1 *Introduction to Object Orientation*

1.1 Introduction

This book is intended as an introduction to object orientation for software engineers and others who are actively involved in the software industry. It assumes familiarity with standard computing concepts such as stacks and memory allocation, and with a procedural language, such as C. From this background, it provides a practical introduction to object technology using Java, one of the newest and best pure object-oriented languages available.

This book introduces a variety of concepts through practical experience with an object-oriented language. It also tries to take you beyond the level of the language syntax to the philosophy and practice of object-oriented development.

In the remainder of this chapter, we will consider the various programming paradigms that have preceded object orientation. We will then examine the primary concepts of object orientation and consider how they enable object orientation to be achieved.

1.2 Programming Paradigms

Software construction is still more of an art than a science. Despite the best efforts of many software engineers, software systems are still delivered late, over budget and not up to the requirements of the user. This situation has been with us for many years. Indeed, the first conference to raise awareness of this problem was the NATO Software Engineering Conference of 1968, which coined the term *software crisis*. Since then a variety of programming paradigms have either been developed explicitly to deal with this issue or have been applied to it.

A programming paradigm embodies a particular philosophy. These philosophies usually represent an insight which sets a new type of best practice. For a programming language to support a particular paradigm, it must not just allow adoption of the paradigm (you can use object-oriented programming techniques in assembler, but would you want to?), it must actively support implementations based on the paradigm. This usually means that the language must support constructs which make development using that paradigm straightforward.

The major programming paradigms which have appeared in computer science can be summarized as follows:

- *Functional* Lisp is the classic example of a functional language, although by no means the only one (ML is a very widely used functional language). These languages place an emphasis on applying a function (often recursively) to a set of one or more data items. The function then returns a value – the result of evaluating the function. If the function changes data items, this is a side effect. There is limited support for algorithmic solutions which rely on repetition via iteration. The functional approach turned out to be an extremely useful way of implementing complex systems for early AI researchers.
- *Procedural* Pascal and C exemplify procedural languages, which attempt to move to a higher level than the earlier assembler languages. The emphasis is on algorithmic solutions and procedures which operate on data items. They are extremely effective, but software developers still encounter difficulties. This is partly due to the increased complexity of the systems being developed, but also because, although high-level procedural languages remove the possibility of certain types of error and increase productivity, developers can still cause problems for themselves. For example, the interfaces between different parts of the system may be incompatible, and this may not become obvious until integration or system testing.
- *Modular* In languages such as Modula 2 and Ada, a module hides its data from users. The users of the module can only access the data through defined interfaces. These interfaces are "published" so that users know the definitions of the available interfaces and can check that they are using the correct versions.
- *Object-oriented* This is the most recent "commercial" programming paradigm. The object-oriented approach can be seen as taking modularization a step further. Not only do you have explicit modules (in this case, objects), but these objects can inherit features from one another. We can of course ask "why another programming paradigm?". The answer to this lies partly in the failure of many software development projects to keep to budget, remain within time-scales and give the users what they want. Of course, it should not be assumed that object orientation is the answer to all these problems; it is just another tool available to software developers.

This book attempts to introduce the object-oriented programming paradigm through the medium of an object-oriented programming language. It assumes that the majority of readers have a background in at least one procedural language (preferably a C-like language), and compares and contrasts the facilities provided by an object-oriented language with a procedural language.

Object orientation, even though it is quite different in many ways from the procedural approach, has developed from it. You should therefore not throw away all that you have learnt using other approaches. Many of the good practices in other languages are still good practices in an object-oriented language. However, there are new practices to learn, as well as new syntax, but remember that it is much more than a process of learning a new syntax – you have a new philosophy to learn.

1.3 Revolution Versus Evolution

In almost every area of scientific endeavour there are periods of evolution followed by periods of revolution and then evolution again. That is, some idea or theory is held to be “accepted” (not necessarily true, but at least accepted). The theory is refined by successive experiments, discoveries etc. Then the theory is challenged by a new theory. This new theory is typically held by a small set of extremely fervent believers. It is often derided by those who are staunch supporters of the existing theory. As time continues, either this new theory is proved wrong and disappears, or more and more people are drawn to the new theory until the old theory has very few supporters.

There are many examples of this phenomenon in science: for example, the Copernican theory of the Earth rotating around the Sun, Einstein’s theory of relativity and Darwin’s theory of evolution. Men such as Darwin and those who led him to his discoveries were revolutionaries: they went against the current belief of the times and introduced a new set of theories. These theories were initially derided, but have since become generally accepted. Indeed, Darwin’s theories are now being refined further. For example, Darwin believed in a mechanism for the fertilization of an egg derived from an old Greek theory (pangenesis). Every organ and tissue was assumed to produce granules which combined to make up the sex cells. Of course, we now believe this to be wrong and it was Darwin’s own cousin, Francis Galton, who helped to disprove the pangenesis theory. It is unlikely that we will enter a new revolutionary phase which will overturn the theory of evolution; however, Einstein’s theory of relativity is already being challenged.

Programming paradigms provide another example of this cycle. The move from low-level to high-level programming was a revolution (and you can still find people who will insist that low-level machine code programming is best). Object orientation is another revolution, which is still happening. Over the past ten years object orientation has become much more widely accepted, and you will find many organizations, both suppliers and users of software, giving it lip service. However, you will also find many in the computer industry who are far from convinced. A senior colleague of mine once told me that he believed that object orientation was severely over-hyped (which it may be) and that he really could not see the benefits it offered. I hope that this book will convince him (and others) that object orientation has a great deal to offer.

It is likely that something will come along to challenge object-oriented programming, just as it challenges procedural programming, as the appropriate software development approach. It is also likely that a difficult and painful battle will ensue, with software suppliers entering and leaving the market. Many suppliers will argue that their system always supported approach *X* anyway, while others will attempt to graft the concepts of approach *X* onto their system. When this will happen or what the new approach will be is difficult to predict, but it will happen. Until then, object orientation will be a significant force within the computer industry.

1.4 Why Learn a New Programming Paradigm?

The transition from a procedural viewpoint to an object-oriented viewpoint is not always an easy one. This begs the question “why bother?”. As you are reading this book you must at least be partly convinced that it is a good idea. This could be because you have noticed the number of job advertisements offering employment for those with object-oriented skills. However, that aside, why should you bother learning a new programming paradigm?

I hope that some of the reasons will become clear during your reading of this book. It is worth considering at least some of the issues at this point.

1.4.1 Software Industry Blues

There is still no silver bullet for the problems in the software industry. Object-oriented technology does not remove the problems of constructing complex software systems, it just makes some of the pitfalls harder to fall into and simplifies traditionally difficult problems. However, difficulties in software development are almost inevitable; many of them arise due to the inescapable intangibility of software, and not necessarily all by accident or poor development methods.

We should not, however, just throw up our hands and say "well if that's the case, it's not my fault". Many of the problems which beset our industry relate to some deficiency in how programmers build software today. For example, if a software development project runs late, then merely adding more people to it is likely to make matters worse rather than get the project back on schedule.

Object technology is not the first attempt at addressing these issues. However, past attempts have met with mixed success for a number of reasons, some of which we consider below.

Modularity of Code

Traditional procedural systems typically relied on the fact that not only would the data they were using not change its type, for example, but the way in which they obtained that data would not alter. Invariably, the function (or functions) that used the data also directly obtained the data. This meant that if the way in which data was accessed had to change, all the functions which used that data had to be rewritten. If you have attended any sort of software engineering course, you will say that what was required was another function which obtained the data (it thus acted as an intermediary). This function could then be used in many different places. However, such application-specific functions tend not to get used in "real-world" systems for several reasons:

- *Small subroutines are too much effort.* Although many people talk about reusable code, they often mean relatively large code units. Small functions of one, two or three lines tend to be defined by a single programmer and are rarely shared among a development team, let alone several development teams.
- *Having too many subroutines leads to too little reuse.* The larger the number of subroutines available, the less likely that they are reused. It is very difficult to search through a code library of small subroutines trying to find one which does what you want. It is often much quicker to write it yourself!
- *It may not be obvious that a function is reusable.* If you are a programmer working on one part of a system, it may not be obvious that the function you are writing is of generic use. If a function is small then it is not identified by the designer as being a useful reusable component.

Ability to Package Software

Another issue is the way in which programming languages package up software for reuse. Many systems assume that the software should be partitioned into modules which are then integrated at compile time. Such fixed compile-time integration can be good for some types of problem, but in many cases it is too inflexible. For example, while this approach can ensure that the modules being reused are compatible, developers may not know until run time which modules they wish to use, and therefore some form of run-time binding is necessary.

UNIX pipes and filters are examples of software systems which can be bound at run time. They act as "glue", allowing the developer to link two or more programs in sequence together. However, in this case there is absolutely no error protection. It is quite possible to link two incompatible systems together.

What would be really useful would be a combination of these features. That is, the ability to specify either compile-time or run-time binding. In either case, there should be some form of error checking to ensure that you are integrating compatible modules. An important criterion is to avoid the need for extensive recompilation when, for example, just one line is altered. Finally, such a system should, by definition, enforce encapsulation and make packaging of the software effortless.

Flexibility of Code

In early procedural languages, for example C or Pascal, there was little or no flexibility. More recent procedural languages have introduced some flexibility, but need extensive specification to achieve it. The result is internal flexibility at the cost of interface overheads, for example in Ada. Object technology allows code flexibility (and data flexibility) with little overhead.

1.4.2 The Advantages Claimed for Object Orientation

There are a range of benefits which can be identified for object-oriented programming languages. Not all of these are unique to object-oriented technology, but that does not matter; we are talking about the good things about object orientation here:

- *Increased code reuse* Languages such as Java encourage reuse. Every time you specify that one class inherits from another (which you do all the time in Java), you are involved in reuse. In time, most developers actively look to see where they can restructure classes to improve the potential for reuse. As long as this is not taken too far, it is an extremely healthy thing to do.
- *Data protection for little effort* The encapsulation facilities provided as part of the language protect your data from unscrupulous users. Unlike languages such as Ada, you do not have to write reams of specification in order to achieve this protection.
- *Easier integration with encapsulation* As users of an object cannot access the internals of the object, they must go via specified interfaces. As these interfaces can be published in advance of the object being implemented, others can develop

to those interfaces knowing that they will be available when the object is implemented.

- *Easier maintenance with encapsulation* This point is really a variation on the last one. As users of an object must use the specified interfaces, as long as the external behaviour of these objects remains the same, the internals of the object can be completely changed. For example, an object can store an item of data in a flat file, read it from a sensor or obtain it from a database; external users of the object need never know.
- *Simplified code with polymorphism* With polymorphism, you do not need to worry about exactly what type of object is available at run time as long as it responds to the message (request for a method to be executed) you send it. This means that it is a great deal easier to write reusable, compact code than in many other languages.
- *More intuitive programming* It has been argued that object orientation is a more intuitive programming paradigm than other approaches, such as the procedural approach. This is because we tend to perceive the world in terms of objects. We see dials, windows, switches, fuel pumps and automated teller machines (ATMs). These objects respond to our use in specific ways when we interact with them. For example, an ATM requires a card, a PIN etc., in a particular sequence. Of course, those of us who have programmed before bring with us a lot of baggage, including preconceptions of what a program should be like and how you develop it. I hope that this book is about to turn all that on its head for a while, before putting everything back together again.

1.4.3 What Are the Problems and Pitfalls of Object Orientation?

No programming language is without its own set of problems and pitfalls. Indeed, part of the skill in becoming fluent in a new programming language is learning what the problems are and how to avoid them. In this section, we concentrate on the criticisms usually levelled at object orientation.

Lots of Confusing Terminology

This is a fair comment. Object orientation is littered with new terms and definitions for what appears to have been defined quite acceptably in other languages. Back in the early 1970s, when Smalltalk, one of the very first object-oriented programming languages, was being researched, many of the terms we now take for granted were already quite well established. However, Smalltalk introduced many of its own terms, which have remained in object-oriented languages to this day. It would be reasonable to assume that even if the inventors of the language liked their own terminology, early users would have tried to get it changed; but they didn't.

One possible answer is that in the past (that is, during the early and mid-1980s) object-oriented languages, such as Smalltalk, tended to be the preserve of academic and research institutions. (Indeed, I was introduced to my first object-oriented language while working on a research project at a British university during 1986–87.) It is often the case that academics enjoy the mystique that a language with

Table 1.1 Approximate equivalent terms

Procedural term	Object-oriented term
procedure	method
procedure call	message
non-temporary data	instance variable
record and procedures	object

terminology all of its own can create. By now, it is so well established in the object-oriented culture that newcomers just have to adapt.

The important point to remember is that the concepts are actually very simple, although the practice can be harder. To illustrate this, consider Table 1.1, which attempts to illustrate the parallels between object-oriented terminology and procedural terminology.

These approximations should not be taken too literally as they are intended only to help you visualize what each of the terms means. I hope that, by the end of the book, you gain your own understanding of their meaning.

Yet Another Programming Paradigm to Master

In general, people tend to like the things they are used to. This is why many people buy the same make of car again and again (even when it gives them trouble). It is also why computer scientists refuse to move to a new word processor, editor, operating system or hardware. Over the years, I have had many "discussions" with people over the use of LaTeX versus Word versus WordPerfect, the merits of Emacs versus vi, of UNIX versus Mac, or of Windows versus DOS. In most cases, the issues raised and points made indicate that those involved in the discussions (including me) are biased, have their own "hobby horse" to promote and do not understand fully the other approach.

Object orientation both benefits and suffers from this phenomenon. There are those who hold it up almost like a religion and those who cast it aside because it is so different from what they are used to. Many justify this latter approach by pointing out that procedural programming has been around for quite a while now and many systems are successfully developed using it. This is a reasonable statement and one which promotes the *status quo*. However, the fact that object orientation is a new software paradigm, quite different from the procedural paradigm, should not be a reason for rejecting it.

Object orientation explicitly encourages encapsulation (information hiding), promotes code reuse and enables polymorphism. Most procedural languages have attempted to present these advantages as well; however, they have failed to do so in such a coherent and concise manner. Ada, for example, is not only a large cumbersome language, it requires an extensive specification to be written to enable two packages to work together. Any error in these specifications and the system does not compile (even if there are no errors or incompatibilities in the code). Ada95 has introduced the concept of objects and classes, although, for most object technology practitioners, the way in which it has done this is both counterintuitive and unwieldy.

Many Object-Oriented Environments Are Inefficient

Historically, object-oriented development environments have been inefficient, processor-intensive and memory-hungry. Such environments tended to be designed for use on powerful workstations or minicomputers. Examples of such environments include Lisp Flavors (which even required specialist hardware, e.g. the Symbolics Lisp machine), Self and Smalltalk-80 (the forerunner of VisualWorks, a commercial Smalltalk development environment). These machines were expensive, sometimes non-standard and aimed at the research community.

With the advent of the PC, attempts were made to rectify this situation. For example, Smalltalk/V was designed specifically to run on the PC and the first version of Smalltalk that I used was on a 286 PC. The current versions of Java products, such as Symantec's VisualCafé and Borland's JBuilder, are now extremely efficient and optimized for use on PC platforms. Indeed, Sun's Java Development Kit (JDK) is available on UNIX, Solaris, Windows 95/98/NT and Mac OS.

Although 16 Mbyte of RAM is advisable on many of these systems, any 486 machine or above provides ample performance. The issue of additional RAM is not large; RAM can be purchased at reasonable rates and many industry pundits predict that 128 MByte (and more) will soon become the industry standard. Indeed, systems are now emerging which assume that a user has access to larger amounts of memory; for example, Microsoft's J++ requires a minimum of 24 MByte to run the debugger.

C++ and object-oriented versions of Pascal (such as Delphi) are no more memory- or processor-intensive than any non-object-oriented language. However, it is worth noting that these languages do not offer the same level of support for the programmer as, for example, Java and Smalltalk. In particular, they do not provide automatic memory management and garbage collection.

Java Environments Are not Suitable for Serious Development

Many object-oriented languages are interpreted, for example, Eiffel, Smalltalk, Objective-C and Java. If you intend to construct large applications in such languages you want to be sure that the performance of the resulting application is acceptable. In addition, for real-world applications you also wish to know that the facilities provided by the language, and any support environments, are up to the task.

We shall first consider the issue of compilation time. At present, all Java compilers produce an intermediate representation called byte codes, rather than a machine executable form. These byte codes are then interpreted by the Java Virtual Machine. The compiler available with Sun's JDK is relatively slow (and certainly slower than compiling comparable programs with Delphi or Visual Basic). However, vendors such as Symantec and Microsoft have developed Java compilers which are much faster (up to twice as fast).

The next issue is performance. The Java interpreter provided with the JDK is not particularly fast and is certainly a lot slower than a comparable C++ program. For applications of any size, this is a serious problem. What are really required are native code compilers for Java. Although some are being developed, these are not currently commercially available. However, companies such as Symantec and Microsoft have produced Just-In-Time (JIT) compilers which compile a piece of code once and then cache it. This means that any code which is executed repeatedly is converted into native machine code only once, resulting in a significant improvement in performance. Although there is still a run-time overhead, it is almost certainly faster than the standard Java interpreter, and some estimates suggest up to a 20-fold improvement.

The next issue is the application development facilities provided by the language. These can greatly affect the development time and the reliability and maintainability of a system.

For example, Smalltalk is a dynamically typed language, which has significant implications for large software development. However, the designers of the Java language took into account many of the problems which programmers have to deal with when developing large applications. In particular, they considered the problems associated with C++ programs (such as memory leaks) which can be extremely difficult to identify and debug. They added features to Java to remove the potential for introducing such bugs (for example, dynamic memory allocation and garbage collection). The Java language developers also considered the compilation of source code and dependencies between source code files. They provide an automatic compilation facility (achieving the same goal as a manually defined make file) to simplify application construction.

Finally, in the real world consideration must be given to issues such as portability (both to current hardware and to future hardware), distribution (as the move towards network computing grows, the network is the computer!) and evolution of the language (as new features are required). Once again, the Java language designers have considered these issues, and Java is "architecture neutral". It assumes nothing about the hardware on which it executes. The designers also made significant attempts to specify rigorously parts of the Java language which might be affected by different platforms. Thus the size of an integer in bytes is explicitly defined and the language explicitly incorporates facilities for dealing with TCP/IP protocols, URLs etc., allowing it to be seamlessly integrated with the Web. Packages and objects also allow new facilities to be added without significantly affecting existing ones.

1.5 Pedigree of Object-Oriented Languages

In the horse or dog breeding world, the pedigree of an animal can be determined by considering its ancestry. While you cannot determine how good a language is by looking at its predecessors, you can certainly get a feel for the influences which have led to the features it possesses. The current set of commercial object-oriented languages have all been influenced to a greater or lesser extent by existing languages. Figure 1.1 illustrates some of the relationships between the various languages.

Figure 1.1 only partially illustrates the family relationships as, for example, Ada95 should have a link from Smalltalk (or possibly C++). However, it attempts to illustrate the most direct influences evident in the various languages. The diagram is also ordered in roughly chronological order. That is, the further down the diagram a language appears, the more recent it is. This means, for example, that Smalltalk predates C++, and that Java is the most recent object-oriented language. Notice that

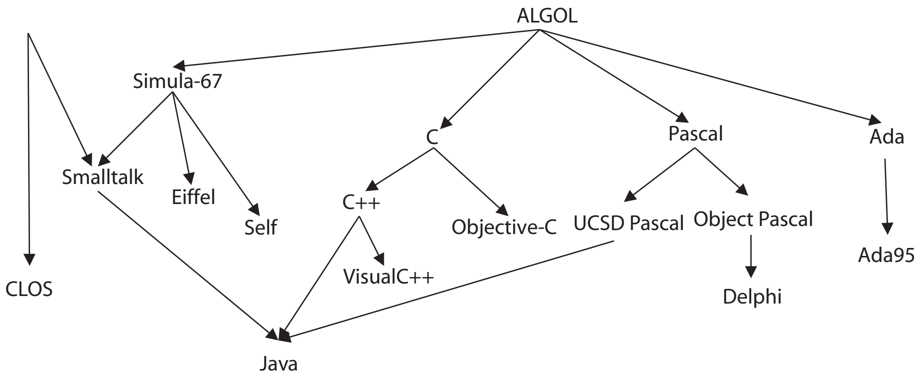

Figure 1.1 Partial Java family tree.

Lisp, ALGOL, C, Pascal and Ada are not object-oriented, and that Simula is, at most, object-based.

The extent to which a language can be considered to be a *pure* object-oriented language (i.e. one which adheres to object-oriented concepts consistently) as opposed to a *hybrid* object-oriented language (i.e. one in which object-oriented concepts lie alongside traditional programming approaches) tends to depend on its background.

A pure object-oriented language supports only the concept of objects. Any program is made up solely of interacting objects which exchange information with each other and request operations or data from each other. This approach tends to be followed by those languages which most directly inherit features from Simula (C++ is a notable exception). Simula was designed as a language for discrete event simulation. However, it was influenced by many of the features from ALGOL 60 and was effectively the first language to use the concepts which we now describe as object-oriented. For example, it introduced the concepts of class, inheritance and polymorphism.

The language which inherits most directly from Simula is Smalltalk. This means that its ALGOL heritage is there for all to see in the form of structured programming constructs (although the syntax may, at first, seem a little bizarre). It is a pure object-oriented language in that the only concepts supported by the language are object-oriented. It also inherits from Lisp (if not syntax, then certainly the philosophy). This means that not only does it not include strong typing, it also provides dynamic memory management and automatic garbage collection. This has both benefits and drawbacks, which we will discuss at a later stage. In contrast, Eiffel, another pure object-oriented language, attempts to introduce best software engineering practice, rather than the far less formal approach of Lisp. Self is a recent, pure object-oriented language which is still at the research stage.

Many language designers have taken the *hybrid* approach. That is, object-oriented constructs have either been grafted onto, or intermixed with, the existing language (for example, C++). In some cases, the idea has been to enable a developer to take advantage of object orientation when it appears appropriate. In other situations, it has eased the transition from one approach to another. The result has often been less than satisfactory. Not only does it mean that many software developers have moved to their new object-oriented language believing that it is just a matter of learning the new syntax (which it is not), they have written procedural programs in which objects are limited to holding data, believing that this is sufficient (which it is not). It is really only safe to move to a hybrid language once you have learnt about object technology using a pure object-oriented language.

1.6 Fundamentals of Object Orientation

The object-oriented programmer's view of traditional procedural programming is of procedures wildly attacking data, which is defenceless and has no control over what the procedures do to it (the rape and pillage style of programming). In contrast, object-oriented programming is viewed as polite and well-behaved data objects passing messages to one another, each data object deciding for itself whether to accept the message and how to interpret what it means.

The basic idea is that an object-oriented system is a set of interacting objects which are organized into classes. Figure 1.2 illustrates a simplified cruise control system from a car. It shows the objects in the system, the links between the objects and the direction in which information flows along these links. The object-oriented implementation of this system would mirror this diagram exactly. That is, there would be an object representing each box; between the boxes, there would be links allowing one object to request a service from, or provide information to, another. For example, the cruise control electronic control unit (ECU) might request the current speed from the speed sensor. It would then use this information when asking the throttle to adjust its position. Notice that we do not talk about functions or procedures which access information from data structures and then call other functions and procedures. There is no concept such as the ECU data structure and the ECU main program. This can be a difficult change of emphasis for some people, and we shall try to illustrate it further below.

The aim of object-oriented programming is to shift the focus of attention from *procedures that do things to data* to *data which is asked to do things*. The task is not to define the procedures which manipulate data, but to define data objects, their attributes and the way in which they may be examined or changed. Data objects (and procedures) can communicate with other data objects only through narrow, well-defined channels.

1.7 The Basic Principles of Object Orientation

- *Encapsulation or data hiding* Encapsulation is the process of hiding all the details of an object that do not contribute to its essential characteristics. Essentially, it means that what is inside the class is hidden; only the external interfaces can be

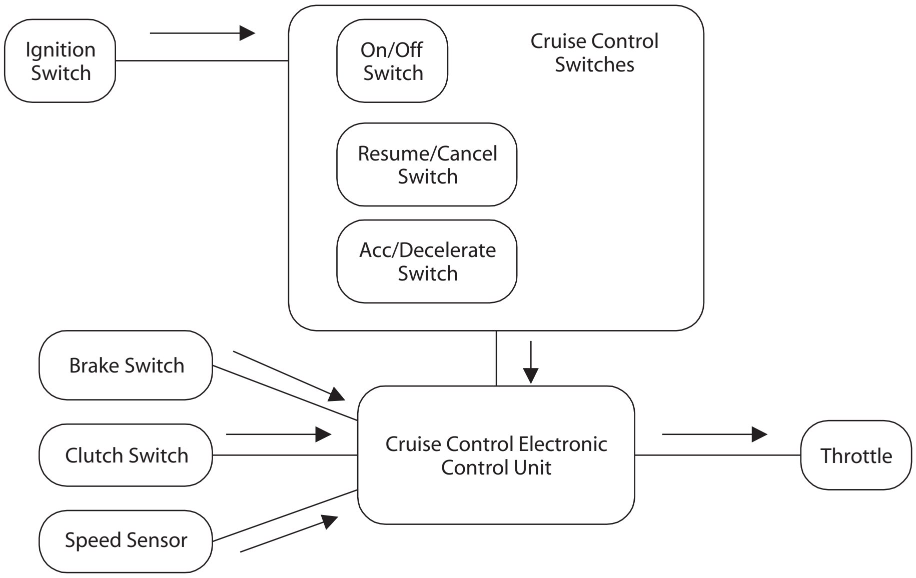

Figure 1.2 A cruise control system as a set of objects.

seen by other objects. The user of an object should never need to look inside the box!

- *Inheritance* Objects may have similar (but not identical) properties. One way of managing (classifying) such properties is to have a hierarchy of of classes. A class inherits from its immediate parent class and from classes above the parent (see the hierarchy in Figure 1.4). The inheritance mechanism permits common characteristics of an object to be defined once but used in many places. Any change is thus localized.

  If we define a concept *animal* and a concept *dog*, we do not have to specify all the things which a dog has in common with other animals. Instead, we inherit them by saying that dog is a **subclass** of animal. This feature is unique to object-oriented languages; it promotes (and achieves) huge amounts of reuse.
- *Abstraction* An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer. That is, it states how a particular object differs from all others.
- *Polymorphism* This is the ability to send the same message to different instances which appear to perform the same function. However, the way in which the message is handled depends on the class of which the instance is an example.

An interesting question to ask is: "How do languages such as Ada, C and Lisp relate to the four concepts above?". An obvious issue is related to inheritance. That is, if we define a concept animal and we then define a concept dog, we do not have to specify all the things which a dog has in common with other animals. Instead, we inherit them by saying that a dog is a **subclass** of animal. This feature is unique to object-oriented languages; it promotes (and achieves) huge amounts of reuse. The next four sections expand on each of these basic principles in more detail.

1.8 Encapsulation

Encapsulation or data hiding has been a major feature of a number of programming languages; Modula 2 and Ada both provide extensive encapsulation features. But what exactly is encapsulation? Essentially, it is the concept of hiding the data behind a software "wall". Those outside the wall cannot get direct access to the data. Instead, they must ask intermediaries (usually the owners of the data) to provide them with the data.

The advantage of encapsulation is that the user of the data does not need to know how, where, or in what form the owner of the data stores that data. This means that if the owner changes the way in which the data is stored, the user of the data need not be affected. The user still asks the owner for the data; it is the data owner that changes how the request is fulfilled.

Different programming languages implement encapsulation in different ways. For example, Ada enables encapsulation using packages which possess both data and procedures. It also specifies a set of interfaces which publish those operations that the package wishes to make available to users of the package. These interfaces may implement some operations or provide access to data held within the package.

Object-oriented languages provide encapsulation facilities which present the user of an object with a set of external interfaces. These interfaces specify the requests to which the object will respond (or, in the terminology of object orientation, the requests which the object will understand). These interfaces not only avoid the need for the caller to understand the internal details of the implementation, they actually prevent the user from obtaining that information. Users of an object cannot directly access the data held by an object, as it is not visible to them. In other words, a program that calls this facility can treat it as a black box; the program knows what the facility's external interfaces guarantee to do, and that is all it needs to know.

It is worth pointing out a difference between the object-oriented approach and the package approach used in Ada. In general, a package is a large unit of code providing a wide range of facilities with a large number of data structures (for example, the `TextIO` package). In an object-oriented language, the encapsulation is provided at the object level. While objects may well be as large and as complex as the typical Ada package, they are often much smaller. In languages such as Smalltalk and Java, where everything is an object, the smallest data and code units also naturally benefit from encapsulation. You can attempt to introduce the same level of encapsulation in Ada, but it is not natural to the language.

Figure 1.3 illustrates the way in which encapsulation works within an object-oriented language. It shows that anything outside the object can only gain access to the data the object holds through specific interfaces (the black squares). In turn, these interfaces trigger procedures which are internal to the object. These procedures may then access the data directly, use a second procedure as an intermediary or call an interface to another object.

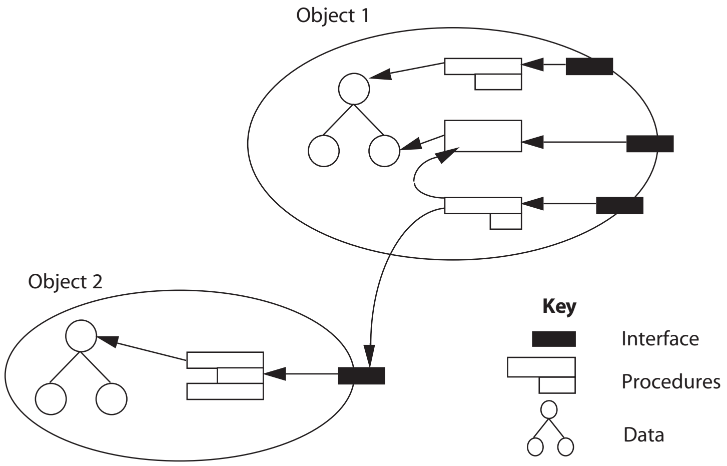

Figure 1.3 Object structure and interaction.

1.9 Inheritance

A class is an example of a particular type of thing (for example, *mammal* is a class of *animal*). In the object-oriented world, a class is a definition of the characteristics of that thing. Thus, in the case of mammals, we might define that they have fur, are warm blooded and produce live young. Animals such as dogs and cats are then instances of the class mammal. This is all quite obvious and should not present a conceptual problem for anyone. However, in most object-oriented languages (Self is an exception) the concept of the class is tightly linked to the concept of inheritance.

Inheritance allows us to state that one class is similar to another class but with a specified set of differences. Another way of putting it is that we can define all the things which are common about a class of things, and then define what is special about each subgrouping within a subclass.

For example, if we have a class defining all the common traits of mammals, we can define how particular categories of mammals differ. The duck-billed platypus is a quite extraordinary mammal which differs from other mammals in a number of important ways. However, we do not want to define all the things which it has in common with other mammals. Not only is this extra work, but we then have two places in which we have to maintain this information. We can therefore state that a duck-billed platypus is a class of mammal that does not produce live young. Classes allow us to do this.

An example which is rather closer to home for most computer scientists is illustrated in Figure 1.4. For this example, we assume that we have been given the job of designing and implementing an administration system for a small software house that produces payroll, pensions and other financial systems. This system needs to record both permanent and temporary employees of the company. For temporary employees, we need to record their department, the length of their contract, when they started and additional information which differs depending on whether they

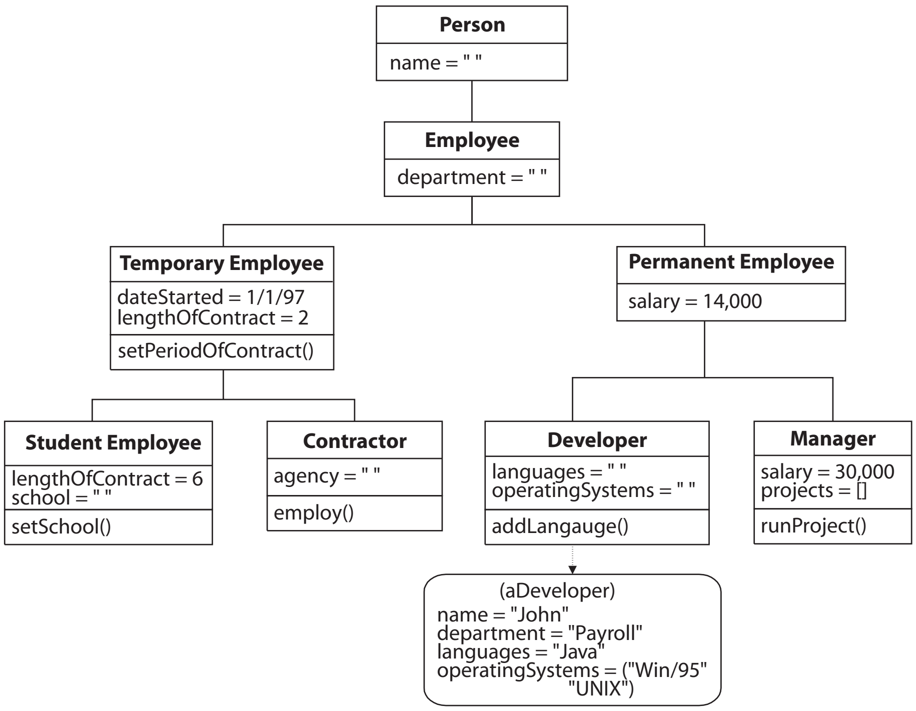

Figure 1.4 An example of inheritance.

are contractors or students on an industrial placement. For permanent employees, we need to record their department, their salary, the languages and operating systems with which they are familiar, and whether they are a manager. In the case of managers, we might also want to record the projects that they run.

Figure 1.4 illustrates a class hierarchy diagram for this application. It shows the classes we have defined and from where they inherit their information.

- *Inheritance versus instantiation* Stating that one class is a specialized version of a more generic class is different from saying that something is an example of a class of things. In the first case, we might say that developer is one category of employee and manager is another. Neither of these categories can be used to identify an individual. They are, in effect, templates for examples of those categories. In the second case, we say that "John" is an example of a developer (just as "Chris", "Myra" and "Denise" may also be examples of developers). "John" is therefore an instance of a particular class (or category) of things known as developers. It is important to get the concept of specializing a class with a subclass clear in your mind. It is all too easy to confuse an instance of a class with a subclass.
- *Inheritance of common information* We place common concepts together in a single class. For example, all people have a name and all employees have a nominated department (whether they are permanent or temporary). All temporary employees have a start date, whether they are contractors or students. In turn, all

classes below `Employee` inherit the concept of a department. This means that not only do all `Manager`s and `Developer`s have a department, but "John" has a department, which in this case is "Payroll".

- *Abstract classes* Figure 1.4 defines a number of classes of which we have no intention of making an example: `Employee`, `Permanent Employee` and `Temporary Employee`. These are termed abstract classes and are intended as placeholders for common features rather than as templates for a particular category of things. This is quite acceptable and is common practice in most object-oriented programs.
- *Inheritance of defaults* Just because we have stated that `Permanent Employee`s earn a default salary of £14,000 a year does not mean that all types of employee have that default. In the diagram, `Manager`s have a default of £30,000, illustrating that a class can overwrite the defaults defined in one of its parents.
- *Single and multiple inheritance* In Figure 1.4, we have only illustrated single inheritance. That is, a class inherits from only one other class. This is the case in many object-oriented programming languages, such as Java and Smalltalk. However, other languages such as C++ and Eiffel allow multiple inheritance. In multiple inheritance, you can bring together the characteristics of two classes to define a third class. For example, you may have two classes, `Toy` and `Car`, which can be used to create a third class `Toy-Car`. Multiple inheritance is a controversial subject which is still being debated. Those who think it is useful fail to see why other languages do not include it, and vice versa. Java does not include multiple inheritance.

1.10 Abstraction

Abstraction is much more than just the ability to define categories of things which can hold common features of other categories of things (for example, `Temporary Employee` is an abstract class of `Contractor` and `Student Employee`). It is a way of specifying what is particular about a group of classes of things. Often this means defining the interface for an object, the data that such an object holds and part of the functionality of that object.

For example, we might define a class `DataBuffer` which is the abstract class for things that hold data and return them on request. It may define how the data is held and that operators such as `put()` and `get()` are provided to add data to, and remove it from, the `DataBuffer`. The implementation of these operators may be left to those implementing a subclass of `DataBuffer`.

The class `DataBuffer` might be used to implement a stack or a queue. `Stack` could implement `get()` as *return the most recent data item added* while `Queue` could implement it as *return the oldest data item held*. In either case, a user of the class knows that `put()` and `get()` are available and work in the appropriate manner.

In some languages, abstraction is related to protection. For example, in C++ and and Java, you can state whether a subclass can overwrite data or procedures (and indeed whether it has to overwrite them). In Smalltalk, the developer cannot state that a procedure cannot be overwritten, but can state that a procedure (or method) is a subclass responsibility (that is, a subclass which implements the procedure in order to provide a functioning class).

Abstraction is also associated with the ability to define abstract data types (ADTs). In object-oriented terms these are classes (or groups of classes) which provide behaviour that acts as the infrastructure for a particular class of data type (for example, `DataBuffer` provides a stack or a queue). However, it is worth pointing out that ADTs are more commonly associated with procedural languages such as Ada. This is because the concepts in object orientation essentially supersede ADTs. That is, not only do they encompass all the elements of ADTs, they extend them by introducing inheritance.

1.11 Polymorphism

Polymorphism is a strange sounding word, derived from Greek,[1] for a relatively simple concept. It is essentially the ability to request that the same operation be performed by a wide range of different types of things. How the request is processed depends on the thing that receives the request. The programmer need not worry about how the request is handled, only that it is. Effectively, this means that you can ask many different things to perform the same action. For example, you might ask a range of objects to provide a printable string describing themselves. If you ask an instance of the `Manager` class, a compiler object or a database object to return such a string, you use the same interface call (`toString`, in Java).

The name "polymorphism" is unfortunate and often leads to confusion. It makes the whole process sound rather grander than it actually is. There are two types of polymorphism used in programming languages: overloading and overriding. The difference in name relates to the mechanism that resolves what code to execute.

1.11.1 Overloading Operators

Overloading occurs when procedures have the same name but apply to different data types. The compiler can determine which operator to use at compile time and can use the correct version.

Ada uses exactly this type of overloading. For example, you can define a new version of the + operator for a new data type. When a programmer uses +, the compiler uses the types associated with the operator to determine which version of + to use.

In C, although the same function, `printf`, is used to print any type of value, it is not a polymorphic function. The user must specify the correct format options to ensure that a value is printed correctly.

1 *Polymorphos* means "having many forms".

1.11.2 Overriding Operators

Overriding occurs when a procedure is defined in a class (for example, `Temporary Employee`) and also in one of its subclasses (for example, `Student Employee`). It means that instances of `Temporary Employee` and `Student Employee` can each respond to requests for this procedure (assuming it has not been made private to the class). For example, let us assume that we define the procedure `toString` in these classes. The pseudocode definition of this in `Temporary Employee` might be:

```
toString(){
  return "I am a temporary employee"
}
```

In `Student Employee`, it might be defined as:

```
toString(){
  return "I am a student employee"
}
```

The procedure in `Student Employee` replaces the version in `Temporary Employee` for all instances of `Student Employee`. If we ask an instance of `Student Employee` for the result of `toString`, we get the string "I am a student employee". If you are confused, think of it this way:

> If you ask an object to perform some operation, then, to determine which version of the procedure is run, look in the class used to create the instance. If the procedure is not defined there, look in the class's parent. Keep doing this until you find a procedure which implements the operation requested. This is the version which is used.

In languages such as Java the choice of which version of the procedure to execute is not determined at compile time, because the compiler would have to be able to determine the type of object and then find the appropriate version of the procedure. Instead, the procedure is chosen at run time. The technical term for this process of identifying the procedure at run time rather than compile time is called "late binding".

1.12 Summary

In this chapter you have been introduced to the background and history of object orientation. You have explored the main concepts which underpin object orientation and have encountered some of the (sometimes arcane) terminology used. There is a great deal of new information in this chapter which can, at times, appear to make obsolete all that you already know.

The object-oriented view of the world can be daunting for a programmer who is used to a more procedural view of the world. To adjust to this new view of the world is hard (and some never do). Others fail to see the difference between an object-oriented programming language and a language such as Ada (we refer here to the pre-Ada95 version). However, object orientation will become second nature to many once they have worked with object-oriented systems for a while. The key thing is to try things out as you go along and, if possible, have someone around who understands a bit about object orientation – they can often illuminate and simplify an otherwise gloomy network of tunnels.

1.13 Further Reading

There are a great many books available on object orientation. Some of the best known include Booch (1994), Budd (1991), Wirfs-Brock *et al.* (1990) and Cox and Novobilski (1991). An excellent book aimed at managers and senior programmers who want to learn how to apply object-oriented technology successfully to their projects is Booch (1996). Another good book in a similar style is Yourdon (1994).

Other books which may be of interest to those attempting to convince themselves or others that object technology can actually work are Harmon and Taylor (1993), Love (1993) and Meyer and Nerson (1993).

Other places to find useful references are the *Journal of Object-oriented Programming* (SIGS Publications, ISSN 0896-8438) and the OOPSLA conferences. The OOPSLA conferences are annual worldwide conferences on Object-oriented Programming: Systems, Languages and Applications. References for the proceedings of some recent conferences are listed at the back of this book. There are also references for the proceedings of the European Conference on Object-oriented Programming (ECOOP).

For further reading on the software crisis and approaches aimed at solving it see Brooks (1987) and Cox (1990). For a discussion of the nature of scientific discovery, refinement and revolution see Kuhn (1962).

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*, 2nd edn. Benjamin Cummings, Redwood City, CA.

Booch, G. (1996). *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley, Menlo Park, CA.

Brooks, F. (1987). No silver bullet: essence and accidents of software engineering. *IEEE Computer*, April.

Budd, T. (1991). *An Introduction to Object Oriented Programming*. Addison–Wesley. Reading, MA.

Cox, B. J. (1990). There *is* a silver bullet. *BYTE*, October, pp. 209–18.

Cox, B. J. and Novobilski, A. (1991). *Object-Oriented Programming: An Evolutionary Approach*, 2nd edn. Addison-Wesley, Reading, MA.

Harmon, P. and Taylor, D. (1993). *Objects in Action: Commercial Applications of Object-Oriented Technologies*. Addison-Wesley, Reading MA.

Kuhn, T. (1962). *The Structure of Scientific Revolutions*. The University of Chicago Press, Chicago, IL.

Love, T. (1993). *Object Lessons: Lessons Learned in Object-Oriented Development Projects*. SIGS Books, New York.

Meyer, B. and Nerson, J. (1993). *Object-Oriented Applications*. Prentice Hall, Englewood Cliffs NJ.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L. (1990). *Designing Object Oriented Software*. Prentice Hall, Englewood Cliffs, NJ.

Yourdon, E. (1994). *Object-Oriented Systems Design*. Prentice Hall, Englewood Cliffs, NJ.