Module 1: A Small Talk about Objects

This module contains a brief introduction to the ideas of object–oriented programming. Basic concepts such as *objects* and *messages* are presented. The notions of *class* and *instance*, and the advantages of a *class hierarchy* are explained. The way in which the correct methods are identified after a message send is described.

In this module we also briefly describe Smalltalk, its origins and the way in which it is used. This introduces the student to the later modules of the series, giving a glimpse of the way in which Smalltalk is different to other programming languages. In particular we outline the elements that combine to make Smalltalk — the language, the programming environment, the class library and the implementation.

Module 1: A Small Talk about Objects....1
- 1.1. Objects....2
- 1.2. Messages....2
- 1.3. Classes and Instances....4
- 1.4. Class Hierarchy....5
- 1.5. Methods....7
- 1.6. What is Smalltalk?....7
  - 1.6.1. A Language....8
  - 1.6.2. A Programming Environment....8
  - 1.6.3. A Class Library....9
  - 1.6.4. A Persistent Object Store....9
- 1.7. The Books....10
- 1.8. Typographical Conventions....10

© Trevor Hopkins and Bernard Horan, 1994

1.1. Objects

The fundamental concept that underpins Object–Oriented Programming is that of the *Object*. An object is a combination of two parts:

“Data” — the *state* of the object is maintained within that object.

“Operations” — all the mechanisms to *access* and *manipulate* that state.

The internal representation of an object consists of variables which either hold data directly or act as pointers to other objects (figure 1.1). The internal variables within the object are not directly accessible from outside that object — the barrier that this provides is called *information hiding*. In an object–oriented language, the “natural” view of objects is from the *outside*; the *inside* of the object is only of interest to itself. Therefore, no object can read or modify the state of any other object, i.e. unlike a data structure (for example, a Pascal record or a C struct), an external entity cannot force a change of state in an object. Access to any part of the state of an object is only possible if the object itself permits it.

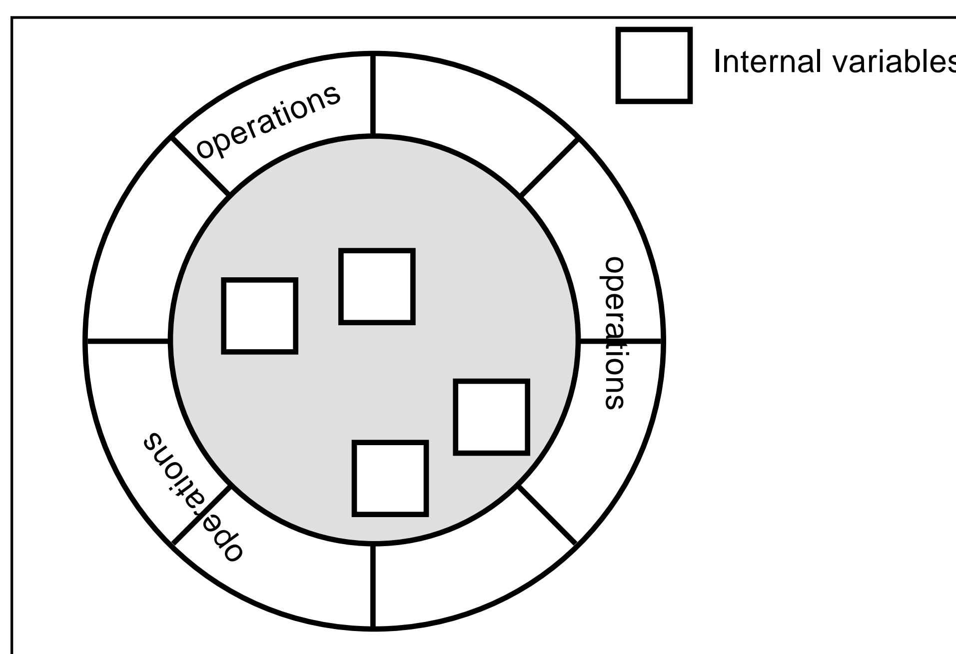

Figure 1.1: An Object

1.2. Messages

Object–Oriented programming languages use *message–sending* as their only means of performing operations. If the receiving object *understands* the message it has been sent, then one of its internal operations (or *methods*) will be performed. This, in turn, may cause some computation to take place (by acting

© Trevor Hopkins and Bernard Horan, 1994

on one or more of the object's internal variables). A result is always returned (i.e. an Object).

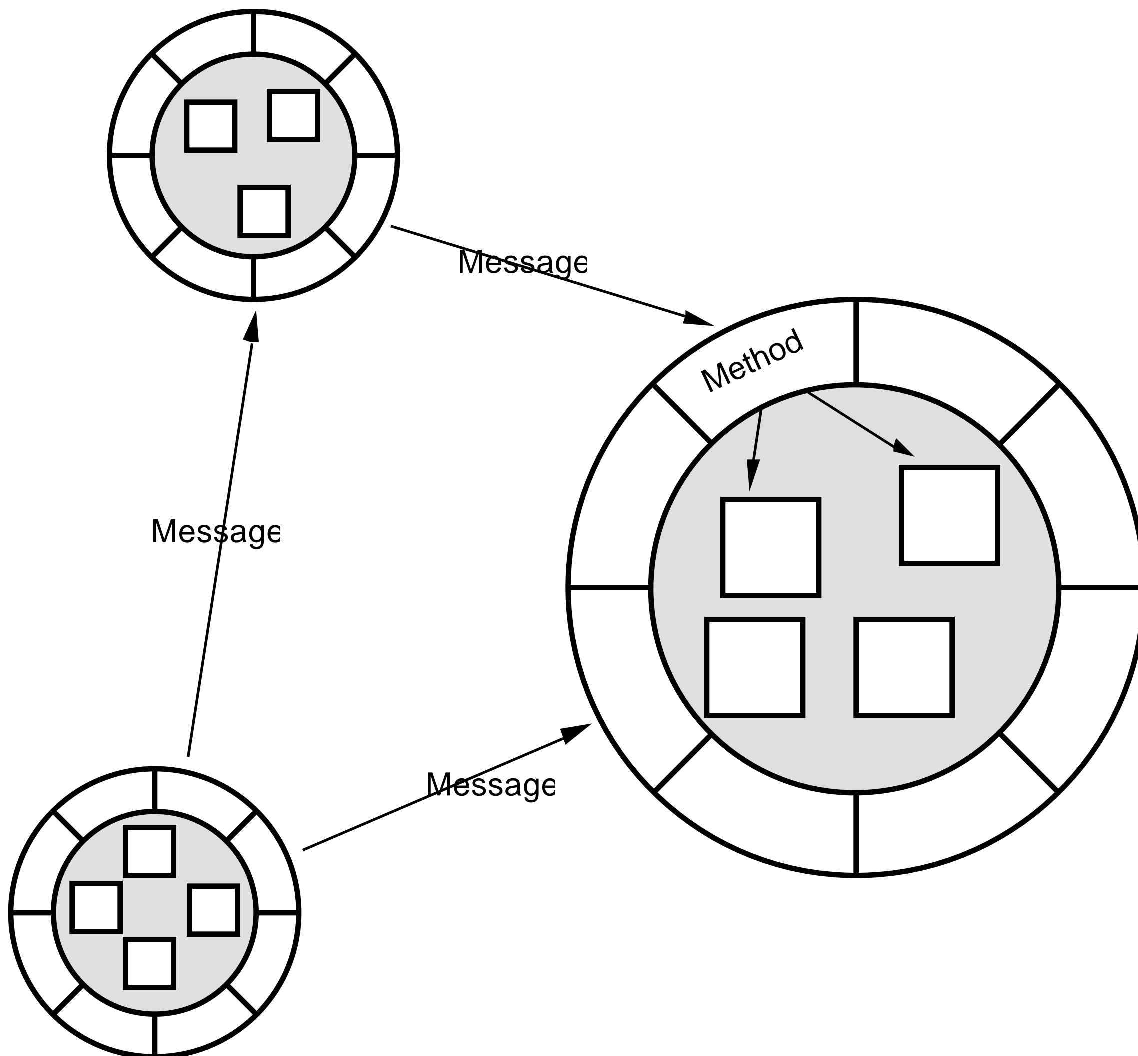

**Figure 1.2: Objects send *messages* to each other;
the receiver's corresponding *method* determines the operations to be performed.**

Since an object's internal state is private and cannot be directly accessed from the outside, the only way of accessing that state is by sending a message to the object. The set of messages to which an object responds is called its *message interface*. However, it is important to note that a message specifies only *which* operation is requested, not *how* that operation is to be fulfilled. The object receiving the message (the *receiver*) determines from its methods (described in some object–oriented language) how to carry out that operation. Similarly, a method that is being performed in response to the receipt of a message may only manipulate other objects (including the receiver's internal variables) by sending them messages. Program flow may therefore be illustrated in terms of communication between many objects (see figure 1.2).

© Trevor Hopkins and Bernard Horan, 1994

For example, a we could request an object to carry out some activity by sending it a message. One or more arguments may be sent to an object along with the name of the message (called the *selector*). The object receiving the message may be able to perform the action entirely on its own; alternatively, it may ask other objects for information, to carry out computation, etc., by sending messages.

The same message sent to different objects can produce different results, since it is the receiver of the message, not the sender, that decides what will happen as a result of a message–send. In this respect sending a message is fundamentally different to calling a procedure in a conventional programming language such as C.

Let us examine the result of sending the message + 5. Here the selector is +, and the argument is the integer 5.

> 6 + 5 returns 11
> (7@2) + 5 returns (12@7).

The result of sending the message +5 to the object integer 6 (equivalent to the expression 6 + 5) is 11. However, the result of sending the same message to the point (7@2) is (12@7). In the first case the receiver is an Integer, and an Integer is returned. In the second case, where the receiver is a Point[1], the operation and the result are different. In this case the addition of a scalar to a point returns another point. This feature — in which many objects are able to respond to the same message in different ways — is called *polymorphism*.

> Ex 1.1: Imagine that you are building a software model of the room in which you are sitting. Produce a list containing some of the objects in the model.

1.3. Classes and Instances

In theory, a programmer could implement an object in terms of the variables it contains and the set of messages to which it responds or understands (and the methods corresponding to those messages). However, it is more useful to share this information between similar objects. Not only does this approach save memory, it also provides a means of re–using code.

The information is shared by grouping together those objects that represent the same kind of entity into what is called a *class*. Every object in an object–oriented programming system is a member of a single class — it is called an *instance* of that class. Furthermore, every object contains a reference to the class of which it

[1]See module 4.

© Trevor Hopkins and Bernard Horan, 1994

is an instance. The class of an object acts as a template to determine the number of internal variables an instance will have, and holds a table of methods (a *method dictionary*) which corresponds to the messages to which all instances of the class will respond.

Therefore, we can see that if objects did not obtain their behaviour from classes, each object would have to “carry around” with it a copy of all the code it could evaluate. By using classes we avoid the potential efficiency problems in a fine–grain object–oriented system. Consequently, the behaviour of an object, expressed in terms of messages to which it responds, depends on its class. All objects of the same class have common methods, and therefore uniform behaviour, but they may have *different* state.

Ex 1.2: Identify the classes from the list of objects in Ex 1.1. For each class, describe the behaviour its instances might have.

1.4. Class Hierarchy

Classes are arranged in a class hierarchy. Every class has a parent class — or *superclass* — and may also have *subclasses*. A subclass *inherits* both the behaviour of its superclass (in terms of its method dictionary), and also the structure of its internal variables. At the top of the hierarchy is the only class without a superclass, called class Object in Smalltalk. Class Object defines the basic structure of all objects, and the methods corresponding to the messages to which every object will respond.

For example, referring to figure 1.3, we can see a class hierarchy where thomas is an instance of class Persian, which itself is a subclass of class Cat, which is a subclass of class Mammal, which may be a subclass of class Vertebrate, etc.

A superclass of which no instances should be created is known as an *abstract superclass*. Such classes are intended to support partial implementations of features that are completed (differently) in subclasses.

Inheritance supports *differential programming* (or programming by modification), i.e. a new class of object may be defined which is similar to an existing one (its superclass) except for a few minor differences. Subclasses therefore allow the programmer to *refine*, or *specialise*, the behaviour of the parent class. This can take a number of forms:

- additional or modified behaviour provided by extra methods;
- the re–implementation of the internal representation of the object’s state;

© Trevor Hopkins and Bernard Horan, 1994

- the addition of extra internal variables;
- any combination of the above.

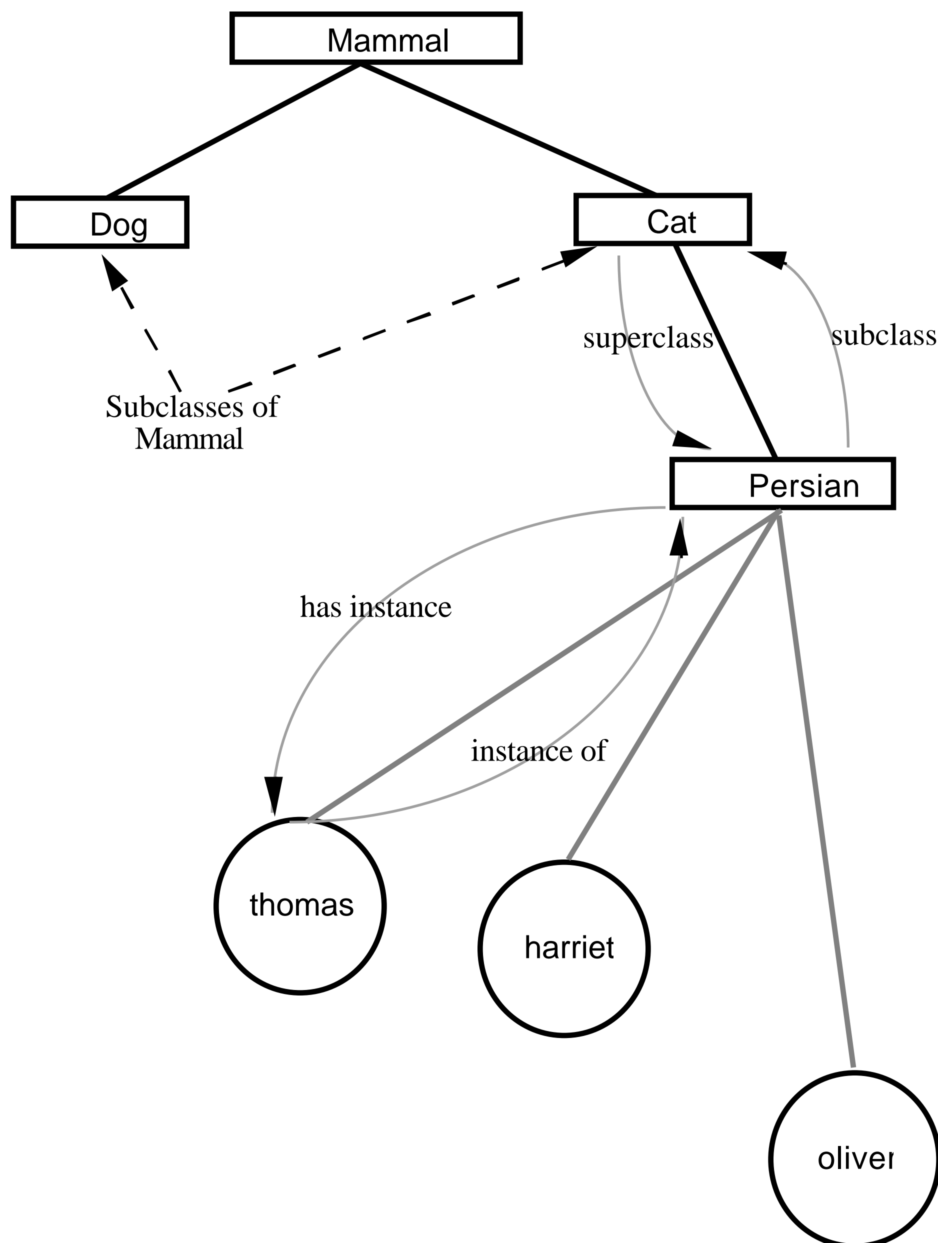

Figure 1.3: An Example of a Class Hierarchy and its instances.

We may therefore exploit inheritance as another technique for re–using code. For example, if the message odd is sent to the Integer 12, the result false will be returned even though the class Integer does not have a method specifying how to make this test. This is because the method corresponding to the message odd is defined in class Number, which is a superclass of class Integer. This means that all subclasses of class Number, including integers, are able to respond to the message odd, since they inherit this method.

© Trevor Hopkins and Bernard Horan, 1994

Within Smalltalk, the class structure is implemented as a *single inheritance* hierarchy. This means that a subclass can only inherit from one parent superclass. Other Object–Oriented programming languages support *multiple inheritance*, in which case a subclass may inherit from multiple parent superclasses.

Ex 1.3: Arrange the classes (identified in Ex 1.2) in a hierarchy (if appropriate). Are any of your classes abstract?

1.5. Methods

A message is sent to an object which is an instance of some class. A search is made in the class's method dictionary for the method corresponding to the message selector. If the method is found there, then it is *bound* to the message and evaluated, and the appropriate response returned. If the appropriate method is not found, then a search is made in the instance's class's immediate superclass. This process repeats up the class hierarchy until either the method is located or there are no further superclasses. In the latter case, the system notifies the programmer that a run–time error has occurred.

The object–oriented message passing mechanism can be compared to a function call in a conventional programming language such as C. It is similar in that the point of control is moved to the receiver; the object sending the message is suspended until a response is received. It is different in that the receiver of a message is not determined when the code is created ("compile time"), but is determined when the message is actually sent ("run time"). This *dynamic* (or late) binding mechanism is the feature which gives Smalltalk its polymorphic capabilities.

Ex 1.4: Identify some of methods for each class (from Ex 1.3).

1.6. What is Smalltalk?

Smalltalk can be seen as just another programming language, with its own rules of syntax for describing classes, objects, methods and messages; and its own rules of grammar for parsing expressions. However, most implementations of Smalltalk provide more than just a language — they provide a programming environment, a library of classes and a persistent object store. Each of these combines to produce a unified framework for the development of object–oriented applications. Each is described in detail below.

© Trevor Hopkins and Bernard Horan, 1994

1.6.1. A Language

Compared to conventional programming languages such as C or Pascal, Smalltalk has an unusual syntax. In addition, Smalltalk has no notion of “type”. Objects are employed to represent everything in Smalltalk, including all the conventional data types that may be found in any programming language: integers, booleans, floating point numbers, characters, strings, arrays, etc. In addition, objects are used to represent display items such as menus, windows, etc., and even the compiler itself. Smalltalk is therefore described as a *uniformly* object–oriented language.

However, the rules of Smalltalk syntax (see module 2) are simple and consistent. When you have mastered these rules and have gained a certain familiarity, it is a straightforward language both to read and write. The problems are no greater than mastering the syntax of Lisp, APL and Forth and, like the adherents of those languages, most Smalltalk programmers argue that the syntax is one of the strengths of the language.

1.6.2. A Programming Environment

Smalltalk was one of the first systems to pioneer the so–called “WIMP” interface (Windows, Icons, Menus & Pointer). It is not surprising, then, to discover that current environments provide numerous tools to enable programmers to browse existing classes and copy and modify pre–written code (module 3). Additional editing tools enable programmers to amend and correct newly created code effortlessly. Other tools allow the programmer access to the underlying filing system (module 3), from which existing source code my be “filed–in”. Additionally, Smalltalk provides change management tools. These are in the form of *Projects* that may be used to contain code specific to particular software Projects and also Browsers to view recent changes. Since Smalltalk is extensible, programmers can tailor these existing tools or create new ones for their own use.

Smalltalk also pioneered the use of an incremental compiler. This means that programmers can avoid lengthy sessions of compiling and linking, and get “instant gratification” from the speed of compilation, thus enabling them to adopt a more exploratory approach to software development. This approach enables the programmer to develop software in a piecemeal fashion, exploring ideas as the application takes shape.

The combination of an incremental compiler and a symbolic debugger (giving interactive access to source code) allows the programmer to inspect and modify currently active objects, carry out in–line testing and modify and re–compile

© Trevor Hopkins and Bernard Horan, 1994

existing code, and then restart the evaluation of the expressions. By inserting break points at suitable points, the programmer is able to step through the evaluation of the code (see module 11).

Thus, the Smalltalk programming environment promotes a programming process that employs an iterative development style for creating an application. The programmer is able to develop a partial solution, test it, add new functionality, test it, and so on, until the complete solution is reached.

1.6.3. A Class Library

Despite its name, Smalltalk is not necessarily small — for example, Smalltalk–80 contains hundreds of classes and thousands of methods, all available on–line in source code form (Smalltalk is written almost entirely in Smalltalk — a small kernel is written in machine code). The classes can be further refined using the inheritance mechanism, or they can be used as internal variables by other objects. Consequently, the effort required to construct new applications is minimised.

The library of classes includes:

- Various subclasses of Number. This includes Integer, Float and Fraction.
- Various data structures. This includes Set, Bag, Array, OrderedCollection, SortedCollection and Dictionary.
- Classes to represent text, font, colour, etc.
- Geometric representations, e.g. Point, Rectangle, Circle, Polygon, etc.
- Classes to assist in the control of concurrency. For example, Process, Semaphore.

1.6.4. A Persistent Object Store

A Smalltalk system consists of two parts:

- The *virtual image* (or “Image”), containing all the objects in the system.
- The *virtual machine* (or “VM”), consisting of hardware and software (microcode) to give dynamics to objects in the image[1]. (VisualWorks refers to the VM as the “Object Engine”.)

This implementation technique was used for several reasons:

[1] Each platform requires its own VM.

© Trevor Hopkins and Bernard Horan, 1994

- To ensure portability of the virtual image. Any Smalltalk image should be executable on any virtual machine. The image is (mostly) isolated from the VM implementation (hardware and software).

- The Smalltalk system is very large, but most of the functionality is in the image. This eases the problems of implementation, as only a relatively simple VM has to be constructed.

Because the complete working environment is saved (the *image*), Smalltalk also acts as a persistent object store. This allows the programmer to create new classes, and even new instances, and keep them from one working session to another.

Ex 1.5: How would you best represent the internal implementation of each of your classes? Describe each class and the relationship between them.

1.7. The Books

Throughout the modules you will see occasional reference to the "Orange Book" or the "Blue Book", etc. Here we refer to one of the four books to be written by authors who were (and some who still are) involved with the development of Smalltalk–80. The "colour" of the book indicates the colour of the background of the illustration on the front cover (as well as for the Addison–Wesley logo on the spine). The full references of the books are as follows:

Blue Book — Goldberg, Adele, and David Robson, Smalltalk–80: The language and Its Implementation, Addison–Wesley, 1983.

Orange Book — Goldberg, Adele, Smalltalk–80: The Interactive Programming Environment, Addison–Wesley, 1984.

Green Book — Krasner, Glenn, ed., Smalltalk–80: Bits of History, Words of Advice, Addison–Wesley, 1983.

Purple Book — Goldberg, Adele, and David Robson, Smalltalk–80: The language, Addison–Wesley, 1989.[1]

These notes are adapted from Smalltalk: An Introduction to Application Development using VisualWorks, by Trevor Hopkins and Bernard Horan, due to be published by Prentice Hall in 1995.

1.8. Typographical Conventions

In the module notes the standard typeface is a serif font. Other fonts are used to distinguish special terms, as follows:

[1] The Purple book is an update/revision of the Blue book.

© Trevor Hopkins and Bernard Horan, 1994

- Menu items are in bold sans–serif, e.g. **Workspace**.
- Smalltalk expressions or variables, as they appear in the system, appear in sans–serif, such as Transcript show: 'Hello' .
- Place holders and variables used in examples are in italics sans–serif typeface, for example *aCollection*.
- Keys that appear on the keyboard are indicated by angle brackets. For example the “carriage return” key: <CR>, or a function key: <F1>.

From now on, when we mention Smalltalk, we shall be referring to the Smalltalk–80 language, unless stated otherwise.

© Trevor Hopkins and Bernard Horan, 1994