its domain of interest, its structure, and the allocation of domain function to that structure."

· Hayes-Roth [1994]

Writing for the ARPA Domain-Specific Software Architecture (DSSA) program, Hayes-Roth says *"that software architecture is ... an abstract system specification consisting primarily of functional components described in terms of their behaviours and interfaces and component interconnections."*

· Garlan and Perry [1995]

David Garlan and Dewayne Perry have adopted the following definition for their guest editorial to the April 1995 IEEE Transactions on Software Engineering devoted to software architecture: *"The structure of the components of a program / system, their inter-relationships, and principles and guidelines govern their design and evolution over a period of time."*

· Booch, Rumbaugh, and Jacobson [1999]

"An architecture is the set of significant decisions about the organisation of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements, the composition of these structural and behavioural elements into progressively larger subsystems, and the architectural style that guides this organisation---these elements and their interfaces, their collaborations, and their composition."

In the range of expressions utilised by the idea of software architecture some terms, or ideas, seem essential: there are *building blocks* that have *interfaces* and these give rise to a *modular* or *hierarchical structure*. These ideas appear as fundamental motivation for object-oriented programming in which *classes* and *objects* are building blocks and *inheritance* gives the modular structure.

2.1 Objects

Objects are software models of the physical and conceptual things we find in the universe around us. Hardware, software, documents, human beings, and even concepts are all examples of objects. For purposes of modelling, an automotive engineer would see tyres, doors, engines, top speed, and the current fuel level as objects. Atoms, molecules, volumes, and temperatures would be objects a chemist might consider in order to create an object-oriented simulation of a chemical reaction. Additionally, a software engineer would consider stacks, queues, windows, and check boxes as objects.

Objects have state. The state of an object is the condition of the object, or a set of circumstances describing the object. For example, the state of a bank account object would include the current balance, the state of a clock object would be the current time, and the state of an electric light bulb would be "on" or "off." For complex objects like an automobile, a complete description of the state would be complex. We use objects to model real world or imagined situations and we typically restrict the possible states of the objects to those that are relevant to our models.

We tend to think of objects as being strictly static. That is, the state of an object will not change unless something outside of the object requests the object to change its state. Certain objects are passive (static). A list of names does not spontaneously add new names to itself, nor would we expect it to spontaneously delete names from itself. Edward Berard wrote an abstract on the testing of object-oriented software which included the evaluation of such factors as encapsulation and inheritance. (See Berard [2000]).

However, it is possible for some objects to change their own state. If an object is capable of spontaneously changing its own state, we refer to it as an active object. Clocks and timers are common examples of active objects.

An algorithm for accomplishing an operation is referred to as a **method**. Unlike operations, methods are not part of the public interface for an object. Rather, methods are hidden on the inside of an object. So, while users of bank account objects would

know that they could make a deposit into a bank account, they would be unaware of the details as to how that deposit actually got credited to that particular bank account.

Systems of interacting objects, on the other hand, resemble applications. For example, suppose that we wanted to construct an object-oriented application that controlled the lifts in a particular building. We would assemble lifts, buttons, lamps, panels, control units and other objects into a working application that would monitor the lifts. Such an application would not be viewed as a library, but as a highly cohesive whole. The lift controller application is a system of interacting objects.

2.2 Object Oriented Programming

In this section we analyse three interpretations of object-oriented development and record their subsequent effect on object-oriented software. The basis of object-oriented development is abstractly defined object types called classes; it is difficult to conceive of a class without an abstract specification. (See Sebasta [1989]).

2.2.1 What is Object Orientation?

Object-Orientation is a paradigm for creating software systems using objects. Objects are tangible and conceptual things we find in the real world. Using object-oriented techniques, the code is broken into modular, reusable chunks called classes. Classes are the "blueprint" for creating instances of objects. These classes can be used throughout an application repeatedly. (See Flanagan [1996]).

2.2.2 The Benefits of Object-oriented Programming?

Object-oriented programming emphasises creating reusable, robust software in a way that is easy to understand. By relating programming to the real world, it becomes much easier to use. Walden and Nerson [1994], list some of the characteristics of object-oriented programming and states that the programmes are,

- **Reusable** – and speeds up modular development,
- **Robust** – and increases quality,
- **Simple** – and easy to maintain,
- **Flexible** – and easy to modify.

2.3 Concept Overview

There are many concepts involved with object-oriented programming. The following are the basic features we encounter:

- **Classes** – A generic blueprint used to create similar objects. We can have a *car* class defining common properties of cars. All of these vehicles would move, brake, reverse, use some form of fuel, etc. Specific instances of these objects may be different in design, having different wheels, engines, doors, colour, but they would all share the same basic characteristics.

- **Encapsulation** – Encapsulation is a principle of object-orientation that provides common interfaces, protects the state of an object and hides its implementation details. This common interface between objects with different internal representation permits interchangeability. "*Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics*". (See Booch [1999]).

- **Inheritance** – Allows reuse of code by building on existing classes. Using inheritance you can create a base class and then extend this class by creating a subclass that has all of the inherent properties of its base class.

- **Polymorphism** – Allows objects to assume many forms. We construct a class ***motor vehicle*** with an engine, a body, steering, brakes, transmission and four wheels. Two instances of this class, ***car*** class and ***lorry*** class, would inherit the properties and behaviours of the parent class. Together with these inherited capabilities, a car could have all those different qualities. We would associate these as model features, but would still recognize it as a car. A lorry could have more than four wheels, an articulated lorry more again. Acceleration and braking

are behavioural patterns and would have a contributory effect on the inherent values of the object. The car and the lorry have the ability to move, and be driven, but attributes will determine behaviour, in many ways.

- **Library** – A collection of classes that can be used to make new classes via inheritance. Such new classes can be added to the library.

In this thesis we use the language of Java and its class structure as an example of an object-oriented language. Shortly, we mention other languages that have a similar structure. It is explained, perhaps, more eloquently by Booch:

"The logical model (i.e., the problem domain) is represented in the class and object structure. In the class diagram one builds up the architecture, or the static model. To deal with complex diagrams, the notation allows class categories to group classes into name spaces, each category being itself a class diagram. The module and process architecture deals with the physical allocation of classes and objects to modules, and with processors, devices and communication connections between them, in few words it describes the concrete hardware with respect to the software components of a system".

Booch [1992].

Class diagrams are widely used to describe the types of objects in a system and their relationships. For example the diagram would display:

(i) the class name,

(ii) the class attributes, and

(iii) the class operations.

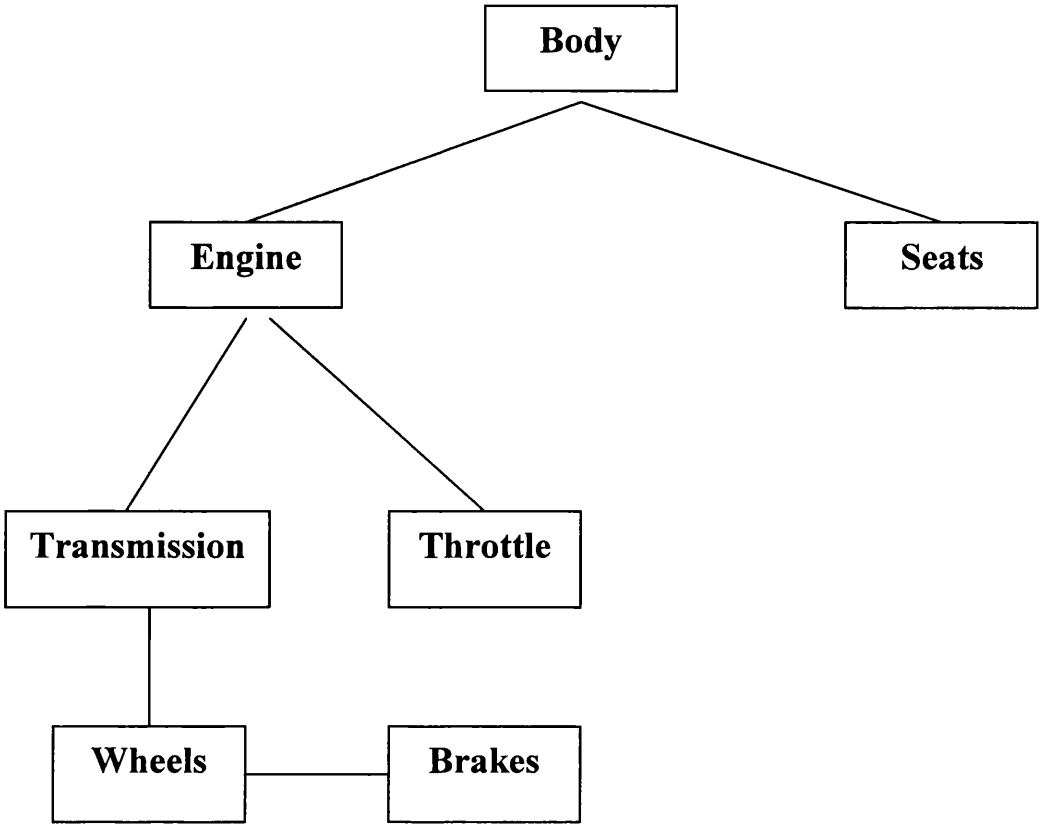


Figure 2.1 Illustration of fundamental objects in a motor vehicle.

A useful analogy set by Kenneth Litwak [2000] would be: *'In order for me to exist, my father and mother had to exist first. In order for my mother to exist, her mother and father had to exist. In order for her parents to exist my grandparents, on my mother's side, had to exist, and so forth.'*

In a similar way, for a Frame object to exist, its parent object, Window, has to exist. For a Window object to exist, its parent class Container, has to be instantiated, and so forth.

The object diagram Figure 2.1, shows how certain objects interact via the exchange of messages. This interaction can be defined as their relationship.

We look at some languages.

2.4 Smalltalk, Eiffel and Python

Smalltalk. The language was developed in the seventies by Alan Kay *et al.* He was the leader of a group of pioneers who were the first to research and then implement object-oriented software. Simula, originally a subset of Algol60, was the keystone of this new language and played an important part in its development. In Smalltalk, everything is an object and belongs to a class. Strings, integers, booleans, etc. are all represented as objects. Java adopted many of these concepts having strings, integers, booleans represented as primitives and objects.

The execution of code is sequential within objects and *message sending* between objects. Any message can be sent to any object; the class of the receiver object determines whether this message is appropriate and accordingly, determines the manner in which it should be processed. Smalltalk also made use of other advanced ideas such as garbage collection which is described as an automatic memory reallocation process. (See Smalltalk [1970 – 2005]).

Smalltalk programs are compiled to bytecode and run by a virtual machine (VM). They are then executable on any hardware platform that is compatible with a VM. The concept was adopted by Java a decade later. The Java Virtual Machine (JVM) is discussed in Chapter 2.6.

Eiffel. Bertrand Meyer, (See Meyer [1997]) is responsible for the inception and development of the Eiffel language and he has been actively concerned with that development since 1985. Eiffel is another object-oriented programming language with the emphasis on it being robust software. Again its syntax is keyword-oriented and is based on the concepts laid down by such languages as ALGOL and Pascal. Eiffel is strongly typed, with automatic memory management (another implementation of garbage collection).

Eiffel is a small language, similar in size to Pascal, and is closely allied to the original concepts of object-oriented software engineering set out in the earlier language

Smalltalk. Numerous universities around the world were quick to realise the potential of the language and adopted it as their primary teaching language.

Python. The Python language has an extensive support system for object oriented programming. It supports inheritance, multiple inheritance, polymorphism. Python treats classes, functions, numbers and modules as objects. It has limited support for private variables and regards the programmer as the controlling influence with regard to safe programming.

Exception handling is supports extensively by Python, it is capable of testing for error conditions and other exception events in a program and it also has the capability to trap the exception whenever that may occur. Exceptions can also be used to instigate the cessation of iterative and deeply-nested message-handling code.

Again, objects may be used as instances of abstract type and can be termed as interfaces possessing data structures with unique identities. The Object (class) would be the basic unit and collectively, they would comprise a program that would allow instances of these units to interface with a class and the data within that class. This ability to communicate independently with other interfaces and with any inherited type is common to the object-oriented languages listed above.

This language combines remarkable power with very clear syntax.. It has modules, classes, exceptions, high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries. The standard library is one of Python's greatest strengths. The library modules can be used with custom modules written in other languages such as C or Python. The greater part of this library is cross-platform compatible and Python programmes are able to run on Unix, Windows, Macintosh, and other platforms without alteration. A comprehensive history is available on the Wikipedia website together with a summary of the language. (See Python Programming Language [2005]).

2.5 Corba

CORBA stands for *Common Object Request Broker Architecture.* It is a high level tool for distributing programming with components and has many interesting ideas.

CORBA standards provide the proven, interoperable infrastructure to Java 2 Platform, Enterprise Edition. It also provides for high-level language bindings, static and dynamic method invocations, local and remote transparency, built-in security and transactions, polymorphic messaging, and many more programming attributes. At the base of CORBA is the ORB (Object Request Broker). A common principle adopted by many computer designers is to allow many different devices to communicate with one and other on the same bus. In software terms, the ORB is the middleware that allows all CORBA objects to communicate with other CORBA objects that run on any number of client and/or server machines. The ORB receives the call from a client object, finds the object that can handle the request, passes any parameters, invokes the method and returns a result. (See O.M.G. [2003]). Here are some of its key features:

- **The Interface Definition Language** – For most developers the starting point for a CORBA application is the Interface Definition Language (IDL). The IDL defines the interfaces that a client object will call and the server object will implement. CORBA IDL is a declarative language supporting C++ style syntax for keywords, preprocessor commands, pragmas, constants, types and methods. *Pragmas* are special compiler commands that control certain features of a C-compiler and are compiler specific. An IDL to Java compiler is used to convert the IDL into language specific client stubs and server implementation skeletons. The latter may be defined as constructs (pieces of code) where implementations and analyses can be shared between instances. Such constructs are *skeletons*, in that they have structure but lack detail.

- **Object** – This is a CORBA programming entity that consists of an *identity*, an *interface*, and an *implementation*, which is known as a *servant*.

- **Servant** – This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk (See Brachla & Griswold [1993]).

- **Client** – This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., `obj-> op(args)`. The remaining components, in Figure 2.3, help to support this level of transparency.

- **Object Request Broker (ORB)** – The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

- **ORB Interface** – An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

- **CORBA IDL stubs and skeletons** – CORBA IDL stubs and skeletons serve as the 'glue' between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target

programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

- **Dynamic Invocation Interface (DII)** – This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive operations) and *oneway* (send-only) calls.

- **Dynamic Skeleton Interface (DSI)** – This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

- **Object Adapter** – This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OO-DB object adapters for persistence and library object adapters for non-remote objects).

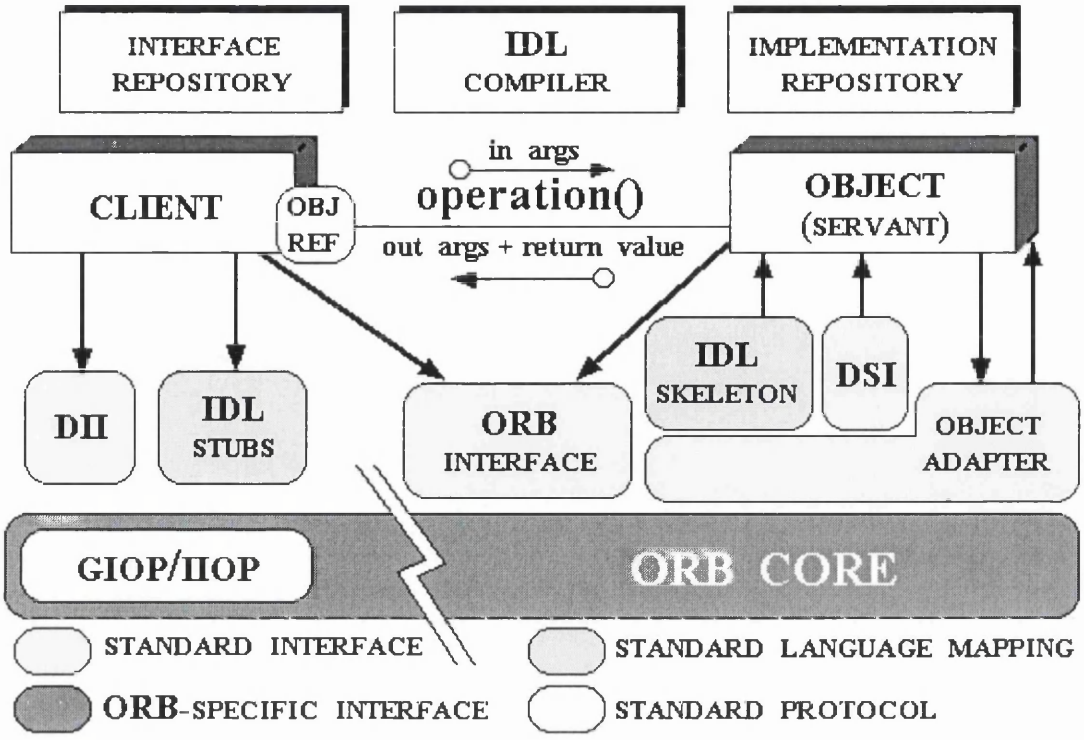


Figure 2.2 CORBA ORB Architecture. (Image from Borland).

2.6 The Java Platform

The Java platform consists of the Java application programming interfaces (APIs) and the Java Virtual Machine (JVM). Java API's are libraries of compiled code used in a program. They are available as ready-made programming templates that are customized in a functional manner by the programmer thus saving a great deal of programming time.

Java is a language developed by Sun Microsystems, Inc. The present versions of the Java Development Kit (JDK) range from JDK1: 1.0, 1.02, 1.1, 1.3, 1.4 and 1.5. In February 1997 Java Development Kit 1.1 was launched. In order to comply with available software at the university and adjunct packages such as Netscape (Supporting the version 1.1), Java version 1.1 may be designated as our program prototype for research purposes. (See Gosling [1996]).

Java **Application Programming Interfaces** or **APIs.** The Java programming language is versatile and adept and is regarded, more widely, as a language tool for the creation of *applets* for the World Wide Web or Internet. An applet is a mini-programming application that is able to operate within a Web Page. (See Horstmann [1996]).

Java's ability to execute code on a remote and secure basis is a major advantage, and this together with its network based user applications, graphical interfaces, multithreading and exception handling, make it the ideal language for the interactive requirements of internet programming. The word Java, when used in this thesis, is a direct reference to the Sun Microsystems, Inc., system. We will discuss the program constructs of Java on several occasions in this thesis. Java software objects are modelled on real-world objects. They, also, have state and behaviour. A software object maintains its state in one or more variables. A variable is an item of data named by an identifier. A software object implements its behaviour with methods. A method is a function (subroutine) associated with an object. One special feature is its virtual machine

A **Java Virtual Machine** or **JVM** is a virtual machine that runs Java *byte code.* This code is most often generated by Java compilers, although the JVM has also been targeted by compilers of other languages. The JVM is a vital component of the Java platform. The availability of JVMs on almost all types of hardware and software platforms enables Java to function as a platform in its own right.

Programmes intended to run on a JVM must be compiled into a standardized portable binary format called *bytecode.* Java was designed to allow these compiled application programmes to be run on any platform without having to be rewritten or recompiled. Java's virtual machine makes this possible.

A program may consist of many classes, in which case, every class will be in a different file. For easier distribution of large programmes, multiple class files may be packaged together in a .jar file. The JVM verifies the bytecode of the program before it is executed.

We enumerate the steps undertaken by the JVM to make a new object.

2.6.1 Allocate memory to store the object.

2.6.2 Move to the top of the class hierarchy.

In the process, invoke the constructor of the superclass from each higher level class up to the Object.

2.6.3 Make an instance of each Object going down the hierarchy inheritance trail.

We evaluate various object-oriented development configurations and compare their individual aspects. An attempt has been made to outline the more intrinsic features e.g. class, class body, inheritance, constructor, method and interface, and avoid repetition and over-elaboration. Other features such as exceptions, threads and sockets, which are strictly methods, can be discounted for purposes of evaluation.

2.7 A Java Class

A class is a blueprint or prototype that defines the variables and the methods common to all objects of a certain nomenclature. In object-oriented software, it is possible to have many objects of the same kind. These objects can be recreated as blueprints (templates). All objects in Java have state and behaviour. A blueprint of an object may be created by a class which can, further, define its data and behaviour. A class may inherit implementation from only one other class (Superclass). The class declaration must state the name of the class and may declare its Superclass with the keyword **extends**. Multiple inheritance is not allowed in Java and, therefore, obviates ambiguous implementation, However, multiple inheritance is permissible when using classes of a special nature. These special classes, called *interfaces,* have no implementation and no state and they, in turn, may optionally, implement one or more additional interfaces. The keyword **extends** declares that the ClassName is the subclass of SuperClassName. A subclass inherits variables and methods, their state and behaviour, from the Superclass. The class inherits all the attributes of the Superclass, which it extends, and can modify or override its attributes.

$$class = Identifier \times SClass\_Identifier \times Interface\_Identifier$$

<class-declaration> ::= <class-modifier> class <identifier> <extends> <implements>

2.7.1 Class Body

The class body is constructed of variable declarations and methods and contains the member variables and methods supported by the class. A class is the set of all items created using a specific pattern. It can be described as a set of all *instances* of that pattern.

<class-body> ::= <list-of-declarations> <list-of-methods>

<list-of-methods> ::= <method> ; <list-of-methods> | ε

$$Body = Sorts \times Constants \times Operations \times Methods.$$

2.8 Inheritance

A class inherits state and behaviour from its superclass. Inheritance provides a powerful and natural mechanism for organizing and structuring software programmes. The nature of an object can be determined by the definition of the class. Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, estate cars, saloon cars, and racing cars are all kinds of cars. In object-oriented terminology, they are all subclasses of the car class. Similarly, the car class is the superclass of estate cars, saloon cars, and racing cars. (See Green [1996-2005]).

2.9 A Constructor

All Java classes have constructors. A constructor is part of a class and is used to initialize a new object of that class type. The class constructor always has the same name as the class and has no return type.

(i) Java supports name overloading for constructors i.e. a class can have any number of constructors with the same name.

(ii) When writing a class, the Runtime System automatically provides a constructor for that class if one has not been included.

(iii) It may be termed that a constructor is a Method that uses its arguments to initialize the state of the new object.

(iv) The compiler can determine which constructor to implement based on the number of arguments used.

2.10 A Method

An object, although capable of performing, usually appears as a component of a larger program or application that contains other objects. Through the interaction of these objects, programmers are able to achieve higher-order functionality and more complex behaviour patterns. This software interaction is a communicative process known as message sending. When object (A) wants another object (B) invoke an object (B) method, a message is sent by object (A) to the object (B).

It follows that a class of objects that define such operations are required to interface with one and other in order that they may facilitate communication. This notion of interface is governed by the language protocol. In Java, classes make declarations to objects by using methods. Their implementation is defined as is their state. The method has two parts: the method declaration and the method body. The method declaration defines all of the method's attributes and the method body contains the Java instructions that implement the method. Java has an explicit case wherein a method may have a body without any instructions (empty parenthesis) and therefore no implementation.

2.11 An Interface

A Java interface is a contract in the form of a collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface. Within the Java language an interface is a device that unrelated objects use to interact with each other. An interface is probably most analogous to a protocol. In fact, other object-oriented languages have the functionality of interfaces and call their interfaces, protocols. An interface is a protocol of behaviour that may be implemented by any class anywhere in the class hierarchy.

2.12 Introduction to Java RMI

Java Remote Method Invocation (Java RMI) enables the program to create distributed Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines.

There are three processes that enable remote method invocation.

(i) The *Client* process that is invoking a method on a remote object.

(ii) The *Server* process that owns the remote object. The remote object is an ordinary object in the address space of the server process.

(iii) The *Object Registry* is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

Java RMI allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. (See Java Sun.com [2002]).

'In the Java distributed object model, a remote object is one whose methods can be invoked from another Java Virtual Machine, potentially on a different host. A remote object implements one or more remote interfaces, which are pure Java interfaces that declare the methods of the remote object. A method invocation on a remote object has the same syntax as a method invocation on a local object'.

The Java distributed object model preserves the Java object model in the following ways:

(i) a reference to a remote object can be passed as an argument or returned as a result in any local or remote method invocation,.

(ii) programmers can utilize natural Java mechanisms for the type-checking and casting of remote objects,

(iii) clients are able to interact with remote interfaces.

When a remote object reference is passed, that reference is made available to the client receiver. It is difficult to foretell the precise eventualities of the applications under review but it appears that RMI is moving closer towards CORBA and the two technologies could merge into a single, seamless, distributed object architecture to take advantage of the strengths of the two object disciplines.

Chapter 3

Decomposition of the Syntax of Java Language

We have reviewed some of the basic ideas about object-oriented programming in Chapter 2, now we write on the definition of the syntax of languages.

The syntax of a programming language is commonly specified using a grammar. A grammar defines the symbols that make up programmes in the language and, in particular, the rules for forming the legal programmes of the language. A practical language will need many rules. Large grammars extending over pages are common. For example, the syntax of Java as defined in Gosling *et al*, [1996] takes up 14 pages and has hundreds of rules.

There are two methods by which we can improve the way in which we read and use a grammar. We can improve:

- the presentation, by adapting and finding another notation more suitable for human or machine consumption, and
- the structure, by employing some concept of modularity with the aid of an importing construct that breaks down grammars into useful components.

The following sections introduce and apply the idea of *modular grammars*. We show how to construct such a grammar and how it is related to standard grammars. To illustrate our definition we give an example based on a **switch** fragment of Java. An example can be seen in Appendix 2.

Modular grammars were introduced informally in courses on formal languages to speed up the description of illustrated examples. See (Stephenson and Tucker [2006]). Here we apply modular grammars to the specification of a major subset of Java to investigate the techniques when used in a large language.

3.1 Grammars and Modular Grammars

3.1.1 Grammars

A grammar is a mathematical idea designed to specify formal languages and is a collection of rules to generate the strings of a language. These rules define how we are able to form these strings by means of systematic substitutions.

A grammar consists of the four components:

(1) a set T of **terminal** symbols, comprising alphabetical characters that appear in strings generated by the grammar,

(2) a finite set N of **non-terminal** symbols, which are placeholders, or variables, for patterns of terminal symbols generated by the non-terminal symbols where

$$N \cap T = \varnothing;$$

(3) a **start symbol** $S \in N$, which is a special non-terminal symbol that appears in the initial string generated by the grammar;

(4) a finite set P of **productions**, which are rules, of the form $u \rightarrow v$ where a non empty string $u \in (T \cup N)^+$ is written on the left side of the production and a string $v \in (T \cup N)^*$ is written on the right side of the production. Strings on the left and right hand side may contain terminals and / or non-terminals.

We display the quadruple $G = (T, N, S, P)$ as follows:

grammar	G
alphabet	T
nonterminals	N
start	S
rules	P

To generate a string of terminal symbols from a grammar we begin with a string consisting of the start symbol. We choose some production with the start symbol on the left hand side and replace it with the right hand side of the production. The process of choosing and applying productions to the string continues. Specifically, a production whose left hand side matches some substring of the current string of terminals and non-terminals is picked and that substring is replaced by the right hand side of the production. This process is repeated until all the non-terminals have been removed.

The *language* of a formal grammar $G = (N, T, P, S)$, denoted as $L(G)$, is defined as all those strings over Σ that can be generated, initially, with the start symbol S and then applying the production rules in P until no more non-terminal symbols are present.

The language $L(G)$ determined by grammar G is defined in the usual way:

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

where $\Rightarrow^*$ is the transitive closure of the one-step application $\Rightarrow$ of production rules. The details of these concepts are in Backhouse [1976].

3.1.2 The Use of Grammars in Syntax Design

The following procedure is used to specify the syntax of a language L:

Stage 1. The language L is built from symbols of some alphabet, so we choose an alphabet T, such that $L \subseteq T^*$.

Stage 2. We create a grammar G, such that $L \subseteq L(G)$.

Stage 3. Finally, unwanted or undesirable strings have to be removed from $L(G)$ to leave L.

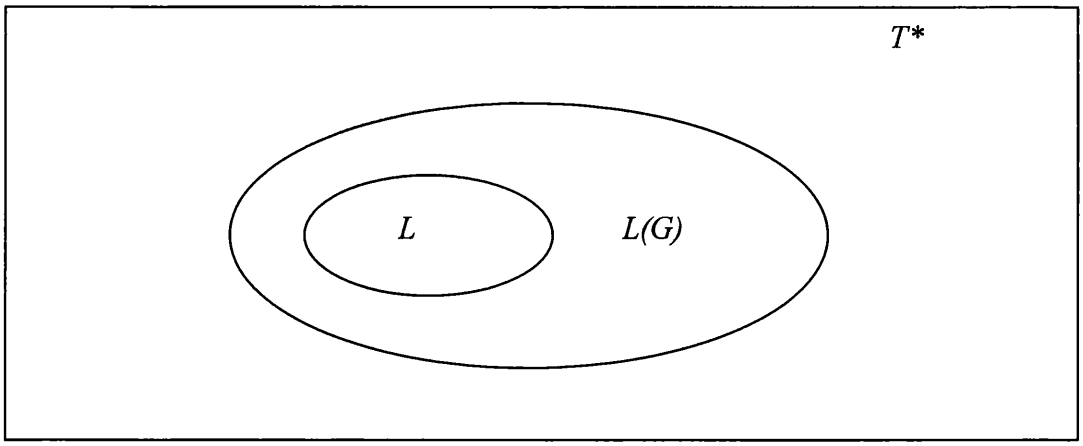


Figure 3.1 A language specification using a grammar.

3.1.3 The Import Construct

To allow a modular approach to the design of syntax, in which different syntactic components can be defined independently, we introduce a notion of a *modular grammar*. This idea is based on a simple addition of an import construct to grammars.

Consider a *modular grammar* which has the simple form:

$$G = (I, T, N, S, P)$$

where $I = \{G_1, \ldots, G_n\}$ is a list of non modular grammars that are to be imported and T, V, S and P represent the notion of a grammar as in (3.1.1). We display it:

grammar	G
import	I
alphabet	T
nonterminals	N
start	S
rules	P

Now the grammar G imports the list I of grammars. For simplicity, suppose that each G_i has *no* imports and

$$G_i = (T_i, N_i, S_i, P_i).$$

grammar	G_i
alphabet	T_i
nonterminals	N_i
start	S_i
rules	P_i

Then the meaning of the grammar G is defined by substituting the G_i's into G and eliminating the **import** construct.

grammar	*Flattened G*
alphabet	$T \cup T_1 \cup \ldots \cup T_n$
nonterminals	$N \cup N_1 \cup \ldots \cup N_n$
start	S
rules	$P \cup P_1 \cup \ldots \cup P_n$

In general, we must import modular grammars into modular grammars. This leads to a hierarchical structure. To define such modular grammars, and in particular, the flattened forms that they denote, we need a more complicated inductive definition.

Definition. A *modular grammar* is defined inductively by two clauses:

If (T, S, V, P) is a grammar then

G = (Φ, T, V, S, P) is a *modular grammar* with flattened form

F(G) = (T, V, S, P).

Let $I = \{G_i, \ldots, G_n\}$ be a set of *modular grammars* with flattened forms

$F(G_i) = (T_i, V_i, S_i, P_i)$

then $G = (I, T, V, S, P)$ is a modular grammar with flattened form

$F(G) = (T_f, V_f, S_f, P_f)$

where

$T_f = T \cup T_1 \cup \ldots \cup T_n$

$V_f = V \cup V_1 \cup \ldots \cup V_n$

$S_f = S \cup S_1 \cup \ldots \cup S_n$

$P_f = P \cup P_1 \cup \ldots \cup P_n$

Clearly, a stand-alone modular grammar is equivalent to the established concept of a grammar. We say that the above grammar, without imports, is the *flattened form* of *G*. We view the modular grammar *G* as a notation for its flattened form Flattened *G*.

More generally, in a modular grammar, the list *I* will be a list of other modular grammars, each of which may contain further imports. The flattened form can be defined by unpacking the grammars named as imports, step by step, provided that no import depends on itself and the process is non cyclic.

3.1.4 Special Features of Modular Grammars

The above definition is not without its complications and some refinements may be required. First, we can be precise about the set of grammars we are using and, upon which, the construction and flattening of grammars are operating. For example, in importing a grammar, we assume that it exists in our domain of grammars.

Secondly, we must take care of the case when we use a grammar that, in turn, imports itself. Here, flattening breaks down, in the sense that, we do not get a stand-alone grammar. For each modular grammar, we can track its dependency on other grammars

by drawing a graph. Normally, we expect this graph to be a tree whose leaves are stand-alone grammars. If a modular grammar depends on itself, this graph is cyclic. In this case one option is simply to ignore the attempt to "self import" and add nothing.

3.1.5 Modular Decomposition

We can consider how to decompose large grammars in different ways. This allows us to explore the idea of splitting grammars into smaller components.

The various components of the grammars used in the decomposition of the language syntax, together with the design and maintenance of the language specification, becomes easier, more manageable and accountable, and simpler to change.

3.1.6 Examples of Modular Grammars

We consider a simple fragment of Java to illustrate the idea of a modular grammar. We choose the fragment **switch** and give a modular grammar $G^{Switch\ Statement}$.

The *switch statement* is used to conditionally perform statements based on an integer expression. The value of this integer determines the invocation of an appropriate *case label* among those listed inside the block which follows. If a matching case label is found, control is transferred to the first statement after the label. If not, control is transferred to the first label following a *default label.* If there is no default label, the entire switch statement is skipped. The *break statements* are necessary because without them, control falls through the subsequent *case statements.* That is, without an explicit break keyword, control will flow sequentially through the case statements. A more detailed description can be seen in Appendix 2.

The architecture of $G^{Switch\ Statement}$ is as Figure 3.2

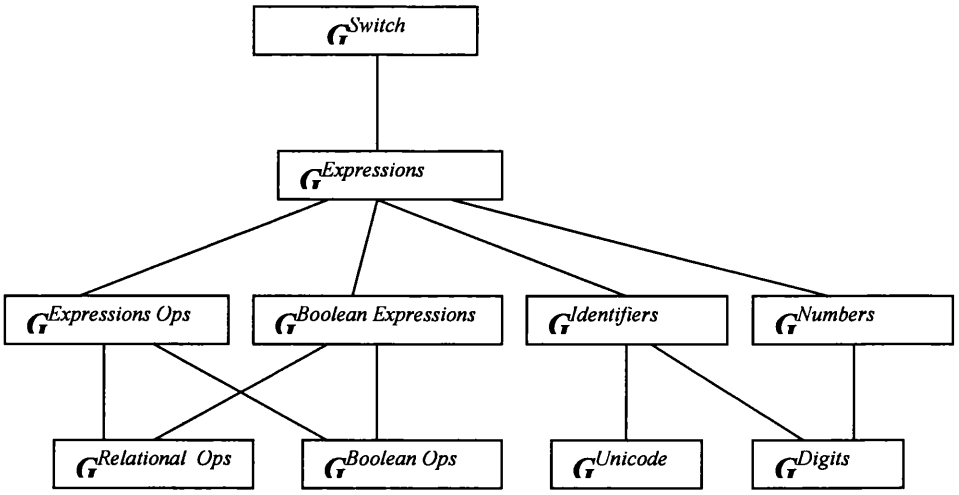


Figure 3.2 Component Grammars for Construction of $G^{Switch\ Statement}$

To avoid over-complication of imports we adopt a simplified process by using the branch G^{Switch} and its dependencies $G^{Expressions}$, $G^{Numbers}$, G^{Digits} as an example. The same would then be true of all branches. We refer to later defined grammars in Chapter 3.

G^{Digits}	3.2.1	$G^{Numbers}$	3.2.2	$G^{Identifiers}$	3.2.3
$G^{Expressions}$	3.2.4	$G^{Operators}$	3.2.5	$G^{Boolean}$	3.2.6

The following examples show how we use the mathematical idea of a grammar to build definitions for a simple programming language. The programming language we have chosen is Java and we itemise the specific components of a **Switch** programming language. A program example, showing the switch statement, is listed in Appendix 2.

(i) digits,

(ii) natural numbers,

(iii) identifiers,

(iv) expressions,

(v) operators,

(vi) booleans,

(vii) programmes

We start with a grammar G^{Digits} for generating digits:

grammar	G^{Digits}		
alphabet	**0,1,. . . , 9**		
nonterminals	*Digit*		
start	*Digit*		
rules	*Digit*	→	**0**
	Digit	→	**1**
		⋮	
	Digit	→	**9**

By means of the new construct, **import** we build a grammar $G^{Numbers}$ utilising the previous grammar G^{Digits}.

grammar	$G^{Numbers}$		
import	G^{Digits}		
alphabet	**. , O, Ox**		
nonterminals	*Number, Real, Octal, Hex*		
start	*Number*		
rules	*Number*	→	*Digit*
	Number	→	*Number Digit*
	Real	→	*Number . Number*
	Octal	→	**O** *Number*
	Hex	→	**Ox** *Number*

The next example, $G^{Expressions}$, calls on additional imports, other than $G^{Numbers}$, and we list them as shown.

grammar	$G^{Expressions}$		
import	$G^{Expression\ Operators}$, $G^{Boolean\ Expressions}$, $G^{Identifiers}$, $G^{Numbers}$		
nonterminals	*Exp*		
start	*Exp*		
rules	*Exp*	→	*Identifier*
	Exp	→	*Number*
	Exp	→	*PrefixUnaryOp Exp*
	Exp	→	*Exp PostfixUnaryOp*

The list of imports include the grammar $G^{Numbers}$ which, in turn, contains another **import** G^{Digits}. A further example can be seen in section 3.2.2.

We move, finally, to Figure 3.3, the Switch Statement Grammar $G^{Switch\ Statement}$,where we show the **import** components, $G^{Relational\ Operators}$, $G^{Boolean\ Operators}$ and $G^{Expressions}$.

grammar	$G^{Switch\ Statement}$		
import	$G^{Relational\ Operators}$, $G^{Boolean\ Operators}$, $G^{Expressions}$		
alphabet	**case, default, break, ; , :**		
nonterminals	*CaseList, Case, Break,* ε		
start	*CaseList*		
rules	*CaseList*	→	*Case*
	CaseList	→	*Case ; CaseList*
	Case	→	**case** *Expression : Break*
	Case	→	**case** *Expression : Statement Break*
	Case	→	**default** *: Break*
	Case	→	**default** *: Statement Break*
	Break	→	**break**
	Break	→	ε

Figure 3.3 A Java Switch Statement Grammar

3.1.7 Context Free Grammars

A *context-free grammar* (CFG) is a formal grammar in which every production rule is of the form V → *w*, where *V* is a non-terminal symbol and *w* is a string consisting of terminals and/or non-terminals. Such a rule is called a *context-free rule*.

The term "context-free" originates from the feature that the variable *V* can always be replaced by *w* in no matter what context it occurs. A set of finite length words, or strings, over some finite alphabet, comprises a formal language and this is said to be context free if there is a context-free grammar which generates it.

Context-free grammars are powerful enough to describe the syntax of programming languages in Stage 2 of their definition. i.e. for any language L we can always find a CFG G such that

(i) $L \subseteq L(G)$,

(ii) and in particular $L(G)$ is a "good approximation" to L.

In fact, almost all programming languages are defined via context free grammars, in this way. However, it is well known that context free grammars cannot define working languages *exactly*, i. e. there are programming languages L such that, for all context free grammars G:

$$L \neq L(G).$$

(See Backhouse [1978]).

Among the well known grammars of the Chomsky hierarchy are regular grammars and context sensitive grammars. Regular grammars are simpler than context-free grammars. Although they are useful in defining simple bits of syntax, like identifiers, they are unable to define the syntax of terms and commands. Context sensitive grammars are more complicated than context-free and can define more complicated languages. However, they are intricate to design, use and parse.

grammar	G
alphabet	T
nonterminals	N
start	S
rules	P

Definition A modular grammar *G = (I, T, V, S, P)* is said to be *context-free* if every rule in P is a context-free rule.

Proposition *The flattened form of a context-free modular grammar is a context-free stand-alone modular grammar, i.e. simply a context-free grammar.*

3.1.8 Backus-Naur Form

With the aid of these modular grammars we are able to construct a simple step by step account of a programming language. However, it is useful to introduce a BNF style notation for the modular context-free grammars.

A description of the grammar using BNF is as follows:

- The terminal symbols are written in bold font.
- Non-terminal symbols are enclosed in angle brackets. e.g., <identifier>, <digit>.
- The start symbol is the non terminal that is first in the list of appropriate productions.
- The symbol ::= (*is defined as*) indicates that the non-terminal expression on the right hand side is represented by the non-terminal productions on the left hand side.
- The symbol | (*or*) indicates that there is a possible alternative production to follow.

3.1.9 Extended BNF

We have previously described the acronym BNF as the Backus Naur Form and its invention is attributed to John Backus and Peter Naur who used it to interpret the ALGOL 60 language. (See Naur [1960]).

Later, some symbols used in regular expressions were added to the original BNF notation, giving rise to Extended BNF (EBNF). EBNF is simple powerful and defines the syntax of a language by using a number of rules. A terminal symbol is a symbol that

cannot be split into a smaller component of the language. In EBNF, these characters have special meanings:

(i) [ ] indicate optional symbols. For example, [x] indicates that x is optional.

(ii) { } indicate repetition.

(iii) ( ) groups items together

(iv) | separates alternatives. For example, x | y is read x or y.

3.2 Logical Decomposition

The model syntax of the language deals with numbers, identifiers and expressions forming the content of program statements. Declarations are types relevant to the Java language whilst modifiers are tabulated and then appropriately sub-divided dependent on their usage.

In the analysis of the decomposition we focus on the method, the class and subsequently the program. It can be argued that many other constructs should be included in this analysis such as exceptions, threads etc., and may be treated as special methods. However, we do not include them in our model.

It should be noted that all exceptions that can be generated, are subclasses of the class **java.lang.throwable.Exception**. With this in mind, and the idea of a hierarchy of errors, it is acceptable that they come under the overall definition set out for a class.

The Java Interface is constructed with a declaration and a body. The declaration may contain modifiers and the interface must be named. It may, also, optionally extend or implement an interface or list of interfaces. The interface body may contain constant declarations or, optionally, method declarations.

- **The Class** has similar characteristics, in that it is constructed with a declaration and a body. Its declaration may, optionally, contain modifiers and it demands a class name. It may, optionally, extend one SuperClass or implement an interface but not a list of interfaces. Its body contains variables and methods which, as members, are supported by the class. Classes parallel data types. They not only define the data fields used to determine the state of an object, they also specify the object's functionality. Once a class is defined, instances of that class can be created.

- **Methods** have similarities to interfaces and classes in that they have a declaration and a body together with the optional modifiers. Here the similarity ends as every method must incorporate parameters and they, optionally, contain arguments. The return value of the data type must be the same as that previously declared by the method.

- **Statements** cover the conditional, iterative, bounded, unbounded and case forms.

- **Types** are categorised into primitives, numbers, characters, boolean and reference types which are pointers to values or sets of values.

The result of this analysis is a modular grammar for a subset of Java that contains the essential constructs which could be extended to include all constructs. To test the "accuracy" of the modular grammar we need to flatten it and compare it with a standard definition of the relevant subset of Java. (See Section 3.3.). This would involve analysis of a standard grammar, as mentioned in Chapter 1, and a removal of the features we have omitted from the text in Section 3.3, The Flattened Grammar of Little Java.

This section defines the syntax of the Java programming language. We incrementally construct this definition on the basis of a logical decomposition of Java, as illustrated in Figure 3.5.

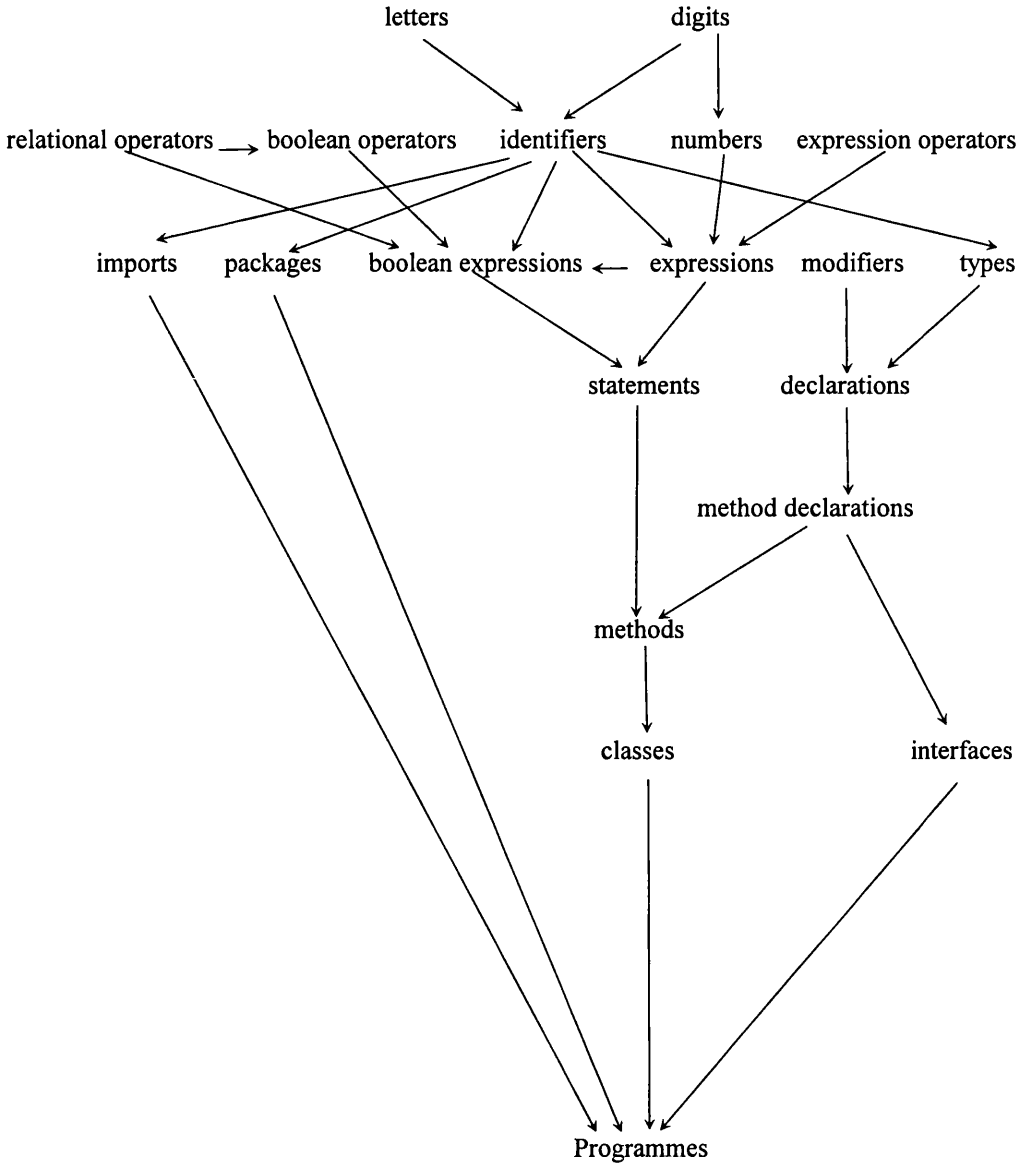


Figure 3.5 A Tree Diagram showing the logical decomposition of Java.

In Fig 3.5 the arrows indicate imports, e.g. *boolean expressions* import *expressions.*

3.2.1 Digits

A digit is one of ten Arabic symbols from 0 to 9; the symbol is used in a system of numeration.

grammar	G^{Digits}		
alphabet	**0,1,. . . , 9**		
nonterminals	Digit		
start	Digit		
rules	Digit	→	**0**
	Digit	→	**1**
		⋮	
	Digit	→	**9**

3.2.2 Numbers

A number is a non-empty sequence of digits. The different numeric types define the degree of precision with which a number is represented and the range of values it can accommodate. Numbers can be whole, or real, and may be in base 10, 8, or 16.

grammar	$G^{Numbers}$		
import	G^{Digits}		
alphabet	**. , O, Ox**		
nonterminals	*Number, Real, Octal, Hex*		
start	*Number*		
rules	*Number*	→	*Digit*
	Number	→	*Number Digit*
	Real	→	*Number . Number*
	Octal	→	**O** *Number*
	Hex	→	**Ox** *Number*

3.2.3 Identifiers

An identifier is constructed as a letter followed by a sequence of letters or digits contained in the Unicode list. Unicode is a list of symbols comprising a character-coded system that supports text constructed from various alphabets. The Website www.unicode.org has a comprehensive and updated collection of codes in use throughout the world. Subsets of these codes are used in Java development, depending on the country originating usage. There are 40,000 characters available in the Unicode set compared with 256 ASCII.

grammar	$G^{Identifiers}$		
import	G^{Digits}, $G^{Unicode}$		
alphabet	**a, b, . . . , z, A, B, . . . , Z**		
nonterminals	*Letter, Identifier*		
start	*Identifier*		
rules	*Identifier*	→	*Letter*
	Identifier	→	*Identifier Letter*
	Identifier	→	*Identifier Digit*
	Identifier	→	*Identifier Unicode*
	Letter	→	**a**
	Letter	→	**b**
		.	
		.	
		.	
	Letter	→	**z**
	Letter	→	**A**
	Letter	→	**B**
		.	
		.	
		.	
	Letter	→	**Z**

3.2.4 Expressions

An expression may be:
(i) an atomic expression formed by an identifier or number; or
(ii) an expression constructed from other expressions by the application of a unary operator (-, +, ++, --) to an expression; or
(iii) a binary operator applied to two expressions.

grammar	$G^{Expressions}$		
import	$G^{Expression\ Operators}$, $G^{Identifiers}$, $G^{Numbers}$		
nonterminals	*Exp,*		
start	*Exp*		
rules	*Exp*	→	*Identifier*
	Exp	→	*Number*
	Exp	→	*PrefixUnaryOp Exp*
	Exp	→	*Exp PostfixUnaryOp*
	Exp	→	*Exp BinaryOp Exp*

3.2.5 Expression Operators

Operators perform some function on either one or two operands. The prefix unary operator ++ before an identifier evaluates to the value of that identifier value after incrementing. The postfix unary operator ++ after an identifier evaluates to the value of the identifier before incrementing. The unary operator -- acts in the same way but decrements instead. Bitwise operators allow you to change data by manipulating bits. Note that the operators -, + may only be placed before an expression when used as unary operators; whereas the other unary operators + + and - - may be applied before or after expressions.

grammar	$G^{Expression\ Operators}$		
import	$G^{\text{Relational Operators}}, G^{\text{Boolean Operators}}$		
alphabet	++ , - - , + , - , * , / , %		
nonterminals	*PrefixUnaryOp, PostfixUnaryOp, Arithmetic, Bitwise*		
start	*PrefixUnaryOp*		
rules	*PrefixUnaryOp*	→	++
	PrefixUnaryOp	→	- -
	PrefixUnaryOp	→	+
	PrefixUnaryOp	→	-
	PostfixUnaryOp	→	++
	PostfixUnaryOp	→	- -
	Arithmetic	→	+
	Arithmetic	→	-
	Arithmetic	→	*
	Arithmetic	→	/
	Arithmetic	→	%
	Bitwise	→	>>
	Bitwise	→	<<
	Bitwise	→	>>>
	Bitwise	→	&
	Bitwise	→	'|'
	Bitwise	→	^
	Bitwise	→	~

3.2.6 Boolean Expressions

A boolean expression is either:

(i) an atomic boolean expression formed from the constants true or false; or an identifier; or

(ii) a boolean expression constructed from other boolean expressions; or

(iii) a result of applying a binary relational operator to expressions.

The value of any boolean expression is either true or false.

grammar	$G^{Boolean\ Expressions}$		
import	$G^{Relational\ Operators}$, $G^{Boolean\ Operators}$, $G^{Expressions}$		
alphabet	**true, false**		
nonterminals	*BoolExp, BoolOp1, Exp*		
start	*BoolExp*		
rules	*BoolExp*	→	**true**
	BoolExp	→	**false**
	BoolExp	→	*Identifier*
	BoolExp	→	*BoolExp BoolOp1 BoolExp*
	BoolExp	→	*Exp RelationalOp Exp*

3.2.7 Boolean Expressions Operators

Relational operators undertake the comparison of two values and the result is Boolean. Boolean expressions can be combined with boolean operators; the result is true or false.

grammar	$G^{Boolean\ Expression\ Operators}$
import	$G^{Identifiers}$, $G^{Expression\ Operators}$
alphabet	**>** , **>=** , **<** , **<=** , **==** , **=** , **&&** , '**||**' , **!**
nonterminals	*RelationalOp, BooleanOp*
start	*RelationalOp*
rules	*RelationalOp* → >
	RelationalOp → >=
	RelationalOp → <
	RelationalOp → <=
	RelationalOp → ==
	RelationalOp → =
	RelationalOp → !=
	BooleanOp → &&
	BooleanOp → '||'
	BooleanOp → !

3.2.8 Statements

Statements may be:

(i) assignments to variables or constants,
(ii) sequencing,
(iii) conditional statements of the form **if**, **else**, **else if**, or **switch**;
(iv) or iteration which may be bounded in the form of **for** loops,
(v) or unbounded in the form of **while** or **do while** loops.

grammar	$G^{Statements}$		
import	$G^{Boolean\ Expressions}$, $G^{Expressions}$		
alphabet	**final , if , else , else if , switch , for , while , do while , (,) , { , }**		
nonterminals	*Statement*		
start	*Statement*		
rules	*Statement*	→	*Type Identifier = Expression;*
	Statement	→	**final** *Type Identifier = Expression;*
	Statement	→	*Statement Statement*
	Statement	→	**if** (*BoolExp*) {*Statement*}
	Statement	→	**if** (*BoolExp*) {*Statement*} **else** {*Statement*}
	Statement	→	**else if** (*BoolExp*) {*Statement*}
	Statement	→	**switch** (*Type Identifier*) {*CaseList*}
	Statement	→	**for** (*Type Identifier = Expression; BoolExp; Expression*) {*Statement*}
	Statement	→	**while** (*Expression*) {*Statement*}
	Statement	→	**do** {*Statement*} **while** (*BoolExp*)

3.2.9 Switch Statement

The switch construct makes selections in case branches based on the value of an expression. The switch statement may use the break keyword to terminate a branch and move to the first statement following the case statement. Alternatively the break keyword may be omitted and the program would flow to subsequent case statements. The last break statement terminates the conditional switch. If a value passes through each case statement without any action taking place, the keyword default may be used to explicitly handle the event. See Appendix 2 for the program example. The following gives a simplified version of *Switch* (no nesting of *break*).

grammar	$G^{Switch\ Statement}$		
import	$G^{Relational\ Operators}$, $G^{Boolean\ Operators}$, $G^{Expressions}$		
alphabet	**case, default, break, ; , :**		
nonterminals	*CaseList, Case, Break,* ε		
start	*CaseList*		
rules	*CaseList*	→	*Case*
	CaseList	→	*Case ; CaseList*
	Case	→	**case** *Expression : Break*
	Case	→	**case** *Expression : Statement Break*
	Case	→	**default** *: Break*
	Case	→	**default** *: Statement Break*
	Break	→	**break**
	Break	→	ε

3.2.10 Declaration

A variable declaration has two components, the type of the variable and its name.

grammar $G^{Declarations}$

import $G^{Identifiers}$

alphabet **case, default, break, ; , :**

nonterminals *Declaration, Type, ListOfDec,* ε

start *Declaration*

rules

Declaration → *Type Identifier*

Declaration → ε

ListOfDec → *Declaration* ; *ListOfDec*

ListOfDec → ε

3.2.11 Modifiers

Modifiers may be used to control the behaviour of a class, interface, method or variable. Not all modifiers can be used on each of these elements as shown in Figure 3.5

Modifier	Class	Interface	Method	Variable
Public	yes	yes	yes	yes
protected	no	no	yes	yes
Private	no	no	yes	yes
Static	no	no	yes	yes
Abstract	yes	yes	yes	no
Final	yes	no	yes	yes
Native	no	no	yes	no
synchronised	no	no	yes	no

Figure 3.6 Modifiers and their usage.

grammar	$G^{Modifiers}$		
import	$G^{Identifiers}$		
alphabet			
nonterminals	*ClassMod, InterfaceMod, MethodMod, AccessMod, VarMod, Public, Abstract, Final, Static, Synchronised, Native, Private, Protected*		
start	*ClassMod*		
rules	*ClassMod*	→	*public abstract*
	ClassMod	→	*Public Final*
	InterfaceMod	→	*Public Abstract*
	MethodMod	→	*AccessMod Static Abstract Synchronised Native*
	MethodMod	→	*AccessMod Static Final Synchronised Native*
	AccessMod	→	*Public*
	AccessMod	→	*Final*
	AccessMod	→	*Protected*
	VarMod	→	*AccessMod Static Final*

3.2.12 Public

The public keyword declares that the object is totally accessible to any invocation from inside or outside the package. A public method or variable is visible wherever the class is visible.

grammar	G^{Public}		
import	$G^{Modifiers}$		
alphabet	**public**		
nonterminals	*Public*, ε		
start	*Public*		
rules	*Public*	→	**public**
	Public	→	ε

3.2.13 Protected

The protected accessor allows access to an object from its class, subclass and all classes within the package.

grammar	$G^{Protected}$		
import	$G^{Modifiers}$		
alphabet	**protected**		
nonterminals	*Protected*, ε		
start	*Protected*		
rules	*Protected*	→	**protected**
	Protected	→	ε

3.2.14 Private

The private accessor is the most restrictive and declares the object to be non-accessible other than to the class in which it is defined.

grammar	$G^{Private}$		
import	$G^{Modifiers}$		
alphabet	**private**		
nonterminals	*Private*, ε		
start	*Private*		
rules	*Private*	→	**private**
	Private	→	ε

3.2.15 Static

A static modifier declares an instance to be a class variable, or a method, to be a class method. Every instance of a class has its own instance variable memory location. A static class variable, or method argument, would have one memory location, irrespective of the number of instances of the class. The variable may be accessed by class name or through an instance of that class.

grammar	G^{Static}		
import	$G^{Modifiers}$		
alphabet	**static**		
nonterminals	*Static*, ε		
start	*Static*		
rules	*Static*	→	**static**
	Static	→	ε

3.2.16 Abstract

The abstract keyword, when used to modify a class, declares that the class exists, solely, to be sub-classed and cannot, therefore, be instantiated. It may be regarded as a prototype or unique parent class from which a child class can be copied. This enables the subclass to inherit state and behaviour from the parent class (super-class). An abstract class may contain abstract methods or non-abstract methods but a class which has an abstract method, must be declared abstract. Methods declared as abstract have no implementation and do not have a method body.

grammar	$G^{Abstract}$		
import	$G^{Modifiers}$		
alphabet	**abstract**		
nonterminals	*Abstract*, ε		
start	*Abstract*		
rules	*Abstract*	→	**abstract**
Abstract	→	ε	

3.2.17 Final

When used as a class modifier it signifies the class cannot be subclassed. The final modifier placed in a method declaration protects the method from being over-ridden by its subclasses. It follows that it would not be possible to declare a class as both abstract and final. If a variable is declared as final it indicates that the value of the variable will not be changed. The final modifier cannot be used on local variables.

grammar	G^{Final}		
import	$G^{Modifiers}$		
alphabet	**final**		
nonterminals	*Final*, ε		
start	*Final*		
rules	*Final*	→	**final**
	Final	→	ε

3.2.18 Native

The native keyword instructs the compiler that a method implementation is to be provided by another programming language. A native method returns a value of any type. The type must match the type specified in the method definition.

grammar	G^{Native}		
import	$G^{Modifiers}$		
alphabet	**native**		
nonterminals	*Native*, ε		
start	*Native*		
rules	*Native*	→	**native**
	Native	→	ε

3.2.19 Synchronised

A thread is a controlled task that operates within a program, independently, and without interference from other threads. Threads of this nature are termed asynchronous. When threads are called upon to run tasks that may require access to common data such as a file then there is need for special handling. Thread methods, in this case, use the synchronised keyword to prevent the class being modified by conflicting threads. It places a lock on the instance that invoked the method, to obviate the invocation of more than one thread at any particular time.

grammar	$G^{Synchronised}$		
import	$G^{Modifiers}$		
alphabet	**synchronised**		
nonterminals	*Synchronised*, ε		
start	*Synchronised*		
rules	*Synchronised*	→	**synchronised**
	Synchronised	→	ε

3.2.20 Types

A type may be primitive or reference.

grammar	G^{Types}		
import	$G^{Declarations}$		
alphabet			
nonterminals	*Type*, *primitive*, *reference*		
start	*Type*		
rules	*Type*	→	*primitive*
	Type	→	*reference*

3.2.21 Primitives

Primitive types are: byte, short, int, long, float, double, char and boolean. The different numeric types define the degree of precision with which a number is represented and the range of values it can accommodate.

Type	Definition
Boolean	true or false
Char	16 bit Unicode character
Byte	8 bit signed two's complement integer
Short	16 bit signed two's complement integer
Int	32 bit signed two's complement integer
Long	64 bit signed two's complement integer
Float	32 bit IEEE 754 floating point value
Double	64 bit IEEE 754 floating point value

Figure 3.7 Java Primitive Data Types.

3.2.22 Reference

Reference types are, as the name implies, types that have a pointer (reference) to the value, or set of values, held by the variable.

grammar	$G^{Reference}$		
import	$G^{Identifiers}, G^{Types}, G^{Numbers}$		
alphabet	[,]		
nonterminals	*Reference, Identifier, Array, Range,* ε		
start	*Reference*		
rules	*Reference*	→	*Identifier*
	Reference	→	*Array*
	Array	→	*Identifier* [*Range*]
	Range	→	*Number*
	Range	→	*Array*
	Range	→	ε

Arrays are generic data types. If the elements, within the array, are of type char then the array is of type char. A sequence of characters is called a string and all Java objects have a string class (and therefore a type string) which deal with string components in a special way. e.g. allowing the characters to be seen as array elements that can be utilised accordingly.

The array range is from 0 to a specified number or, the parameters may be left empty. Java does not cater for multidimensional arrays but you may have arrays of arrays.

3.2.23 Interfaces

An interface is a collection of declared methods and constants. It does not provide implementation for these methods. The interface is constructed from:

(i) the interface declaration, and
(ii) the interface body.

grammar	$G^{Interfaces}$
import	$G^{Identifiers}$, $G^{Declarations}$, $G^{Method\ Declarations}$
alphabet	{ , }
nonterminals	*Interface, InterfaceDec, InterfaceBody*
start	*Interface*
rules	*Interface* → *InterfaceDec { InterfaceBody }*

3.2.24 Interface Declaration

The interface declaration may optionally contain a modifier. It is placed before the interface identifier in order to regulate access to any invocation instigated elsewhere in the program. The declaration must state the name of the interface and optionally declare its SuperInterface with the keyword *extends*, and optionally implement one, or more, interfaces.

grammar	$G^{Interface\ Declarations}$		
import	$G^{Declarations}$, $G^{Method\ Declarations}$		
alphabet	**interface, extends, implements, ,**		
nonterminals	*Interface, InterfaceDec, InterfaceMod InterfaceBody, implements, InterfaceList*, ε		
start	*InterfaceDec*		
rules	*InterfaceDec*	→	*InterfaceMod* **interface** *Identifier* **extends** *Implements InterfaceList*
	Extends	→	**extends** *Identifier*
	Extends	→	ε
	Implements	→	**implements** *InterfaceList*
	Implements	→	ε
	InterfaceList	→	*Interface*
	InterfaceList	→	*Interface* , *InterfaceList*

3.2.25 Interface Body

The interface body may contain constant declarations and, optionally contain one, or more, method declarations. They have to be defined within the interface declaration.

grammar	$G^{Interface\ Body}$		
import	$G^{Declarations}$, $G^{Interface\ Declarations}$		
alphabet	;		
nonterminals	*InterfaceBody, ListofDecs, MethodDecList, MethodDec*		
start	*InterfaceBody*		
rules	*InterfaceBody*	→	*ListofDecs MethodDecList*
	MethodDecList	→	*MethodDec* ; *MethodDecList*
	MethodDecList	→	*MethodDec*
	MethodDecList	→	ε

The interface methods have parameters but no body and can only be implemented by calls from the class or other objects within inherited subclasses. An interface may extend any number of other interfaces. They do not provide multiple inheritance.

All statements within the interface body are implicitly public, static and final. The modifiers private and protected are not allowed in this case.

3.2.26 Methods

A method's implementation is constructed from a method declaration and, optionally, a method body.

grammar	$G^{Methods}$
import	$G^{Identifiers}$, $G^{Declarations}$, $G^{Method\ Declarations}$
alphabet	{ , }
nonterminals	*Method, MethodDec, MethodBody*
start	*Method*
rules	*Method* → *MethodDec { MethodBody }*

3.2.27 Method Declaration

The method declaration may optionally contain a method modifier. It is placed before the method identifier in order to regulate method access, state or behaviour. The declaration must state a method name, the return type, the number and type of its arguments. Java insists on the return value of data type to be identical to the method declaration data type. Methods may return reference data types or primitive data types. If no return value is required, the keyword void must be placed before the method.

grammar	$G^{Method\ Declarations}$
import	$G^{Identifiers}$, $G^{Declarations}$
alphabet	**method**, {, }, ,
nonterminals	*MethodDec, MethodMod, MethodBody, argList*
start	*MethodDec*
rules	*MethodDec* → *MethodMod* **method** *Identifier* { *argList* }

3.2.28 Method Body

The method body is constructed from variable declarations and, statements. The method body may contain local variables and methods supported by the class. Member variables can be static or non static. Methods can be declared in the same way.

grammar	$G^{Method\ Body\ and\ Method\ Expression}$		
import	$G^{MethodDeclarations}$		
alphabet	**new, type, {, }, void**		
nonterminals	*MethodBody, MethodDec, Statement, MethodExp, Identifier, argList, type,* ε		
start	*MethodBody*		
rules	*MethodBody*	→	*Statement*
	MethodBody	→	*type Identifier, MethodExp*
	MethodBody	→	*Identifier MethodExp*
	MethodBody	→	**New** *MethodDec*
	MethodBody	→	ε
	MethodExp	→	*expression, identifier, { argList }*
	ArgList	→	*type Identifier, argList*
	ArgList	→	ε
	ReturnType	→	*type*
	ReturnType	→	**void**

The keyword, *new*, is an unary operator and instantiates an object, or an array, to a memory allocation of that type. It creates the object, but a constructor is invoked to initialise a new object of a type previously declared. A constructor must have the same name as the class, in which it appears. The object is said to 'overload' the class identifier.

Overload methods, utilised in this manner, are distinguishable to the compiler by:

(i) the number and
(ii) the type of the arguments passed by the method.

3.2.29 Classes

A class is a template that can be used to instantiate other objects. It is constructed from:
(i) the class declaration and
(ii) the class body.

3.2.30 Class Declaration

The class declaration, may optionally contain a modifier. It is placed before the class identifier in order to regulate access to any invocation instigated elsewhere in the program. The declaration must state the name of the class and, optionally, declare its superclass with the *extends* keyword, and, optionally, implement one or more interfaces. (See Section 4). The keyword *extends* declares that the ClassName is the subclass of SuperClassName. A subclass inherits variables and methods, their state and behaviour, from the superclass. The class inherits all the attributes of the superclass, which it *extends*, and can modify, or override, those attributes.

3.2.31 Class Body

The class body is made up of variable declarations and methods and contains the member variables and methods supported by the class. Methods may be instance or class. Variables within the class body are of three types, instance, class and local.

grammar G^{Class}

import $G^{Methods}$ $G^{Identifiers}$ $G^{Modifiers}$

alphabet **class**, {, }, **extends**, **implements**, ;, ,

nonterminals *Method, ClassDec, ClassMod, Extends, Implements, ListOfDecs, ListOfMethods, InterfaceList, Interface,* ε

start *Class*

rules

Class	→	*ClassDec* { *ClassBody* }
ClassDec	→	*ClassMod* **class** *Identifier Extends Implements*
ClassBody	→	*ListOfDecs* , *ListOfMethods*
ListOfMethods	→	*Method* ; *ListOfMethods*
ListOfMethods	→	ε
Extends	→	**extends** *Identifier*
Extends	→	ε
Implements	→	**implements** *InterfaceList*
Implements	→	ε
InterfaceList	→	*Interface InterfaceList*
InterfaceList	→	ε

3.2.32 Special Cases

Java provides standard methods to deal with errors. The process is termed exception handling and is a means to deal with unusual conditions which may arise when programmes are executed. The class *Exception* and *Error* together with their subclasses, inherit methods from their parent class *Throwable*. These methods are useful as aids when debugging program errors. The exceptions are 'caught' by the try method and can be dealt with accordingly. They can be categorised as follows:

- Try-catch Exceptions caught in the body of the Try method should be handled by the catch method and outside of that try block.
- If an optional method is present, it must include a return statement.

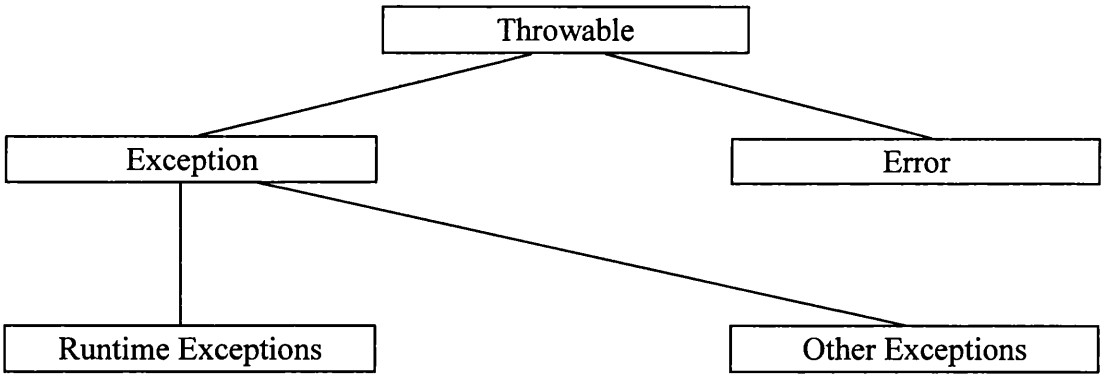


Figure 3.8 An illustration of some special cases.

The object of the previous paragraph is to emphasise that the methods above, although specialist, are worthy of mention.

Every non static instance member variable of a class has its own memory address but a static, or class variable, would have one memory location, shared by the instances of the class. The variable may be accessed by class name or through an instance of that class. The functionality of this is that there are times when subclasses may depend, favourably, on certain variables retaining a common value. Methods that utilise class variable must be termed class methods.

3.2.33 Programmes

Java is an interpretative, object-oriented, programming language. Its origins are based on other well-known languages and its derivation is such that it is able to perform in a platform independent manner. The Java compiler is used to convert the source program into byte-code. This code can then be executed, within the Java runtime environment, with the aid of the Java interpreter.

A program, or application, consists of

(i) Optionally, imports;
(ii) optionally, a package name (See 3.2.35);
(iii) classes, and
(iv) optionally, interfaces.

grammar	$GP^{rogram,\ List\ of Classess\ and\ List\ of\ Interfaces}$		
import	$G^{Methods}\ G^{Interfaces}\ G^{Identifiers}$		
alphabet			
nonterminals	*ClassList, InterfaceList, Interface,* ε		
start	*Program*		
rules	*Program*	→	*Import Package ClassList InterfaceList*
	ClassList	→	*Class*
	ClassList	→	*Class ClassList*
	InterfaceList	→	*Interface InterfaceList*
	InterfaceList	→	ε

3.2.34 Imports

The constructs of importing discussed in Chapter 3.1.4 (Import Constructs) can be improvised to import data types. To import any data type, we require two stages:

The language Java, allows the programmer to import other classes or interfaces or classes and interfaces. The collection is termed a package and the package is identified by a unique name. An asterisk, in an import statement, can only be used to specify all programmes in the package. (e.g. import graphics.*). Similarly a full stop may be used in a specific manner (import graphics.rectangle).

grammar	$G^{Imports}$		
import	$G^{Identifier}$		
alphabet	**import, ;**		
nonterminals	*Import,* ε		
start	*Import*		
rules	*Import*	$\rightarrow$	**import** *identifier* **;** *Import*
	Import	$\rightarrow$	ε

3.2.35 Packages

If a program name is not designated specifically as a package the Java run-time system assigns the application a package default setting without a name. To create a package, the package keyword, together with a package name, is placed directly before the first class declaration in the program. A package is a group of related classes, and, or interfaces, that comprise the program. Only one package can be attributed to a source file at any one time.

grammar	$G^{Packagess}$		
import	$G^{Identifier}$		
alphabet	**package,**		
nonterminals	*Package,* ε		
start	*Package*		
rules	*Package*	→	**package** *identifier*
	Package	→	ε

We show an example of the above in the form of a typical Java class programming setup.

Package Mycounter.library;

Public class Counter{

}

The classes and interfaces within the Java Development Kit (JDK) are members of packages bundled in such a way as to facilitate connectivity and functionality. They are termed as imports. Java might need to find other classes named in the main class definition. The compiler has to know where to look for these classes in the import statements.

Import java.awt.* will search for all classes in the java.awt directory to find the appropriate class definition.

3.3 The Flattened Grammar of Little Java

Let us now flatten our modular grammar for a subset of Java and briefly compare it with a standard definition of Java syntax. Here is the flattened version of the modular grammar in 3.2.

grammar $G^{\text{Little Java}}$

alphabet 0,1,. . . , 9, . , O, Ox, a, b, . . . , z, A, B, . . . , Z, ++ , - - , + , - , * , / , %, true, false, >= , < , <= , == , = , && , '||' , !, final , if , else , else if , switch , for , while , do while , (,) , { , }, case, default, break, ; , :, :, public, protected, private, static, abstract, final, native, synchronised, [,], **interface, extends, implements, method, {, }, new, type, void, class, import, package**

nonterminals *Digit, Number, Real, Octal, Hex, Letter, Identifier, Exp, , Expression PrefixUnaryOp, PostfixUnaryOp, Arithmetic, Bitwise, BoolExp, BoolOp1, RelationalOp, BooleanOp, Statement, CaseList, Case, Break, ε, Declaration, Type, ListOfDec, ClassMod, InterfaceMod, MethodMod, AccessMod, VarMod, Public, Abstract, Final, Static, Synchronised, Native, Private, Protected, primitive, reference, Reference, Array, Range, Interface, InterfaceDec, InterfaceBody, InterfaceMod, InterfaceList, ListofDecs, MethodDecList, MethodDec, Method, MethodBody, MethodMod, argList, Statement, MethodExp, type, ClassDec, ClassMod, Extends, Implements, ListOfDecs, ListOfMethods, ClassList, Import, Package*

start Program

rules

Digit	→	**0**
Digit	→	**1**
	⋮	
Digit	→	**9**
Number	→	*Digit*
Number	→	*Number Digit*
Real	→	*Number . Number*
Octal	→	**O** *Number*
Hex	→	**Ox** *Number*
Identifier	→	*Letter*
Identifier	→	*Identifier Letter*
Identifier	→	*Identifier Digit*
Identifier	→	*Identifier Unicode*
Letter	→	**a**

Letter → **b**
.
.
.
Letter → **z**
Letter → **A**
Letter → **B**
.
.
.
Letter → **Z**
Exp → *Identifier*
Exp → *Number*
Exp → *PrefixUnaryOp Exp*
Exp → *Exp PostfixUnaryOp*
Exp → *Exp BinaryOp Exp*
PrefixUnaryOp → ++
PrefixUnaryOp → - -
PrefixUnaryOp → +
PrefixUnaryOp → -
PostfixUnaryOp → ++
PostfixUnaryOp → - -
Arithmetic → +
Arithmetic → -
Arithmetic → *
Arithmetic → /
Arithmetic → %
Bitwise → >>
Bitwise → <<
Bitwise → >>>
Bitwise → &
Bitwise → |
Bitwise → ^
Bitwise → ~
BoolExp → **true**
BoolExp → **false**
BoolExp → *Identifier*
BoolExp → *BoolExp BoolOp1 BoolExp*
BoolExp → *Exp RelationalOp Exp*
RelationalOp → >
RelationalOp → >=
RelationalOp → <
RelationalOp → <=
RelationalOp → ==
RelationalOp → =

RelationalOp → !=
BooleanOp → &&
BooleanOp → '||'
BooleanOp → !
Statement → *Type Identifier = Expression;*
Statement → **final** *Type Identifier = Expression;*
Statement → *Statement Statement*
Statement → **if** (*BoolExp)* {*Statement*}
Statement → **if** (*BoolExp*) {*Statement*} **else** {*Statement*}
Statement → **else if** (*BoolExp*) {*Statement*}
Statement → **switch** (*Type Identifier*) {*CaseList*}
Statement → **for** (*Type Identifier = Expression; BoolExp; Expression*) {*Statement*}

Statement → **while** (*Expression*) {*Statement*}
Statement → **do** {*Statement*} **while** (*BoolExp*)
CaseList → *Case*
CaseList → *Case ; CaseList*
Case → **case** *Expression : Break*
Case → **case** *Expression : Statement Break*
Case → **default** *: Break*
Case → **default** *: Statement Break*
Break → **break**
Break → ε
Declaration → *Type Identifier*
Declaration → ε
ListOfDec → *Declaration* **;** *ListOfDec*
ListOfDec → ε
ClassMod → *public abstract*
ClassMod → *Public Final*
InterfaceMod → *Public Abstract*
MethodMod → *AccessMod Static Abstract Synchronised Native*
MethodMod → *AccessMod Static Final Synchronised Native*
AccessMod → *Public*
AccessMod → *Final*
AccessMod → *Protected*
VarMod → *AccessMod Static Final*
Public → **public**
Public → ε
Protected → **protected**
Protected → ε
Private → **private**
Private → ε
Static → **static**

Static	→	ε
Abstrac	→	**abstract**
Abstract	→	ε
Final	→	**final**
Final	→	ε
Native	→	**native**
Native	→	ε
Synchronised	→	**synchronised**
Synchronised	→	ε
Type	→	*primitive*
Type	→	*reference*
Reference	→	*Identifier*
Reference	→	*Array*
Array	→	*Identifier* **[***Range***]**
Range	→	*Number*
Range	→	*Array*
Range	→	ε
Interface	→	*InterfaceDec { InterfaceBody }*
InterfaceDec	→	*InterfaceMod* **interface** *Identifier* **extends** *Implements InterfaceList*
Extends	→	**extends** *Identifier*
Extends	→	ε
Implements	→	**implements** *InterfaceList*
Implements	→	ε
InterfaceList	→	*Interface*
InterfaceList	→	*Interface* **,** *InterfaceList*
InterfaceBody	→	*ListofDecs MethodDecList*
MethodDecList	→	*MethodDec* **;** *MethodDecList*
MethodDecList	→	*MethodDec*
MethodDecList	→	ε
Method	→	*MethodDec { MethodBody }*
MethodDec	→	*MethodMod* **method** *Identifier { argList }*
MethodBody	→	*Statement*
MethodBody	→	*type Identifier, MethodExp*
MethodBody	→	*Identifier MethodExp*
MethodBody	→	**New** *MethodDec*
MethodBody	→	ε
MethodExp	→	*Expression, identifier, { argList }*
ArgList	→	*type Identifier, argList*
ArgList	→	ε
ReturnType	→	*type*
ReturnType	→	**void**
Class	→	*ClassDec { ClassBody }*

ClassDec → *ClassMod* **class** *Identifier Extends Implements*
ClassBody → *ListOfDecs* , *ListOfMethods*
ListOfMethods → *Method* ; *ListOfMethods*
ListOfMethods → ε
Extends → **extends** *Identifier*
Extends → ε
Implements → **implements** *InterfaceList*
Implements → ε
InterfaceList → *Interface InterfaceList*
InterfaceList → ε
Class → *ClassDec { ClassBody }*
ClassDec → *ClassMod* **class** *Identifier*
Extends Implements
ClassBody → *ListOfDecs* , *ListOfMethods*
ListOfMethods → *Method* ; *ListOfMethods*
ListOfMethods → ε
Extends → **extends** *Identifier*
Extends → ε
Implements → **implements** *InterfaceList*
Implements → ε
InterfaceList → *Interface InterfaceList*
InterfaceList → ε
Program → *Import Package ClassList InterfaceList*
ClassList → *Class*
ClassList → *Class ClassList*
InterfaceList → *Interface InterfaceList*
InterfaceList → ε
Import → **import** *identifier* ; *Import*
Import → ε
Package → **package** *identifier*
Package → ε

3.4 Evaluation

Languages used in practical programming can never have small syntax. The subset of Java essentially determined by methods and inheritance is not small. The flattened grammar with its hundreds of rules illustrates this. The advantages of the modular approach to syntax seems to be:

1. that mathematical properties of modular grammars are closely related to conventional grammars because of the simple definition of flattening and

2. modular grammars lead to a systematic unfolding of the syntax in which each syntactical category can be reflected on. This is a useful tool for programming language designers as they make decisions that could affect users for years.

3. Modular grammars make it easy for reliability to

   a. change the syntax of a part of a language,

   b. specify fragments and subsets.

4. Modular grammars could easily maintain new language processing tools that support modular construction of new programming languages.

Chapter 4

A Model for Interface Definition Languages

In this chapter we now turn from syntactic specifications to the second topic of the thesis that of modelling interfaces. Commercially, many leading IT organisations in the world have encountered great difficulty trying to deal with rewrites and poorly prepared programming code. The commercial world has realised the significance of ideas about interfaces and have embraced them.

The topic of interface definition languages is relatively new. Computer scientists have been researching the subject for a number of years in connection with how to build software in a modular way using software components. More generally, interfaces have become prominent in object-oriented programming languages and tools. (See discussion of Corba in Chapter 2). However, the algebraic specification community has used mathematically sound concepts of interfaces to ensure pure, uncluttered and well scrutinised non-ambiguous programming specifications for over thirty years.

What are interfaces? What is an interface definition language? An attempt at an abstract and general model of the concept of interface is Rees, Stephenson and Tucker [2003]. We will explain the main ideas of that paper, in preparation for our analysis of their application in Java.

4.1 Interfaces

An interface definition language is used to define the interfaces of components and how they are involved in making a system interface. An interface offers no implementation for any of its operations but gives names to a collection of operations that combine to carry out logical operations within that system interface.

How do we make the notion of interface explicit in the mathematical modelling of any programming language?

To make progress on this question we consider some general ideas about interfaces and express them in a model of an interface definition language. Specifically, to examine the general notion of an interface, we follow the ideas in Rees, Stephenson and Tucker [2003]. In this paper an interface is treated very abstractly as follows. They model an Interface Definition Language as sets of interfaces, called *repositories*, and give them some algebraic operations resulting in a sort of algebra of interfaces. This algebraic structure allows them to define a notion of system architecture via the algebraic idea of term. A key idea is to allow an interface to have imports. They show how a *dependency trail* represents the data dependencies of an interface and investigate the properties of these interfaces. The imported interface dependency is made redundant by a transformation technique known as 'flattening' which is a sort of assembly process.

In the thesis we deal, primarily, with object-oriented systems and so the question is, can such a simple approach offer anything to object-oriented programming? Can the approach be the basis which could be used, in principle, for other languages and their respective architectural styles? Interfaces should define the interaction between separate software systems at each stage of abstraction and should provide details of common data definitions together with information on the interaction and control of data within a system environment. This applies, particularly, to larger systems. By investigating a real OO language like Java we can test the ideas and see what works and what is missing.

4.1.1 Interface Components

We assume that the abstract structure of an interface has these three components:

(i) Name of Interface,

(ii) Import List of Interfaces,

(iii) Body of Interface.

The focus of the abstraction is the identity of an interface, i.e. its *Name*, and the other interfaces upon which it may rely, *Imports*. What makes the notion general is the ability to choose different bodies.

The main interest in Rees *et al.*, [2003] is the process of assembling an interface from the component interfaces named in imports. This process is called 'flattening'. To define the process we need to define:

(i) basic interfaces that have no components and can be implemented directly; these are so called stand-alone interfaces,

(ii) abstract properties and operations on bodies that put interfaces together and

(iii) a global space of interfaces where all interfaces can be found.

4.1.2 Stand-alone Interface

Definition. An *interface* has a name identifier, imports and a body. An interface that has a body with no imports is said to be a *stand-alone interface*. Thus an interface is a triple of the form:

$$(Name, Import, Body).$$

Such an *interface* is capable of operation without calling on any outside source. It is the integral component from which other interfaces can be constructed. Its basic constructors are given in Figure 4.1.

algebra	*Stand_Alone_Interface*
import	*Identifier, Body*
carriers	*Stand_Alone_Interface*
constants	
operations	*make_interface:* $Identifier \times Body \rightarrow Stand\_Alone\_Interface$ *name_tag:* $Stand\_Alone\_Interface \rightarrow Identifier$ *body:* $Stand\_Alone\_Interface \rightarrow Body$
definitions	$name\_tag(make\_interface(n, B)) = n$ $body(make\_interface(n, B)) = B$

Figure 4.1 Stand-alone Interface.

4.1.3 Properties of Operations and Bodies

Bodies vary in their specification. To manipulate bodies algebraically, Rees *et al.*, [2003], use the following:

(i) A body with no content which is a constant *null*.

(ii) An operation *join* to concatenate two bodies, and

(iii) *tag* to rename a body's components.

Algebra	*Bodies*
Imports	*Identifiers*
Carriers	*Body*
Constants	*null:* $\rightarrow$ *Body*
Operations	*join* : *Body* $\times$ *Body* $\rightarrow$ *Body* *Tag* : *Body* $\times$ *Identifier* $\rightarrow$ *Body*
Definitions	*Tag(null, n) = null* *Tag(join(B, C), n) = joinTag(B, n), Tag(C, n))*

Figure 4.2 Body of Interface.

The following algebra illustrates the production of a stand-alone interface. In order to remove imports we use the operator *mkSAIntf*. This application, when invoked, forces the interface with imports to be a Stand Alone Interface, an interface without imports. The operation directly responsible for this is called *trivial* which, as its name implies, mathematically zeros the interface name. In order to maintain the contents of the original import the operator *extends,* in conjunction with *mkIntf,* and adds to the component declarations of the existing interface, the name, imports and body of another. The process of removing the imports is termed 'flattening'.

Algebra *Interface*

Import *SAInterfaces, Names*

Carriers *Interface*

Constants

operations

intf :	*Identifier* × *Names* × *Body* → *Interface*
name :	*Interface* → *Identifier*
imports :	*Interface* → *Names*
body :	**Interface → Body**
extend :	*Interface* × *Interface* → *Interface*
trivial :	*Identifier* → *SAInterface*
mkInft :	*SAInterface* → *Interface*
mkSAIntf :	*Interface* → *SAInterface*

definitions

name(intf (n, I, B)) = n

imports(intf (n, I, B)) = I

body(intf (n, I, B)) = B

extend(I, J) = intf(name(I), concat(cut(name(J), imports(I)), imports(J)), join(body(I), tag(body(J), name(J)))

Trivial(n) = sa_intf(n, null)

mkIntf(I) = intf(name(I), empty$_{names}$, body(I)

$$mkSAItnf(I) = \begin{cases} sa\_intf(name(I), (body(I)) \text{ if } imports(I) = empty_{names}; \\ trivial(names(I)) \text{ otherwise.} \end{cases}$$

Figure 4.3 Modelling a Stand-alone Interface.

Rees, Stephenson and Tucker [2003].

4.2 Imports and Repositories

Interfaces may import the features of other interfaces by invoking the interface name. This, in turn, demands that all named interfaces, together with imports, are contained in the global list of interfaces. It is this full and unambiguous listing that makes up a library of interfaces, or repository.

The problem of defining a 'well formed' interface is solved by placing conditions on the library, or repository. An interface that has a body with no imports is said to be a stand-alone interface; it is the integral component from which other interfaces can be constructed and it must be free of repetition. When an interface is not 'stand-alone' stipulations must be met for the interface, together with its body, to be well-formed.

Firstly, any interface listed among the imports must be present within the repository. Secondly, in calling an interface via the import mechanism repeatedly we do not encounter a cycle in the list of names found in that repository. An interface name that is needed does not, subsequently, re-import itself. Shortly, we will define the Dependency Trail which we can use to formalise and rule out this cyclic behaviour.

The key points are:

(i) a well formed interface can always be flattened into a stand-alone interface, and

(ii) if every interface is well-formed then the repository is said to be well-formed.

A repository is defined as a non-empty list of interfaces with unique names.

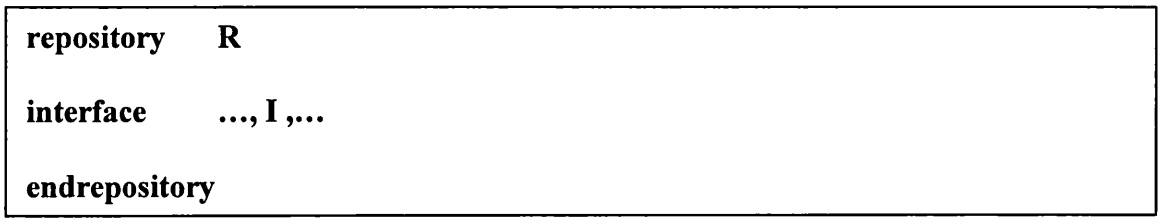


Figure 4.4 A Repository.

In the case of a well formed Repository we may define the algebra:

algebra	*Repository*
import	*Interface*
carriers	*Repository*
constants	
operations	*named*: *Repository* → *Names*
definitions	$named(I_1, \ldots, I_k) = name(I_1), \ldots, name\ (I_k)$

Figure 4.5 Repository Algebra.

We use repositories to define operations that specify what actually happens when an interface is imported into another interface. Specifically they are based on two items:

(i) tagging, the process of recording information on the location (address) of the various interface names:

$$Tag_{intI} : Name \times intf_I \rightarrow Name$$

$$Tag\ (N, B) = body\ (Tag_{intI}(N, intf_I\ (B)), \ldots, Tag_{intm}\ (N, intf_m\ (B)));\ and$$

(ii) joining, the process of adding one components of one body to the components of another. It satisfies properties such as:

$$Join\ (B, B) = B$$

4.3 Dependency Trail Definition

Within a repository, each interface would demand a unique name. An interface may import the features and content of another interface by declaring its name. An interface thus formed is dependent on the named import. The list, or record of dependent interfaces, is termed a *dependency trail*. So that we may interrogate the properties of these interfaces, we employ the technique known as 'flattening' which transforms the import declaration properties of an interface. We extend the original interface by adding the import definition. The imported interface dependency is made redundant when the

import is removed and the 'flattened' interface is termed a stand alone interface. The dependency factors are substituted with other unique identifiers within the trail.

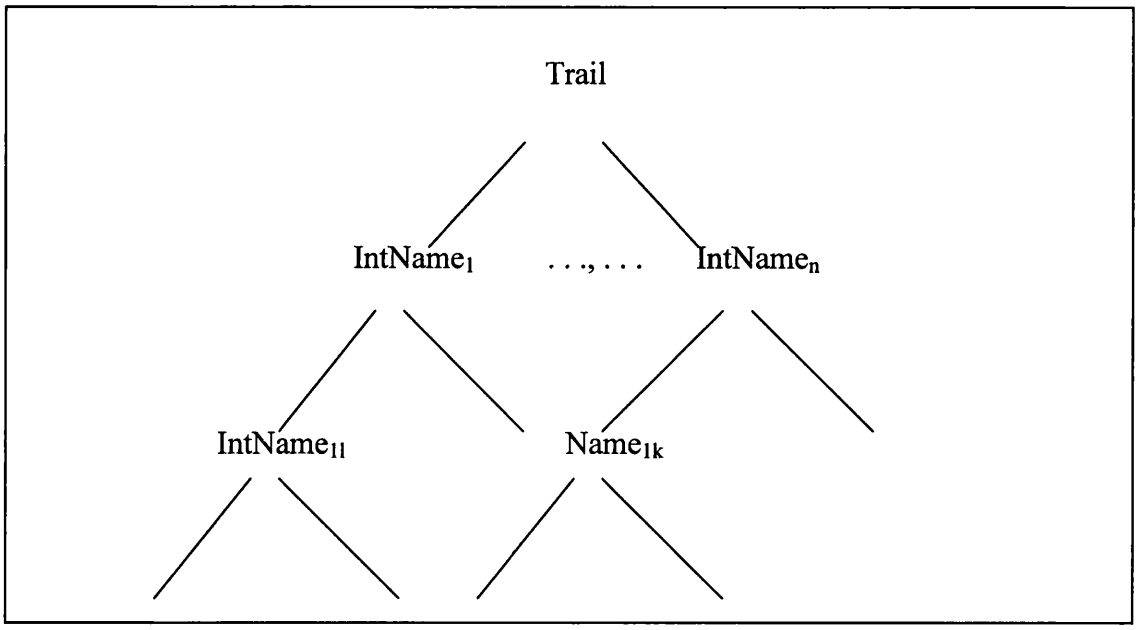
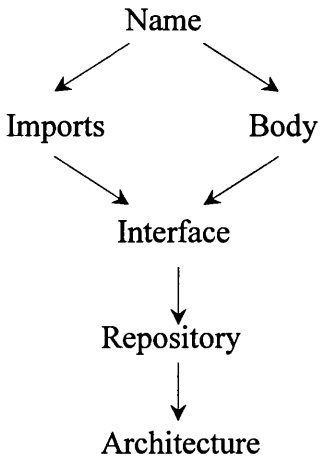


Figure 4.6 Dependency Trail Tree.

4.4 Architecture and Flattening

We have defined an interface as a declaration of name, a list of imports and a body. We have defined repositories as sets of interfaces. An Architecture is a structured set of interfaces defined by a term based on the import operation within the repository.



With an abstract notion of architecture we can define a *Dependency Trail* as a function to maintain a list of interfaces within a repository. The function is recursive on the structure of the architecture and moves through the dependency trail checking and removing any name repetitions that may be found. The architecture and the repository has a trail of import dependencies and flattening entails the removal of these dependencies:

$$Flatten : Architecture \rightarrow Stand\text{-}Alone,$$

$$flatten\ (A)\ \varepsilon\ Stand\text{-}Alone$$

The well-formed interface is defined as the non-ambiguous adaptation of an import interface. If the constraints imposed on the syntax are observed in keeping with the 'flattening' process then the Interface can be said to be a *Stand-Alone* interface and that the interface is well-formed and, therefore, the architecture is well-formed.

$$f : Architecture \times Interface \times Name^* \rightarrow Interface$$

Examples. Here is a simple example illustrating the ideas for data interfaces. We specify interfaces *Real, Bool*, *RealBool* and *RealBoolEqual.* We show an interface *Real* with no imports:

repository

interface **Real**

imports

sorts *real*

ops *add :* $real \times real \rightarrow real$
minus : $real \times real \rightarrow real$
mult : $real \times real \rightarrow real$
div : $real \times real \rightarrow real$

endinterface Real

Figure 4.7 Interface Real.

In Figure 4.8 the interface *Bool* has no imports, and is termed a *Stand-alone* Interface.

interface **Bool**

imports

sorts bool

ops

true : $\rightarrow$ *bool*
false : $\rightarrow$ *bool,*
not : *bool* $\rightarrow$ *bool*
and : *bool* $\times$ *bool* $\rightarrow$ *bool*
or : *bool* $\times$ *bool* $\rightarrow$ *bool*

endinterface Bool

Figure 4.8 Interface Bool.

The interface *RealBool* is constructed by invoking interface *Real* and interface *Bool* with the import definition.

Interface **RealBool**

imports *Real, Bool*

sorts *real, bool;*

ops

endinterface RealBool

Figure 4.9 Interface RealBool.

In Figure 4.12, as the Interfaces *Real* and *Bool* are Imports they no longer have to be declared as operations. The operation *eq*, however, is not an import and is, therefore listed as an operation statement.

Interface **RealBoolEqual**

Imports *Real, Bool*

sorts

ops

eq : $real \times real \rightarrow bool$

endinterface RealBool

Figure 4.10 Interface RealBoolEqual.

In order to 'flatten' the interface *RealBool* we remove the import dependencies. The interface *Real* and the interface *Bool* are both listed as operations in the new interface *RealBool*. This is illustrated in Figure 4.11.

interface RealBool

imports

sorts *real, bool;*

ops

add : $real \times real \rightarrow real,$
minus : $real \times real \rightarrow real,$
mult : $real \times real \rightarrow real,$
div : $real \times real \rightarrow real,$
true : $\rightarrow bool,$
false : $\rightarrow bool,$
not : $bool \rightarrow bool,$
and : $bool \times bool \rightarrow bool,$
or : $bool \times bool \rightarrow bool,$
eq : $real \times real \rightarrow bool;$

endinterface RealBool

Figure 4.11 New Interface RealBool with no Imports.

When 'flattening' occurs certain problems come to light. The following points are problem areas that may be encountered in such circumstances:

- Names in an import list of interface dependencies may not be found in a repository. Thus flattening does not produce a stand-alone interface.
- Names may be duplicated in a repository.
- The interface has been used recursively in the assembly process, if its own name appears in its dependency trail.
- Methods which may or may not contain parameters.
- Queries which do not alter state but invoke a response, value or await a reply.

For strategies that could be employed when such questions arise, see Rees, Stephenson and Tucker [2003].

4.5 General Flattening Algorithm

Given a repository and interface with imports, we can attempt to trace dependencies by traversing the dependency graph. In order to flatten a signature, we need to combine the interfaces that it depends on. So, how do we combine interfaces? Let us suppose we have an operation

Expand : Interfaces with Imports $\times$ *Interfaces with Imports* $\rightarrow$ *Interfaces with imports*

so that

$$Expand(\Sigma, \Sigma^{I})$$

is an interface with the same name as Σ, and will join the imports, sorts, constants and operations of the interfaces Σ, Σ^{I}.

We use records to represent interfaces and produce a template algorithm for performing as follows:

```
//Given an interface Σ as input, creates its flattening version Flattened

//Creates a copy Flattened of Σ and renames it as Flattened Σ.
Flattened= Σ;
Flattened. Name = concat ("Flattened" = Σ.Name);

// Pick out the imports of I of Σ
I = Σ.Imports;

//Whilst there is an import in Flattened
while I != 0 {

//Pick an import I
  switch {
      case i ε I :

            //Replace i with the imports that i depends on
            I = I – {i} ∪ Extract(i, R).Imports;

            //Update Flattened import list
            Flattened.Imports = I;

            //Flatten Flattened with the interface named i in the repository R
            Flattened = Expand(Flattened, Extract(i, R));
            break:

      }
}
```

Figure 4.12 Flattening Template.

We illustrate the definition with the help of the following:

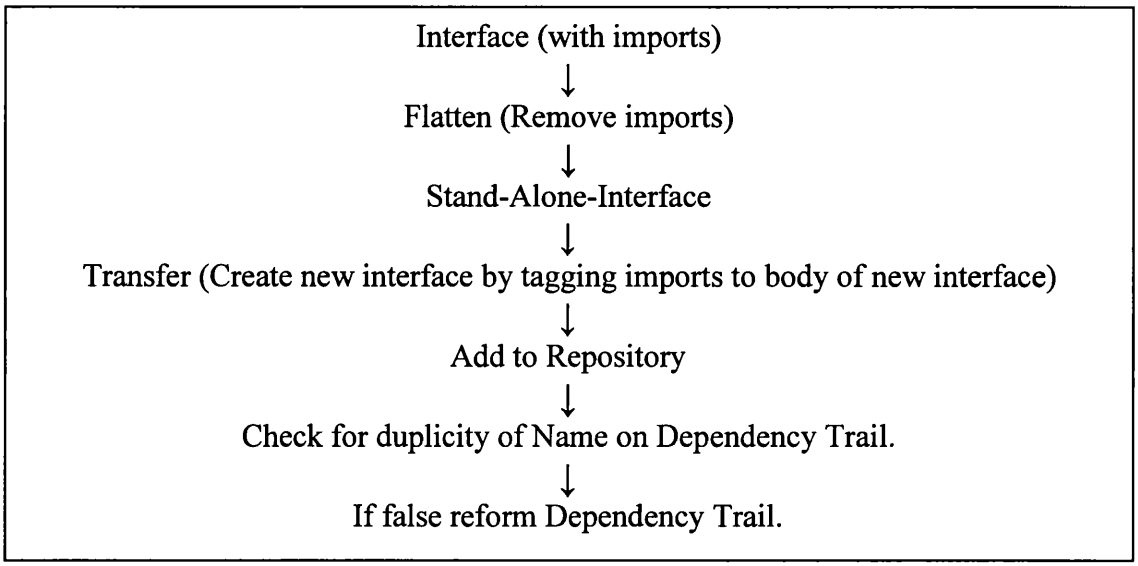


Figure 4.13 Reformation of Dependency Trail.

Full details of the recursion definition and flattening can be found in Rees *et al* [2003].

Chapter 5

A Subset of Java and its Interfaces

"Inside every large language is a small language struggling to get out..."

Attributed to C. A. R. Hoare.

Introduction

In this chapter we will use the general ideas on interfaces in Chapter 4, to explore the role of interfaces in the object-oriented language Java. We will test the application of the general interface model to notions of interface for Java. Do existing Java notions conform to the model? Is there a new interface notion, based on the model that makes sense in Java?

First, in Sections 5.1 – 5.3, we will consider a subset of Java called *Little Java,* based upon classes and inheritance. The idea is to consider the key features that can be used in Little Java without fear of degradation or conflict when comparing it with the larger language.

Secondly, in Section 5.4, we will introduce a small *abstract* object-oriented interface definition language called OO-IDL into which we translate the class constructs of *Little Java,*

$$T : \textit{Little Java} \rightarrow \textit{OO-IDL}.$$

The syntax and semantics of OO-IDL is shaped by the general model of interfaces in Chapter 4, and the syntax and semantics of algebraic specification languages. Indeed, we complete the semantic definition of OO-IDL by translating it into a model expressed in the interface definition language AS-IDL of algebraic specifications, via

$$M : \textit{OO-IDL} \rightarrow \textit{AS-IDL}$$

Of course, AS-IDL is the familiar language of algebraic signatures, and its semantics is given by interpreting signatures by algebras via some mapping

$$[\ ]: \mathit{AS\text{-}IDL} \rightarrow \mathit{Algebras}$$

that assigns to each signature some algebra or class of algebras; the methodology works for either choice. Thus, the range Algebras of [ ] could be either

(i) a class of algebras, or

(ii) a class of classes of algebras, respectively.

These classes of algebras could be derived from specifications, e.g. by taking initial or loose semantics of a system of axioms. For easy definitions we can assume it is a class of algebras of different signatures.

By combining these steps in this way,

$$[\ ] \bullet M \bullet T: \mathit{Little\ Java} \rightarrow \mathit{Algebras},$$

we give algebraic semantics for abstract object-oriented interfaces and, ultimately, for classes in Little Java: if class c $\in$ Little Java then it has a semantic model in, for example, the algebra

$$[M(T(c))] \in \mathit{Algebras}.$$

In keeping close to Java, with its size and syntactic structure, we encounter difficulties in defining the map T, formally and in general. We explain T via examples; the definitions of M and [ ] are less laborious.

Of special interest is what happens to the idea of modularity and flattening in these three languages. In this way, we make an algebraic model of some OO interface constructs as realised in Java and examine the concept of flattening.

Flattening enables the semantic definition [ ] to be simple and easy to understand. The languages OO-IDL and AS-IDL are based firmly on the model in Chapter 4.

5.1. Object-oriented Languages and Little Java

An object-oriented programming language is designed to emphasise a modular approach to programming in which software consists of units or components also programmed in the same language. The programming concepts that capture the idea of programming units or components are *class* or *object* and the program concepts that capture putting units together are connected with *inheritance.* In this section we enumerate all the essential features of object-oriented programming that we are to study, using *Little Java.*

In general, we wish to consider the interaction and behaviour of the following:

(i) one class and another,

(ii) data types,

(iii) how a class may extend another class,

(iv) how methods within a class interface with other classes and methods,

(v) how implementation of methods can return a value or return or *void*,

(vi) the interaction between the *interface* and other *interfaces*,

(vii) how an *interface* may extend and implement an *interface* or a list of *interfaces*.

5.1.1 Classes

The class construct has several sub-constructs which we consider in turn. These components are displayed below.

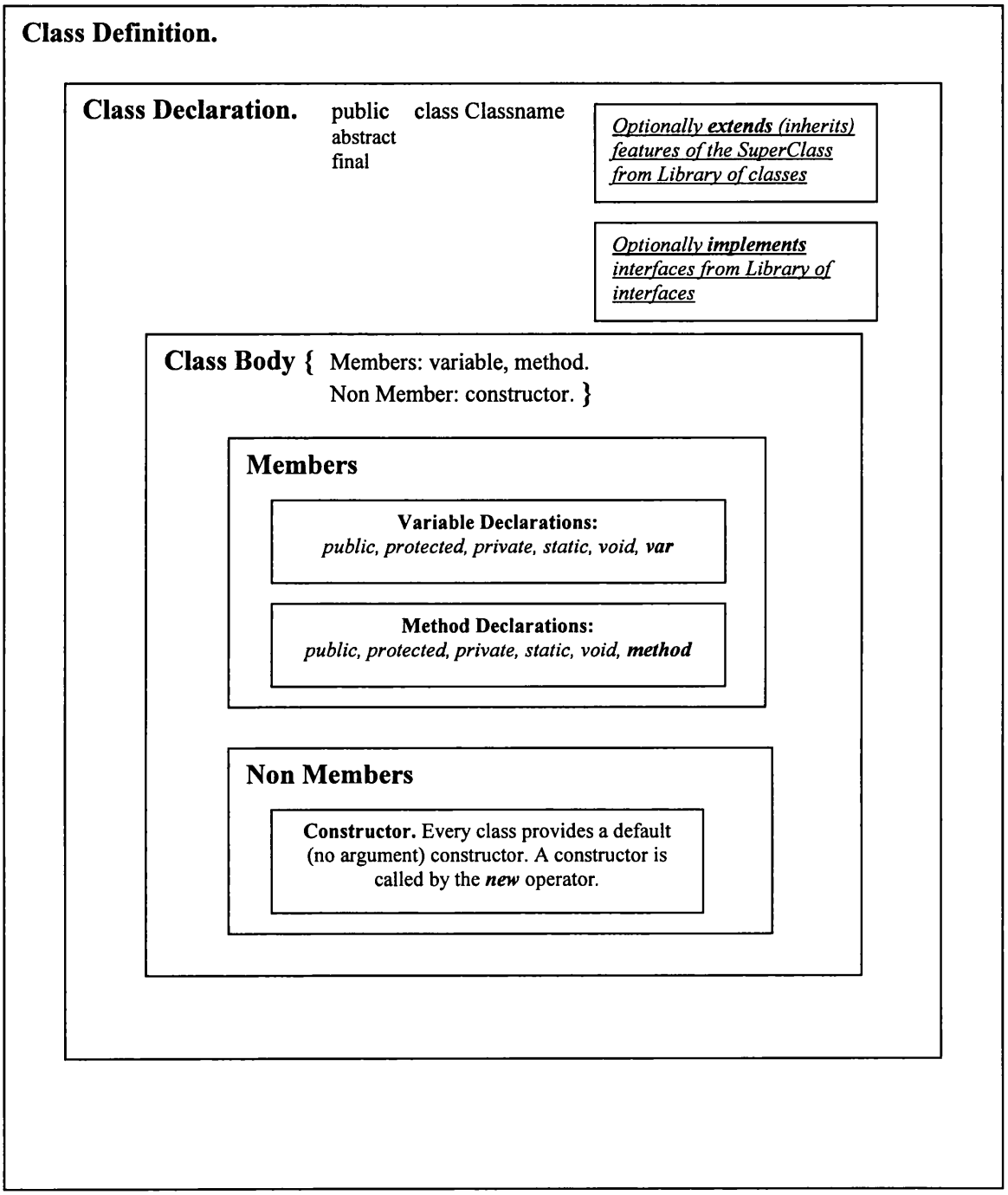


Figure 5.1 Class Schematic.

5.1.2 Class Definition

A class is a template that can be used to instantiate other objects. It is constructed from:

(i) a class declaration and

(ii) a class body.

In summary,

Class Definition = Declaration + Body.

5.1.3 Class Declaration

The declaration must state the name of the class and, optionally, declare its superclass with the keyword **extends** and, optionally, implement one, or more interfaces. (See Section 2.7).

The class declaration may optionally contain a modifier. It is placed before the class identifier in order to regulate access to any invocation instigated elsewhere in the program.

5.1.4 The Class Body

The class body is constructed of variable declarations and methods and contains the member variables and methods supported by the class. It can be described as a set of all *instances* of that pattern. There is a need to formulate a list of definitions and methods within the body of the class. (See Section 2.7.1). In summary,

Body = Declarations + Methods.

5.1.5 Member Variables

Collectively, static variables and instance variables are called *member variables*, or just *members*. Variables defined inside a method are called ***local***, temporary variables and **static final** variables are termed *constants.*

(i) Variables with the modifier **static** before them are part of the class in which they appear and are, therefore, called class variables. They are members to this class and no other class, e.g. **static,** int, num;

(ii) **static** *class variables* are allocated once for a class and are declared in the class body, not in a method;

(iii) **static final** *declaration,* e.g. **static final x = 3**. This declares that **x** is a *constant* and is applicable to the member class only.

5.1.6 A Method

In Java, classes use methods to communicate with objects. The method has two parts: the method declaration and the method body. The method declaration defines all of the method's attributes and the method body contains the Java instructions that implement the method. Java has an explicit case wherein a method may have a body but no instructions and no implementation:

$$Method = Declaration + Body.$$

5.1.7 Method Declaration

Method declarations describe code that may be invoked by method invocation expressions. A class method is invoked relative to the class type; an instance method is invoked with respect to some particular object that is an instance of the class type. A method, whose declaration does not indicate how it is implemented, must be declared **abstract**. A method may be declared **final**, in which case it cannot be hidden or overridden and no precedence is given to other methods to alter the behaviour of this particular method.

The method declaration may, optionally, contain a method modifier. It is placed before the method identifier in order to regulate method access, state or behaviour. The declaration must state a method name, the return type, the number and type of its arguments. Java insists on the return value of data type to be identical to the method declaration data type. Methods may return reference data types or primitive data types. If no return value is required, the keyword void must be placed before the method.

5.1.8 Method Body

The method body is constructed from

(i) variable declarations and,

(ii) statements.

The method body may contain local variables and methods supported by the class. Member variables can be static or non static. Methods can be declared in the same way.

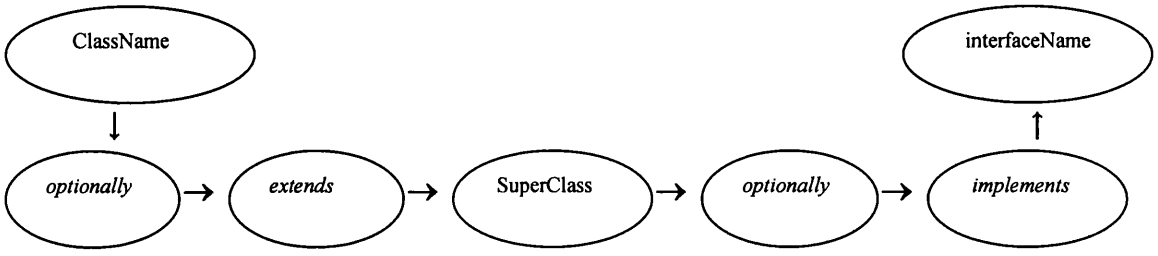
5.1.9 Constructor

A constructor is a part of a class and is used to initialize a new object of that class type. The class constructor always has the same name as the class and has no return type. A constructor uses its arguments to initialize the state of the **new** object. Java supports name overloading for constructors, i.e. a class can have any number of constructors with the same name.

When writing a class, the Runtime System automatically provides a constructor for that class if one has not been included. The compiler can determine which constructor to implement based on the number of arguments used.

5.1.10 Inheritance

In the Java language, an interface defines a set of methods. A class that implements an interface refers to the protocol defined by that interface. All objects in Java have state and behaviour. A blueprint of an object may be created by a class which can, further, define its data and behaviour. A class may inherit state and behaviour from only one other class, its Superclass. The class declaration must state the name of the class and may declare its Superclass with the keyword **extends**.



The keyword **extends** declares that the ClassName is the subclass of SuperClassName. A subclass inherits variables and methods, their state and behaviour, from the Superclass. The class inherits all the attributes of the Superclass, which it extends, but can, modify, or override, these attributes.

We include the **implement** clause keyword within the class declaration for the class implementation of an interface or interfaces. Multiple inheritances are not permissible within Java but the Java platform supports multiple inheritances when using classes with special interfaces. These special classes, called *interfaces,* have no implementation and no state and they, in turn, may optionally, implement one, or more, additional interfaces.

An interface is a collection of declared methods and constants. It does not provide implementation for these methods. The interface is constructed from:

(i) The **interface** declaration and,

(ii) The **interface** body

The keyword **implement** declares an interface or a list of interfaces. It may optionally contain a modifier which is placed before the interface identifier in order to regulate access by any invocation instigated elsewhere in the program. The declaration must state the name of the interface and optionally declare its SuperInterface with the keyword **extends**, and optionally implement one, or more, interfaces.

5.1.11 Library

To complete our lists of concepts we need to add the idea of a library of pre-existing classes that can be imported into the programmes of *Little Java.* For theoretical purposes, it does not matter too much what choice we make as long as we have some given classes to use.

5.1.12 Software Architecture in Little Java

We see component-based software as an extension of basic object-oriented software. It is clear that object-oriented principles can be used as a basis for the specification and

design of other architectural styles e.g. pipe filter architecture. (See Garlan and Shaw [1996] and Rees *et al* [2003]).

We are interested in modelling a simple notion of software architecture for object-oriented programmes in *Little Java*. In Java there is a specific notion of interface that is derived from the notion of class. A Java interface is a contract in the form of a collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface. Within the Java language an interface is a device that unrelated objects use to interact with each other. An interface is probably most analogous to a protocol; the behaviour of the interface may be implemented by any class, anywhere, in the class hierarchy. Thus, there is a notion of architecture derivable from Java.

$$\textit{Java Architecture} = \textit{Structured set of Java interfaces.}$$

$$\textit{Java program interface} = \textit{Package} + \textit{Imports} + \textit{Class Name} + \textit{Body.}$$

5.1.13 Grammar Listing

In Chapter 3 we gave a subset of Java designed to illustrate modular syntax techniques. We based the subset on JDK 1.1 and, subsequently, named it J1. *Little Java* is a subset of J1,

$$\textit{Little Java} \subseteq \textit{J1} \subseteq \textit{Java 1.1.}$$

We now list the syntactic features retained in our construct of *Little Java*. (See Gosling & McGilton [1996]).

Essentially we adopt the basic Java language fundamentals. The language itself is vast, particularly when you consider the enormous library of classes and methods that are available to the programmer. We look at the fixed Java structures and build on these to establish our subset. We will also show how the **import** keyword may be used to provide further classes and methods, if and when required, without compromising the language simplicity.

It should be accepted that all features not deemed to be essential to our simplistic approach should be left out of the language. We give some examples of these advanced features: overloading, messages to super, base types, null pointers, abstract method declarations, shadowing, access control (public, private, etc)., threads and exceptions. (See Green [1996-2005]).

We assume that the set of variables includes the special variable *this*, but that *this* is never used as the name of an argument to a method. Every class has a Superclass that we declare with the keyword **extends**. The type of an expression may depend on the type of any methods it invokes, and the type of a method depends on the type of expression within its body. (See Java Forums [1996-2005]).

With *Little Java*, rigorous arguments allow us to provide classes, methods, fields, inheritance, and dynamic typecasts, with semantics closely following that of Java. Little Java thus illustrates many of the interesting features of a working set of principles for the full language, while remaining efficient and compact. (See Sun Microsystems, Inc. [1995-2005]).

5.2 Detailed List of Constructs

We consider, in specific detail, a concrete syntax of *Little Java*. In the preparation of this subset we were influenced by Felleisen & Friedman [1998] and Igarashi, Pierce and Wadler [2001]. We begin with a basic program notion to write a typical class and then list the essential components of our simple Java1 subset, namely *Little Java.*

5.2.1 Types

(i) Fundamental primitive types: int, double, boolean.
(ii) The other primitive types: short, long, byte, char and float.

We refer to the table, Figure 3.6 in Chapter 3.2.1, to illustrate all fundamental types.

Type	Table Definition No 1.
Boolean	A boolean value, true or false
Char	16 bit Unicode character
Byte	8 bit signed two's complement integer
Short	16 bit signed two's complement integer
Int	32 bit signed two's complement integer
Long	64 bit signed two's complement integer
Float	32 bit IEEE 754 floating point value
Double	64 bit IEEE 754 floating point value

Types derived for *Little Java* listed in second table.

Type	Table Definition No 2.
Boolean	A boolean value, true or false
Char	16 bit Unicode character
Int	32 bit signed two's complement integer
Float	32 bit IEEE 754 floating point value
Double	64 bit IEEE 754 floating point value

5.2.2 Operators

(i) Arithmetic operators: + , -, *, /, %
(ii) Increment/decrement operator: ++, --
(iii) The assignment operator =
(iv) The combined arithmetic/assignment operators +=, -=, *=, /=, %=
(v) Relational operators ==, !=, <, <=, >, >=

5.2.3 Logical

Logical operations &&, ||, !

5.2.4 String concatenation

+

5.2.5 Arrays.

(i) Arrays: One dimensional arrays and two dimensional rectangular arrays are part of the 'Little Java' subset.
(ii) Both arrays of primitive types (e.g. **int[]** and arrays of objects. Initialisation of named arrays **(int[] a = { 1, 2, 3 };)**

5.2.6 Control Structures.

(i) **if, if else,**

(ii) **while, for, return.**

5.2.7 Modifiers

(i) The accessor modifiers **public, protected, private** and **final.**
(ii) The modifier **void** that indicates that no return value is expected.
(iii) The modifier **Static** that denotes a *class variable* or *class method.*

5.2.8 Null

Null is a reference and is part of the *Little Java* subset. It is not a keyword but is classed as a special literal of the *null* type.

5.2.9 This

The use of *this* is restricted to passing the implicit parameter in its entirety to another method (e.g. obj.method (*this*)) and to descriptions such as "the implicit parameter *this*".

5.2.10 Super

The use of the keyword, *super*, is restricted to invoking a superclass constructor super(args), e.g. *super. superClassMethodName( ).*

5.2.11 Initializer

We implement constructors that initialize all instance variables. Class constants are initialised with an *initializer.*

5.2.12 Library

The Java language allows the programmer to import classes, interfaces and their respective methods, from a comprehensive library. The library components may be accessed by a process called importing. These lists, or packages, are made available to a programmer by using the **import** keyword at the beginning of a Java program.

Thus, **Java.lang.String** is a class called **String** in the package **java.lang**. All classes in this package extend the immutable object class Java.lang.object.

5.2.13 Java.lang

Java.lang.String class	The String class represents character strings. Strings in Java are sequence of characters similar to characters in this paragraph. The matched brackets indicate that an array of Strings is required. An array is a linear collection of these characters and the name args is given to this array. This name part of the main declaration can vary.
Java.lang.System class	The System class contains several useful class fields and methods. It cannot be instantiated (extend). Among the facilities provided by the System class are standard input, output streams; access to externally defined "properties"; a means of loading files and libraries and a utility method for quickly copying part of an array.

5.2.14 Input and Output

In Java, you need to have a method named main in at least one class. An example of the syntax is found below.

Java.lang.System	The System class contains several useful class fields and methods. It cannot be instantiated (extends). Among the facilities provided by the System class are standard input, output and error output streams; access to externally defined "properties" and a means of loading files and libraries.

An example of how the System class *main* method could be used within a class.	public static void main(String [ ] args)
	{
	{ String message = "Hello";
	System.out.println (message);
	}
	}

We now define the following components as a subset of our earlier Java language namely, J1. Our new subset is called *Little Java*. A table is drawn to illustrate the library architecture.

Selected methods for *Little Java*, are to be found in Java.lang.String:

(i) boolean compareTo(Object other)
(ii) boolean equals(Object other)
(iii) int length()
(iv) String substring(int from, int to)
(v) String substring(int from)
(vi) int indexOf(String s)

5.2.15 Java.lang.Math

(i) static int abs(int x)

(ii) static double abs(double x)
(iii) atic double pow(double base, double exponent)
(iv) static double sqrt(double x)

Java.lang.Math	The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

5.2.16 Java.lang.Object

(i) boolean equals(Object other)
(ii) String toString(Object other)

Java.lang.Object	In Java the Object class is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

We have mentioned previously that the Java Core Application Programming Interface *(package java.io)* specifies a list of class interfaces and explicit interfaces for the purpose of manipulating data streams.

In the Java programming language, the names of classes that are defined inside packages always start with the package name. Classes in the same package are automatically imported, as are the classes in the java.lang package. For all other classes, you must supply an import statement to import a particular class e.g. *import java.awt.Rectangle*; or to import all classes in a package, using the on demand notation *import java.awt.**. The *java.io* package is an ideal example of a repository of related interfaces.

Here are a few of the many packages available in the Java Platform that could be useful as libraries to *Little Java.*

(i) **Java.io** is the package for reading and writing (input and output).

(ii) **Java.applet** provides the classes necessary to create an applet and the classes an applet uses for communication purposes.

(iii) **Java.awt** is the package that contains all of the classes for creating user interfaces and for painting graphics and images.

5.3 Examples of Language Features and Flattening in Java

We make reference to Chapter 5, Section 5.1.10 which deals with Java inheritance and show three example programmes that will help us to understand the meaning of Java Class inheritance in terms of flattening.

5.3.1 The Base Class

In Figure 5.2 we define the straightforward and simple class, BaseClass.

```
import java.util.ArrayList;
public class BaseClass {
        protected ArrayList list = new ArrayList();
        public void add(String s) {
                list.add(s);
        }
        public String get(int i) {
                String s;
                try {
                     s = (String)list.get(i);
                }
                catch (Exception e) {
                      s = "Error";  return s;
                }
        }
        public void delete(int i) {
                try {
                    list.remove(i);
                }
                catch (Exception e) {
                     s = "Very Bad"; return s;
                }
        }
}
```

Figure 5.2 A Java Program Demonstrating the Base Class.

5.3.2 The SubClass

In our second example we define class SubClass and invoke the Java keyword **extends** in order that BaseClass inherits the properties and attributes of the parent class.

```
Import java.util.ArrayList;

public class SubClass extends BaseClass {

        public void clear() {

                list.clear()

        }

}
```

Figure 5.3 The Sub Class, or Extended Base Class.

Our SubClass is termed the subclass of our Superclass (BaseClass) with all the attributes of the 'BaseClass'.

5.3.3 The Flattened Class

The flattening process involves the removal of imports. If we remove these imports from SubClass, certain inherited attributes *extended* from BaseClass will be lost. Specifically, variables will no longer be accessible to the SubClass. In order to correct this we combine the first two classes, without imports, to a third program, FlatClass as illustrated in Figure 5.4.

The FlatClass functions in exactly the same way as the previous versions with imports but the declarations of both classes are combined into the one class.

```
import java.util.ArrayList;
public class FlatClass {
        private ArrayList list = new ArrayList();
        public void add(String s) {
                list.add(s);
        }
        public String get(int i) {
                String s;
                try {
                        s = (String)list.get(i);
                }
                catch (Exception e) {
                        s = "Error"; return s;
                }
        public void delete(int i) {
                try {
                        list.remove(i);
                }
                catch (Exception e) {
                        s = “Error”; return s;
                }
        }
        public void clear() {
                list.clear();
        }
}
```

Figure 5.4 The Flattened BaseClass / SubClass.

5.3.4 Class Signatures

In Java we can abstract from programmes the components that are important for interfaces, using class signatures and interfaces (5.3.5). The three classes Base, Sub and Flat class can be stripped down to simple declarations of method names as follows:

```
Public class BaseClass{

        public void add(String s){
        }
        public String get(int i){
        }
        public void delete(int i){
        }
}
```

Figure 5.5 The BaseClass.

```
Public class SubClass extends BaseClass{
        public void clear( ) {
        }
}
```

Figure 5.6 The SubClass.

```
Public class FlatClass {
        Public void add( String s){
        }
        } public String get(int i){
        }
        public void delete(int i){
        }
        public void clear( ) {
        }
}
```

Figure 5.7 The Flat Class.

These signature classes contain the names of key components of our three classes. This brings us closer to the level of abstraction we need for our IDL models.

5.3.5 Java Interfaces

The interface construct of Java also allows us to write these declarations:

```
interface Base {
        public void add(String s);
        public String get(int i);
        public void delete(int i);
}
```

Figure 5.8 The Base Interface.

```
interface Class extends Base {

        public void clear();
}
```

Figure 5.9 The Class Interface.

```
interface Flat {

        public void add(String.s);

        public String.get(int i);

        public void delete(int i);

        public void clear();

}
```

Figure 5.10 The Flattening Interface.

5.4 An Abstract Object-oriented IDL

We have stripped down Java to *Little Java* which is based on the constructs of classes, inheritance, and a library. In this section we reflect on those three concepts and create a simple abstract model of them. This model is an abstract IDL called *OO-IDL*. Moreover, we focus on the role of interfaces in their use. We begin by looking at this idea of classes and their interfaces.

Methods can only be created as part of a class. Sometimes, named methods have typed parameters that return values. The objects, or interfaces, declare interaction between components in the manner described above. The method within that class(object) may be called from other instantiated objects.

We have defined the Java methods earlier in 5.1 but we now differentiate between kinds of methods.

(i) *Command methods*,

(ii) *Query methods.*

These methods are rarely distinguished in working languages but have a quite distinct semantical behaviour that should be visible in an interface.

Firstly, we deal with command methods. Commands simply change the state of an implementation depending upon the values of some parameters. As functions they have the form: the i-th command is

$$Com_i : state \times r^i_1 \times \ldots \times r^i_{k(i)} \rightarrow state$$

Secondly, queries return a value, or values, as well as change the state of an implementation. Thus, as a function they have the form: the j-th query is

$$QmState_j : state \times r^j_1 \times \ldots \times r^j_{l(j)} \rightarrow state \times r^j$$

This can be unpacked into its co-ordinate functions.

$$QmState_j : state \times r^j_1 \times \ldots \times r^j_{l(j)} \rightarrow state$$

$$QmData_j: state \times r^j_1 \times \ldots \times r^j_{l(j)} \rightarrow r^j$$

Thus, the general form of an interface in OO-IDL is given in Figure 5.11.

interface	Body OO-IDL interface with commands/queries
import	
sorts	..., *s*, ...
constants	
operations	..., f_i: $s_1 \times \ldots \times s_n \rightarrow s$, ...
declarations	..., d_i, ...
methods	
(commands)	$mCom_1$: $t_1^1 \times \ldots \times t^1_{k(1)} \rightarrow void$
	.
	.
	.
	$mCom_p$: $t_1^p \times \ldots \times t_{k(p)}^p \rightarrow void$
(queries)	$mQuery_1$: $r^1_1 \times \ldots \times r^1_{l(1)} \rightarrow r^1$
	.
	.
	.
	$mQuery_q$: $r^q_1 \times \ldots \times r^q_{l(q)} \rightarrow r^q$
endinterface	

Figure 5.11 OO-IDL.

5.5 Transforming Little Java into OO–IDL

When mapping *Little Java* programmes to the abstract forms of OO-IDL we

(i) make explicit all implicit notions of dependency and inheritance;

(ii) classify each method as a command or query;

(iii) map inheritance, **extends** in *Little Java*, to **import**.

We use the program examples of BaseClass (Figure 5.2), SubClass (Figure 5.3) and FlatClass (Figure 5.4) to illustrate our OO – IDL translation. Java modifiers such as **public, static** and **void** are used to control usage of class expressions and assignments during the running of these program but do not alter their state. In order to simplify our transformation it would be acceptable, semantically, to omit these modifiers when mapping to our signature.

The revised BaseClass program components before transformation.

interface	BaseClass
sorts	..., *s*, ... , *string, int*
constants	
operations	
methods	
(commands)	*add* : *string* → *void*
	delete : *int* → *void*
(queries)	*get* : *int* → *string*
endinterface	

Figure 5.12 The BaseClass Interface.

The revised SubClass program components before transformation.

interface SubClass

imports *BaseClass*

sorts

constants

operations

methods

commands : *clear : → void*

Figure 5.13 The SubClass Interface.

The revised FlatClass program components before transformation.

interface Flattening

sorts *int, String*

constants

operations

methods

commands) *add : string → void*
delete : int → void
clear : → void
(queries) *get : int → string*

endinterface

Figure 5.14 The FlatClass Interface.

5.6 Algebraic Specification Interface Definition Language AS-IDL

There are many algebraic specification languages. They have much in common because they are well founded semantically on a small collection of precise mathematical concepts, most notably many sorted algebras and axiomatic theories. Axiomatic theories are used to specify a system component and the algebras represent possible models. Many early languages, such as the OBJ family, emphasised initial algebras and term rewriting.

The Common Algebraic Specification Language (CASL) was designed by the Common Framework Initiative (CoFI), for algebraic specification and research. It emphasises loose semantics and theorem proving. (See Mosses [2004]).

The simple notion of many sorted signatures which play a basic role in all aspects of algebraic specifications is, in fact, a precise concept of an interface for data types and systems modeled by algebras. Thus, an algebraic specification language with its axiomatic theories removed is an interface definition language. Furthermore, it is an IDL that is relatively easy to understand and is, in fact, equipped with the same basic theory.

Of greater importance for our investigation is the fact that we can

(i) model anything using algebras, and

(ii) understand this IDL

Our simple IDL for signatures has three components

(i) signatures,

(ii) inheritance based on imports and flattening,

(iii) a library.

5.7 Translation of OO-IDL to AS-IDL

When translating from OO-IDL to AS-IDL we

(i) make explicit the role of the state of the class and

(ii) model the behaviour of their methods by functions on state and parameters.

In the transition from Little Java to OO_IDL we introduce distinctions between method commands and queries. The mapping to signatures is quite direct and we have the correlation.

OO_IDL		*Signature*
p method commands	→	*p ops*
q method queries	→	*2q ops*

For example, the general form for an OO-IDL in Figure 5.11 translates into

signature	Translated Body of OO-IDL interface-Commands/ Queries
sorts	$state, \ldots, s, \ldots$
constants	$c: \rightarrow s, \ldots$
operations	$f_i: s_1 \times \ldots \times s_n \rightarrow s, \ldots$
(methods)	
(commands)	$mCom_1: state \times t_1{}^1 \times \ldots \times t_{k(1)}{}^1 \rightarrow state$
	.
	.
	.
	$mCom_p: state \times t_1{}^p \times \ldots \times t_{k(p)}{}^p \rightarrow state$
(queries)	$mQ_1: state \times r^1{}_1 \times \ldots \times r^1{}_{l(1)} \rightarrow r^1$
	$mQState_1: state \times r^1{}_1 \times \ldots \times r^1{}_{l(1)} \rightarrow state$
	.
	.
	.
	$mQ_q: state \times r^q{}_1 \times \ldots \times r^q{}_{l(q)} \rightarrow r^q$
	$mQState_q: state \times r^q{}_1 \times \ldots \times r^q{}_{l(q)} \rightarrow state$
endinterface	

Figure 5.15 Body of OO-IDL interface-Commands / Queries.

Here are translations on three OO-IDL components in Fig 5.12, Fig 5.13 and Fig 5.14.

The OO-IDL given in Figure 5.12 becomes:

signature	BaseClass	
sorts	*string, int, state*	
constants		
operations		
(commands)	*add*	*: state × string → state*
	delete	*: state × int → state*
(queries)	*get*	*: state × int → state*
	get	*: state × int → string*
endinterface		

Figure 5.16 Revised example of OO-IDL Interface-BaseClass.

The OO-IDL given in Figure 5.13 becomes:

signature	SubClass	
import	*BaseClass*	
sorts	*string, state*	
constants		
operations		
(commands)	*clear*	*: state → state*
endinterface		

Figure 5.17 Revised example of OO-IDL Interface-SubClass.

The OO-IDL given in Figure 5.14 becomes:

signature	FlatClass	
sorts	*string, int, state,*	*..., s, ...*
constants		
operations		
(commands)	*add*	*: state × string → state*
	delete	*: state × int → state*
	clear	*: state → state*
(queries)	*get*	*: state × int → state*
	get	*: state × int → int*
endinterface		

Figure 5.18 Revised example of OO-IDL Interface-FlatClass.

Now we are able to use the standard semantic methods for abstract data types to give a semantic model for OO-IDL and, hence, *Little Java* programmes.

$$J \rightarrow T(J) \rightarrow MT(J)$$

We define semantics of *Little Java* J to be [[MT(J)]] which is some algebra or class of algebras as we discussed in the introduction of Chapter 5. We have, also, to determine the expressive power of a language in which inheritance is defined by flattening. We suspect it may be as expressive as Java.

Chapter 6

Conclusions and Further Work.

This thesis investigates two theoretical aspects of the formal definition of programming languages, using case studies in Java. First, we define modular grammars which can be used to decompose large grammars. Modular grammars allow the modular definition of formal languages. They provide concepts of component and architecture for grammars and languages. We show that this modular method can be used to define a modern practical language like Java.

Second, we describe recent general work on the definition of interfaces and interface definition languages (IDLs). In Rees, Stephenson and Tucker [2003], there is an analysis of the idea of interfaces and an algebraic model of a general IDL. We apply these ideas to analysing aspects of interfaces in Java. These ideas extend the methods used in Stephenson & Tucker [2006].

This latter task is more complicated and, in conclusion, we reflect on the method here as it leads to ideas for further research.

In Chapter 5, we attempted to implement the results of research into the general form of interface definition languages, discussed in Chapter 4. We defined 'Little Java', a subset of the programming language Java, and endeavoured to describe a series of translations from 'Little Java' to an abstract object-oriented interface definition language OO-IDL and, thence, to an interface definition language AS-IDL for abstract data types. The AS-IDL can be given its algebraic semantics in a number of well-understood ways.

Consider the process of translation in separate stages. First, the aim of the translation from Little Java into OO-IDL is represented by the following commutative diagram, where flattening is preserved.

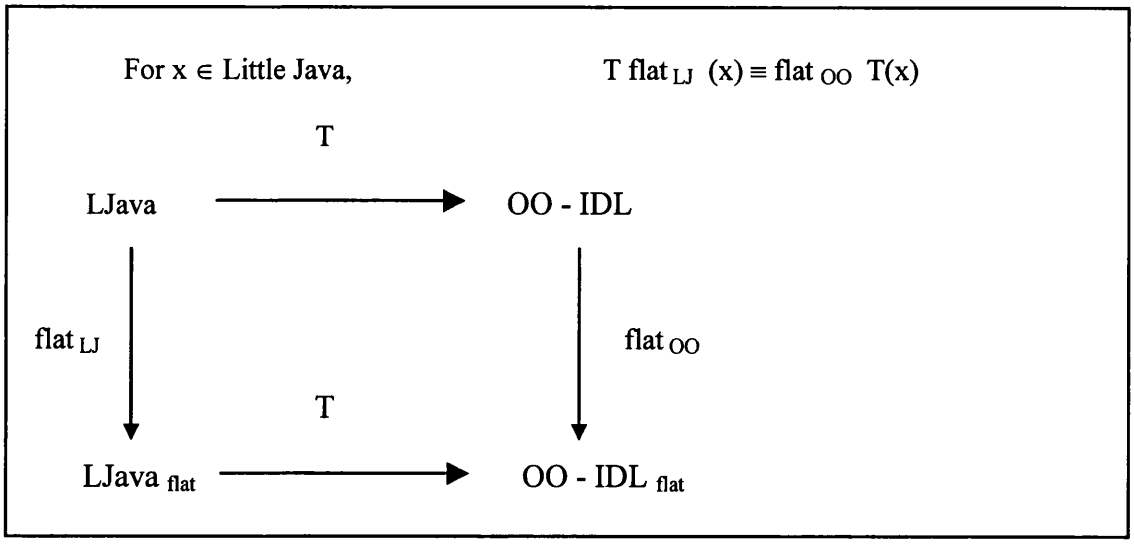


Figure 6.1 Fragment of Java.

Second, the aim of the translation from OO-IDL into AS-IDL is represented by the following commutative diagram, again in which flattening is preserved and some form of algebraic semantics is chosen.

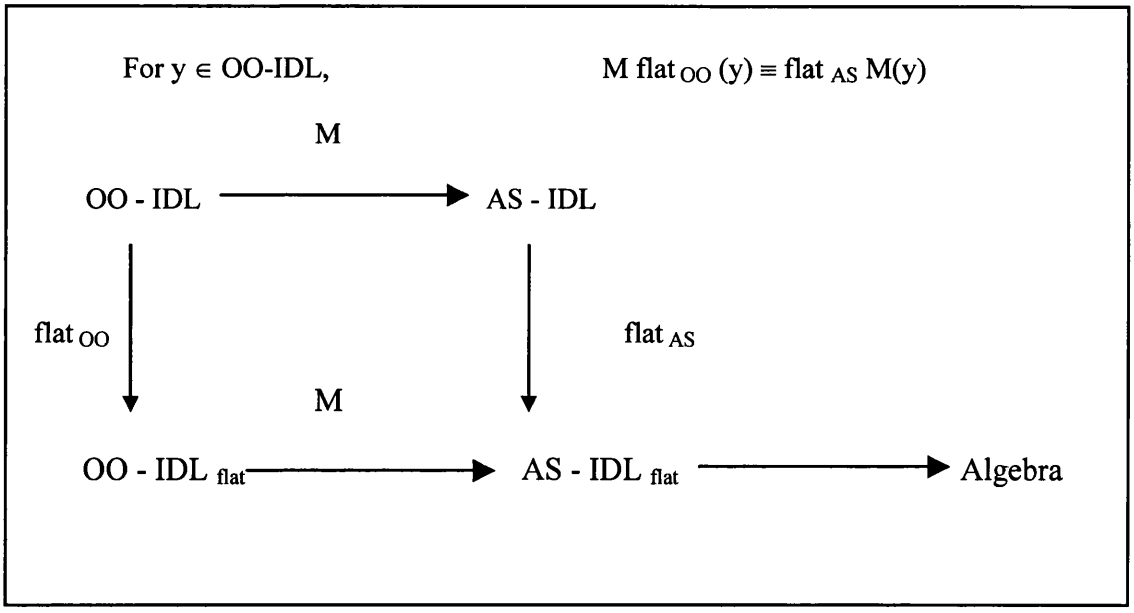


Figure 6.2 Abstract Object Orientated IDL.

Putting the two steps together we have the following commutative diagram, in which flattening is preserved.

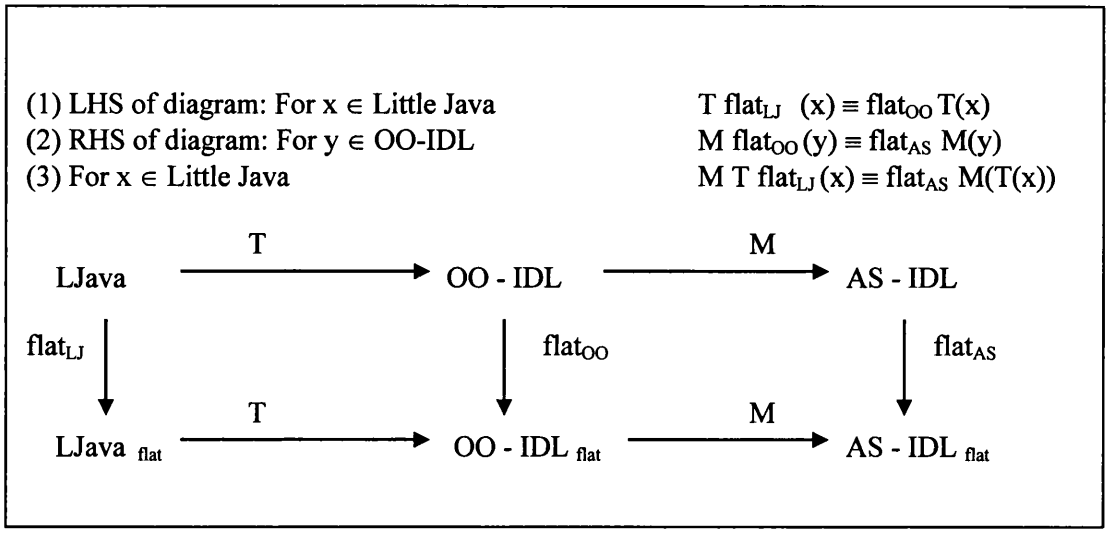


Figure 6.3 The Big Picture / Theoretical Framework.

We think that our work shows that there is a case for saying that the algebraic model of Rees, Stephenson and Tucker [2003] can capture the very basic structure of class interfaces in a simple subset of Java. For such a simple language a series of translations can result in an algebraic semantics for such a subset of the Java language. We have not used the algebraic structures of the interface model in Chapter 4 to define these translations but we believe it could be done by structural inductions. We are content to explore and demonstrate the feasibility of the general ideas. It would be interesting to determine the expressive power of a language in which an inheritance is defined purely by flattening. We suspect it may be as expressive as Java with its more complex inheritance.

We could investigate, further, the semantics of class notions in Java, and other languages such as Eiffel and C$^{\#}$. For example, missing features such as **public** and **private** could be included in OO-IDL and AS-IDL. The basic methodology of this thesis should be applicable to other languages and selected further constructs, i.e. the technique of mapping a fragment, *Little L,* of an OO language *L,* to OO-IDL and then to AS-IDL, in such a way that some form of inheritance native to *L* is preserved, should work:

$$L \rightarrow OO\text{-}IDL \rightarrow AS\text{ - }IDL$$

In particular, if OO-IDL proves to be stable and robust under translation for other languages, and its mapping

$$M : OO\text{-}IDL \rightarrow AS\text{-}IDL$$

models, semantically, features relevant to a variety of languages, then our concept and methods are useful.

Chapter 7

Thesis Summary and Evaluation.

In the first year of research, in consultation with Dr Stephenson, it was decided that, in view of my interest and background, the research topic would be based on the theoretical aspects of the syntax and semantics of the Java programming language.

On reflection, we would, perhaps, re-arrange certain sections to give us a clearer picture of how we apply the transfer of algebraic rules to Little Java. This has been achieved in some areas. For example, Chapter 4 sets out certain rules and illustrations, in a hierarchical manner, prior to the introduction of Chapter 5.

In Chapter 2, we write of *'building blocks* that have *interfaces* which give rise to a *modular* or *hierarchical structure'* and then give interpretations of object-oriented development in different languages before itemising the basic features of the Java language. We considered, in general, that the thesis would investigate certain theoretical aspects of the formal definition, or specification, of programming languages. We considered aspects of syntax and semantics and centered our study on the object-oriented language, Java, and adopted its methods as a base concept for our case study. Our work, in this area, was influenced by the authors of software architecture such as Dhal and Nygaard, (See Dhal and Nygaard [1966]), Garlan and Shaw, (See Garlan and Shaw [1996]), Bass *et al*, (See Bass *et al* [1998]), Hayes-Roth, (See Hayes-Roth [1994]), Garlan and Perry, (See Garlan and Perry [1995]), Booch *et al*, (See Booch *et al* [1999]). Our example programmes, namely, Smalltalk, Eiffel and Python lead to the introduction of Java and the Virtual Machine-Java byte code.

In Chapter 3, we introduce Modular Grammars as a preparation to the modular decomposition of Java. We show the main language components of Java and explain their function and relationship with other components. We illustrate this by the principal of logical decomposition. This, together with our list of modular grammars, illustrates the basic structure of the large language. This interpretation contributed to the construction of our subset, *Little Java*.

Particular attention was paid to the overall theme of modularity, hierarchical structure, and architecture of languages. Subsequently, we employed a process of simplification on these section headings with the aid of simple theoretical tools.

For syntax we would consider modular grammars and Backus–Naur Form and use them to give a decomposition of Java syntax. More specifically, we examine examples and case studies in Java together with a modular construction of a subset of the language.

In Chapter 4, we write a preparation for IDL. For semantics we considered the mathematical modelling based on the concept of interfaces and their semantic applications. We developed modular grammars and applied them to large, current and real programming languages.

Our paper gives algebraic specifications of libraries of interfaces. We explain the importance of the dependency trail and its data dependencies, and the properties of the interface. We added the import definition and dealt with the imported interface, dependency, redundancy and the technique known as flattening. We investigated some theoretical concepts such as modular grammars, abstract interfaces and flattening, etc, and their application to larger languages.

Our aim would be to illustrate the meaning of modularity and flattening in these three languages. The general concepts on interfaces, discussed in Chapter 4, have a bearing on the work outlined in Chapter 5. We defined a subset '*Little Java*' from the concrete syntax of Java and provided a simplified interface definition language with certain omissions. These omissions, already listed, would not compromise the full Java language. The process of translating this subset to an abstract object-oriented interface definition language, OO-IDL and, thence, to an interface definition language AS-IDL for abstract data types (in our case, algebraic signatures into mapping). The mapping examples in Chapter 6 help to illustrate the theory.

What have we written? We have selected an object oriented language, namely, Java and analysed its content and form and interpreted it in a theoretical and modular manner by decomposition. We have introduced the principle of interfaces and the removal of imports, resulting in 'flattening'. Finally, we have translated from 'Little Java' to an

abstract object-oriented interface definition language OO-IDL and, thence, to an interface definition language AS-IDL for abstract data types.

We would anticipate that in this changing scientific environment, certain sections of the thesis could prove invaluable to future study, if only for its attempt to evaluate the transition of an imperative language (albeit a subset of that language) to an OO – IDL.

We can only accept that other people's interpretation of the subject matter of the thesis would vary enormously. We would be extremely pleased to learn, from prospective readers, their views and suggestions, but this is unlikely. Nevertheless, we hope they find the thesis to be informative and of educational worth.

The history of the Java language is compelling and interesting. We thought it was important that we include a summarised account of its origin and its subsequent development. The revised history is listed in the Appendix.

Appendix 1

We outline the history of the Java language and pay particular attention to the early years, its inception and its gradual progress due primarily, and in no small measure, to the rapid growth of the World Wide Web (Internet).

The switch program example is illustrated in Appendix 2.

A.1.1 A Revised History of the Java Language

This is an excerpt from James Gosling's account of the History of Java.

This research paper is predominately linked with the language Java and we, therefore, discuss the importance of its early history and how, and why a certain company gathered a dedicated group of individuals to research the possibility of creating an interface capable of communicating with the various network protocols in existence at that time; an interface which would work on a common communication platform and capable of meeting the demands presented by these systems.

We write about the instigation of this language and the reasons for its inception. The following paragraphs help to illustrate the fundamental issues involved and the history and development of the Java Language.

(See Gosling [1996]).

A.1.2 A Brief History of the Internet and Related Networks

In 1973 an American project was instigated by the Government Defence Agency to research the techniques and technologies of packet networks. The main objective was to undertake a study on communication protocols and subsequently formulate a system of networked computers, capable of communicating, with one another, transparently. As a direct result of this study and research a system, named simply, the 'Internet' was born. Two of the protocols to evolve from this research are:

(iii) *TCP/IP Protocol Suite: Transmission Control Protocol.*

(iv) *(TCP) and Internet Protocol (IP).*

In 1986, the U.S. National Science Foundation (NSF) initiated the development of the NSFNET which, today, provides a major backbone communication service for the Internet. With its 45 megabit per second facilities, the NSFNET carries on the order of 12 billion packets per month between the networks it links. The National Aeronautics and Space Administration (NASA) and the U.S. Department of Energy contributed additional backbone facilities in the form of the NSINET and ESNET respectively. In Europe, major international backbones such as NORDUNET and others provide connectivity to over one hundred thousand computers on a large number of networks. Commercial network providers in the U.S. and Europe are beginning to offer Internet experience and access support on a competitive basis to any interested parties. (See Segal [1995]).

A.1.3 World Wide Web

Tim Berners-Lee, now Sir Tim Berners-Lee, the creator of the World Wide Web, first released in 1991, at CERN in Switzerland currently heading the World Wide Consortium, has shown great interest in a new concept called R.D.E. (Resources Definition Framework). The theory purports that it will allow software to travel through cyberspace adapting itself to various encountered conditions and behavioural situations, thereby performing tasks on behalf of the human user.

The idea has been termed 'The Semantic Way' and it is claimed that the concept will change the lives of the world population by turning the Internet into a place that is as intelligible for computers as it is for human beings. A frightening concept, perhaps, and one, which has been recognized by the likes of worldwide giants I. B. M., Hewlett Packard and Nokia as a worthwhile project that has prompted them to invest heavily in research. There are sceptics who state that it is an idea that will ultimately be destined for use by academics as a theory worthy of interest. This is merely conceptual but the proposal has its own supporters who see it as a tool capable of transforming the way in which we handle information on the Web. Berners-Lee saw the World Wide Web as a

massive portal of information available to anyone and capable of inter connection on a composite linkage platform.

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." (See Tim Berners-Lee, James Hendler, Ora Lassila [May 2001]).

The problem with these millions of pages of information and the millions of connections made by computers is that they are stored in a format that is intelligible to human thought and understanding. This is difficult for computers that have no concept of natural languages and is one of the reasons why so many link operations (non-computer aided) have to be carried out by hand.

The race is on to bring about a change in the way we store our information. If computers are to take on the task of interpreting the content of files and act accordingly, then the contents have to be coded in a way that will facilitate this interaction. This will, ultimately, allow computers to read directly without interpretation.

The first-ever Millennium Technology Prize was today awarded to Tim Berners-Lee for the invention of the WWW service on the internet. The prize trophy, "The Peak", was presented by the President of The Republic of Finland, Ms Tarja Halonen, in Finlandia Hall. In his acceptance speech Tim Berners-Lee said"We must remember that the web is a long way from revealing its full potential. The extension from human-readable to include also machine-readable information is just one direction of development".

At this stage, and as an integral part of our research, it would be proper to introduce the contribution made to the development of the World Wide Web, by Tim Berners-Lee, a graduate of Queen's College Oxford in 1976. In 1980, whilst working as an independent software consultant, he wrote a program for storing information, using random associations. This program called "Enquire" was never published, but it proved to be the conceptual basis for further development of the World Wide Web. He continued his research, working with various bodies such as CERN (European Organisation for Nuclear Research), and in 1989, he proposed a global hypertext project, to be known as

the World Wide Web. This project was started in October 1990 and was operational, and on the Internet by the summer of 1991. (See Campbell-Kelly and Aspray [1996]).

In 1994, Tim Berners-Lee founded the World Wide Web Consortium at the Massachusetts Institute of Technology. Since that time he has served as its Director and his aim has been: *"to lead the Web to its full potential, ensuring its stability through rapid evolution and revolutionary transformations of its usage."* The Consortium may be found at http://www.w3.org/.

Various research and educational institutions, together with regional and local bodies, formed networks. In the early days, a great deal of support originated from the United States and its federal and state governments. It must be emphasised, however, that industrial bodies have made an enormous contribution to the building of the network in the early days. This is true in all major countries throughout the world, particularly Europe. Towards the end of 1991, the Internet had grown to over 5,000 networks and by 1994 approximately thirty six countries, serving over 700,000 host computers and 4,000,000 users, were in operation. Figures taken from *'A Brief History of the Green Project'*. (java.sun.com/people/).

The continued rise in population of Internet users and the relative system networks grew internationally and globally to include commercial, private, educational, research, business and government organizations across the world. The recent proliferation of web sites attests to the rapid growth and success of the Internet. Designed to provide information, showcase products and services, and form the basis of an electronic marketplace, each web site can be broken up into multiple pages of information, each identified by a URL. Today, there are millions of URLs in use throughout the world.

A.1.4 The Internet Technical Evolution

The foundation for the information superhighway has been laid. The Internet - which interconnects thousands of public and private networks worldwide - today provides millions of users with access to information from around the globe. This complex web of networks forms the pathway for a global information revolution that will eventually link businesses, public and private agencies, and educational centers with one another.

A.1.5 Mosaic

To address this need Netscape Communications Corporation was founded in April 1994 by Dr. James H. Clark (Silicon Graphics) and Marc Andreessen, (NCSA Mosaic Software and Graphical User Interface). The development of Netscape continued and the innovative Netscape Navigator client software was eventually available for download over the Internet.

A.1.6 Java People

In 1991, under the guidance of James Gosling of Sun, a small research development team was commissioned by Sun to evaluate concepts that were compatible with "next wave" development in computing. The emphasis was to be centered on digitally controlled consumer devices and computers.

In the summer of 1992 after a concentrated effort, lasting over eighteen months, the Green Team (as they were now called) had perfected an interactive, handheld home-entertainment device controller with an animated touch screen user interface. The appliance, named Star Seven, was capable of controlling a wide range of appliances, and at the same time, displayed animation using a new, processor-independent language. The new language, pioneered by James Gosling, was christened "Oak," after the tree outside his window.

A.1.7 First Person

Sun turned the Green Team into a separate company and re-named it FirstPerson. The new company was instructed to find a market for their innovative device and they targeted, initially, the Cable companies, hoping to initiate the new interactive software into their cable networking systems. This, unfortunately, did not prove fruitful and they needed to look elsewhere for recognition. Within the three days, John Gage, James Gosling and colleagues decided that the Internet, already growing in popularity, was the obvious choice for the type of network configuration they were seeking and not the cable TV industry as they had originally thought.

The Internet and its inherent technology matched the technology of Java. They possessed the same parallel capability of moving media content across networks. It also offered the capability to move "behaviour" in the form of applets along with the content.

Gosling explains: "*We had already been developing the kind of `underwear' to make content available at the same time the Web was being developed. Even though the Web had been around for 20 years or so, with FTP and telnet, it was difficult to use. Then Mosaic came out in 1993 as an easy-to-use front end to the Web, and that revolutionized people's perceptions. The Internet was being transformed into exactly the network that we had been trying to convince the cable companies they ought to be building. All the stuff we had wanted to do, in generalities, fit perfectly with the way applications were written, delivered, and used on the Internet. It was just an incredible accident. And it was patently obvious that the Internet and Java were a match made in heaven. So that's what we did.*" (See Gosling [Java.sun.com]).

When two or more networks are joined together it becomes an internet. In the year 1994 the Internet connected, roughly, 60,000 independent networks into a vast global internet. It was a widely used means of moving media content throughout this network utilising HTML. Hyper Text Markup Language is used, internationally, to publish hypertext on the World Wide Web. It is a non-proprietary format based upon the descriptive Standardised Markup Language SGML, and can be created and processed by a wide range of text editors, simple or sophisticated.

In that same year using Mosaic as a template, the team, subsequently, developed a demonstration program that was capable of object animation within a Web browser. The resultant program was named "WebRunner" and was the forerunner of the HotJavaTM browser. A year later the improved program was demonstrated to a Technology conference. The audience witnessed the moving text images for the first time and was quick to realize the immediate potential of the new technology. The WebRunner binary code was released over the Internet for developers and anyone interested in their work. The response was amazing and in a few months the number of downloads had reached over 10,000. This, together with the ever increasing number of e-mails amounting to thousands each day, was handled by the dedicated and overworked team to the best of

their ability. The success of the project had generated such an enormous reaction that it was rapidly becoming unmanageable. "They simply saturated the line," said Gosling. Sun, the originators of the project, committed themselves to the new Java Technology. This act, in itself, was regarded by the team as monumental, but what followed, was to prove even more momentous.

On May 23, 1995, John Gage, director of the Science Office for Sun Microsystems, and Marc Andreessen, co-founder and executive vice president at Netscape, announced that JavaTM technology was officially in being and was to be incorporated into Netscape Navigator™. Java technology was created as a programming tool by Patrick Naughton, Mike Sheridan, and James Gosling of Sun in 1991. The original members of the Java technology team numbered less than thirty people but this small group created and developed a technology that would greatly influence the computing world. Judging by the outcome of this research, they had vindicated themselves admirably.

Unexpectedly the news that Netscape's Marc Andreessen had signed an agreement to integrate Java technology into the Navigator browser surprised the Java team. They, together with the rest of the world, learned the facts when Andreessen and a Sun executive appeared briefly on stage to cement the deal with a handshake.

A recent survey by Matthew Gray [2003], gave the number of host users as over 170,000,000. We are lead to believe that most, if not all, of these would have Java technology implementation to some degree. (*Internet Statistics* [http://www.isc.org/]).

Some of these implementations are listed:

JDK, the sandbox, applets, thousands of Java technology-oriented startups, over a thousand books on Java technology, JavaBeans architecture, Java Studio, Netscape Communicator, Internet Explorer, various search engines, Internet service providers, 170 million Internet users, 56K and cable modems, broadband, electronic commerce, servlets, Java Foundation Classes, Enterprise JavaBeansTM components, Swing, JavaOS for BusinessTM, and commitments from major players such as IBM.

Within two years, the JavaOne conference had attracted 10,000 developers. In the third year, the renamed Java technology, now Sun Java Software Division employed 800 people, a few more than the original Green Team.

Since that introduction in May 1995, the Java platform has been adopted more quickly across the industry board than any other new technology in computing history. It seems that when people see that a particular product has development potential they make a concerted effort to further that development. Java has successfully exhibited its prowess by turning static Web pages into interactive, dynamic, animated documents. It can also boast that it works with distributed platform-independent applications. Since its debut, it has taken the international community by storm and enhanced Web browsers, almost everywhere, with animation, audio, video and real-time interactivity.

Appendix 2

Switch Program Example

We include this example to further illustrate the work covered in 3.2.9. The *switch statement*, in the Java language, is used to conditionally perform statements based on an integer expression. The following sample program, SwitchDemo, declares an integer, *month* whose value represents the name month in a calendar. The program displays the name of the month, based on the designated integer, using the *switch* statement:

```
public class SwitchDemo {

  public static void main(String[] args) {

    int month = 8;

    switch (month) {

      case 1:  System.out.println("January"); break;

      case 2:  System.out.println("February"); break;

      case 3:  System.out.println("March"); break;

      case 4:  System.out.println("April"); break;

      case 5:  System.out.println("May"); break;

      case 6:  System.out.println("June"); break;

      case 7:  System.out.println("July"); break;

      case 8:  System.out.println("August"); break;

      case 9:  System.out.println("September"); break;

      case 10: System.out.println("October"); break;
```

```
        case 11: System.out.println("November"); break;
        case 12: System.out.println("December"); break;
      }
   }
}
```

The *switch* statement evaluates its expression, in this case the value of month, and executes the appropriate *case* statement. Thus, the output of the program is: August. Of course, you could implement this by using an *if statement*:

```
int month = 8;
if (month == 1) {
   System.out.println("January");
} else if (month == 2) {
   System.out.println("February");
}
. . . // and so on
```

Deciding whether to use an *if statement* or a *switch statement* is a matter for the programmer who needs to observe the need for clarity and functionality. An *if statement* can be used to make decisions based on ranges of values or conditions, whereas a switch statement can make decisions based only on a single integer value. Also, the value provided to each *case statement* must be unique.

Another point of interest in the switch statement is the *break statement* after each case. Each break statement terminates the enclosing switch statement, and the flow of control continues with the first statement following the switch block. The break statements are

necessary because without them, the case statements fall through. That is, without an explicit break, control will flow sequentially through subsequent case statements. Following is an example, *Switch Demo*, which illustrates why it might be useful to have case statements fall through:

```
public class SwitchDemo2 {

  public static void main(String[] args) {

    int month = 2;
    int year = 2000;
    int numDays = 0;
    switch (month) {
      case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
        numDays = 31;
        break;
      case 4:
      case 6:
      case 9:
      case 11:
        numDays = 30;
        break;
      case 2:
        if ( ((year % 4 == 0) && !(year % 100 == 0))
            || (year % 400 == 0) )
          numDays = 29;
        else
          numDays = 28;
```

```
            break;
        }
        System.out.println("Number of Days = " + numDays);
    }
}
```

The output from this program is:

```
Number of Days = 29
```

Technically, the final break is not required because flow would fall out of the *switch statement* anyway. However, we recommend using a *break* for the last *case statement* just in case there is need to add further case statements at a later date. This makes code modification easier and less error-prone. The keyword *break* is used to terminate loops in *Branching Statements*.

Finally, the *default* statement can be used at the end of switch statements to handle all values that aren't explicitly handled by one of the case statements.

```
int month = 8;
. . .
switch (month) {
   case 1:  System.out.println("January");break;
   case 2:  System.out.println("February");break;
   case 3:
System.out.println("March"); break;
   case 4:
System.out.println("April"); break;
   case 5:
System.out.println("May"); break;
   case 6:
System.out.println("June"); break;
   case 7:
System.out.println("July"); break;
   case 8:
```

```
System.out.println("August"); break;
   case 9:
System.out.println("September"); break;
   case 10:
System.out.println("October"); break;
   case 11:
System.out.println("November"); break;
   case 12:
System.out.println("December"); break;
   default: System.out.println("That is not a valid month!"); break;
}
```

Bibliography

Backhouse R. C. [1979].

R. C. Backhouse, *Syntax of Programming Language, Theory and Practice*, Prentice–Hall International, London, 1979.

Backus J. [1960].

J. Backus, *Syntax of the proposed Algebraic Language of the Zurich ACM-GAMM Conference.* In *Proceedings of an International Conference on Information Processing,* UNESCO, Paris, 1959. Butterworth, London, 1960.

Bass *et al.* [1998].

Len Bass, Paul Clements and Rick Kazman, *Software Architecture in Practice*, Addison Wesley, 1998.

Berard E. [1989].

Edward V. Berard, *The Object Agency, Inc.* Web-site at: http://www.toa.com/pub/oodarticle.htm 1989.

Berners-Lee *et al.* [2001].

Tim Berners-Lee, James Hendler, Ora Lassila, *The Semantic Web*, Scientific American, May 2001.

Booch *et al.* [1999].

Grady Booch, Rumbaugh, and Jacobson. *The UML Modelling Language User Guide.* Addison-Wesley, 1999.

Booch. [1999].

Grady Booch, *Object-Oriented Analysis and Design with Applications,* Addison-Wesley, September1999.

Borland. [2003].

Budi Kurniawan "Java for the Web" 2004.

Brachla G & Griswold D. [1993].

Gilad Bracha & David Griswold, Strongtalk: type checking Smalltalk in a production environment, In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications,* p.215-230, September 26-October 01, 1993, Washington, D.C., United States

Buril C. W. [1967].

C. W. Buril, *Foundations of Real Numbers*, McGraw-Hill, 1967.

Buschmann *et al.* [1996].

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, & M. Stal. *A System of Patterns.* John Wiley & Sons, 1996.

Campbell-Kelly M. [1982-1996].

M. Campbell-Kelly. The Development of Computer Programming in Britain (1945 to 1955) *Annals of the History of Computing,* 4(2):121-139,1982.

M. Campbell-Kelly and William Aspray, *Computer: A History of the information Machine*, Basic Books, 1996.

Campione & Walrath. [1996].

M. Campione & K. Walrath. *The Java Tutorial*, Addison – Wesley, 1996.

Campione M., Walrath K. & Alison Huml A. [2001].

M. Campione, K. Walrath & Alison Huml, *The Java Tutorial*, Third Edition Addison – Wesley, 2001.

Chomsky N. [1956].

Noam Chomsky, *Syntactic Structures*, Mouton & Co. 1957.

Chomsky N. [1956].

IRE Transactions on Information Theory, IT-2 (3):113 {124,1956. Cohn, 1982.

Dahl & Nygaard. [1966].

Johan Dahl and Kristen Nygaard, *The Simula Language*, 1966.

Eiffel Software. [1985].

http://www.eiffel.com/index.html

Fact Guru. [2005].

Fact Guru Object-oriented Software Engineering [2005]. Website:

http://www.site.uottawa.ca:4321/oose/index.html - Javawrapperclass

Felleisen M. & Friedman P. [1998].

Matthias Felleisen and Daniel P. Friedman, *A little Java, A Few Patterns*, The MIT Press, London. [1998]

Flanagan D. [1996].

D. Flanagan, *Java in a Nutshell*, O'Reilly, 1996.

Garlan and Perry. [1995].

David Garlan, Dewayne Perry. *Introduction to the Special Issue on Software Architecture.* IEEE Transactions on Software Engineering, 21(4), Apr. 1995.

Garlan and Shaw. [1996].

David Garlan and Mary Shaw, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

Goguen J. *et al.* [1977].

J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, *Initial Algebra Semantics and Continuous Algebras*. Journal of the ACM, 24: 68, 95, 1977.

Goldberg & Robson. [1983].

J. Goldberg and D. Robson, Smalltalk-80: *The Language and Its Implementation,* Longman Publishing Co., MA, 1983.

Gosling J. [1996].

James Gosling, *Java Application Programming Interface,* Java Team, 1996.

James Gosling & Henry McGilton, *The Java Language Environment,* 1996.

Website at:

http://www.javaworld.com/javaqa/2002-07/02-qa-0719-multinheritance.html

Gray M. [1993-95].

Matthew Gray, Measuring the Growth of the Web, Massachuessets Institute of Technology, 1993-95, Build a Web Site, Prima Publishing, 1995.

Green R. [1996-2005].

Roedy Green , Java Glossary, Canadian Mind Products, 1996-2005.

Website: http://mindprod.com/jgloss/super.html

Hayes-Roth F. [1994].

Frederick Hayes-Roth, "*Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program,*" Teknowledge Federal Systems. Version 1.01 February 4, 1994.

Horstmann C. [1996].

G. Cornell & C. Horstmann. *Core Java.* Sunsoft Press, 1996.

ICEIMT. [1997].

International Conference on Enterprise Integration and Modelling Technology,

Website: http://www.mel.nist.gov/workshop/iceimt97

Igarashi A., Pierce B. and Wadler P. [2001].

Atsushi Igarashi, Benjamin Pierce and Philip Wadler, *Featherweight Java: A Minimal Core Calculus for Java and GJ,* ACM Transactions on Programming Languages and Systems, Vol. 23, No. 3, May 2001, Pages 396–450 [2002].

Website: http://www.eecs.umich.edu/~bchandra/courses/papers/Igarashi_FJ.pdf

Jackson & McClennan. [1997].

J. Jackson & A. McClennan, *Java by Example*, Sunsoft Press, 1997.

Java Forums. [1996-2005].

Developers Java.Sun, Website:

http://forum.java.sun.com/thread.jspa?threadID=557037&start=360

Kay A. [1996].

Alan Kay, *The Early History of SmallTalk*, in Bergin, Jr., T.J., and R.G. Gibson. *History of Programming Languages - II,* ACM Press, New York NY, and Addison-Wesley Publ. Co., Reading MA 1996, pp. 511-578, with additional commentary and transcripts

Website: http://en.wikipedia.org/wiki/Alan_Kay

Kleene S. [1909-1994].

Stephen C. Kleene, *Mathematical Logic*, New York, John Wiley, 1967.

Litwak K. [2000].

Kenneth Litwak, *Pure Java™ 2*, Sams Publishing, 2000.

Meinke & Tucker [1993].

K. Meinke & J. Tucker, *Many Sorted Logic and its Applications*, J. Wiley & Sons, 1993.

Merrick & Allen [1997].

P. Merrick & C. Allen, *Web Interface Definition Language,* W3C 1997.

Meyer B. [1997].

Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall Technical Reference, 1967. Bertrand Meyer, *Object-oriented Software Construction, Second Edition*, Upper Saddle River, NJ: Prentice-Hall PTR, 1997.

Mosses P. D. [2004]

Peter D. Mosses, Editor *CASL Reference Manual,* LNCS 2960 (IFIP Series), Springer, 2004. XVII + 528 pages.

Naur P. [1960].

Peter Naur , J. W. Backus , F. L. Bauer , J. Green , C. Katz , J. McCarthy , A. J. Perlis , H. Rutishauser , K. Samelson , B. Vauquois , J. H. Wegstein , A. van Wijngaarden , M. Woodger, *Report on the Algorithmic Language ALGOL 60*, Communications of the ACM, v.3 n.5, p.299-314, May 1960 .

Neumann [1995].

P. Neumann, *Computer Related Risks*, Addison-Wesley, 1995.

Niemeyer et al. [1996].

Patrick Niemeyer & Josh Peck, *Exploring Java*, O'Reilly, 1996.

O.M.G. [2003].

The Object Management Group. Website: http://www.omg.org/

Python Programming Language [2005].

Python Programming Language, *Object-oriented Programming.*

Website: http://en.wikipedia.org/wiki/Python_programming_language#Object-oriented_programming

Rees, Stephenson and Tucker. [2003].

D. Ll. Rees, K. Stephenson & J. V. Tucker, *The Algebraic Structure of Interfaces*, Science of Computer Programming, Elsevier, 2003.

Sebasta Robert W. [1989].

Robert W. Sebesta, *Programming the World Wide Web 2003*, Addison Wesley 1989.

Segal B. [1995].

Ben Segal, *A Short History of Internet Protocol at CERN*, CERN IT-PDF-TE 1995.

Website: http://ben.home.cern.ch/ben/TCPHIST.html

Smalltalk [1970 -2005].

Smalltalk Programming Language, *Object-oriented Programming*.

Website: http://en.wikipedia.org/wiki/Smalltalk

Stephenson K. [2002].

Karen Stephenson, *An Algebraic Approach to Syntax, Semantics and Compilation,* University of Wales, Swansea, 2002.

Stephenson and Tucker [2006].

Karen Stephenson & J. V. Tucker, *Data, Syntax & Semantics,* University of Wales, Swansea, 2006.

Sun. [2000].

Sun Microsystems Inc., *A Grammar for the Java Programming Language*, 2000.
Sun Microsystems Inc. *Remote Method Invocation(RMI),* 2003.

Website: http://java.sun.com/products/jdk/rmi/

Van der Linden. [1996].

Peter van der Linden, *Just Java*, Sunsoft Press,1996.

Walden and Nerson. [1994].

Kim Walden and Jean-Marc Nerson, *Seamless Object-oriented Programming,* Prentice-Hall, Reference and Method Book on BON, 1994.