Programming Fundamentals 2

This chapter begins the examination of what programming is and how a programming language like Java works. In particular, it introduces the basic ideas and concepts of imperative programming, which is about writing programs using instruction sequences and updatable state. While Java is known as an object-oriented programming language it is also an imperative language and relies on the key principles introduced in this chapter. The following chapters will expand on these principles and show how they lead to the idea of object-oriented programming.

2.1 Introduction

To construct any computer-based system, we must use some process to translate the idea for the use of the computer into lines of source code which can be compiled and executed. This process typically includes the tasks of: *requirements gathering*, what we would like the system to do; *analysis*, finding out how the system should behave; *design*, deciding the structure of the system to be constructed; *implementation*, writing the source code; and *testing*, *verification* and *validation*, making sure the system does what we claim! The tasks of implementation and testing also include the task of *debugging*, which is the finding and removing of errors (usually called bugs, hence the term debugging) in our program.

Measured in terms of the quantity of source code required for the implementation of the system, computer-based systems come in varying sizes from the very small to the very large. The term *programming* is usually applied to the task of constructing small- to medium-sized systems. The term *information systems engineering* (sometimes called *software engineering*) is usually applied to the activity of constructing medium- to large-sized systems. The basic process of development is essentially the same in all cases but because of the scale of the problem being addressed, information systems engineering not only subsumes programming but requires tools and techniques over and above those required for programming. It is not the purpose of this book to introduce information

systems engineering but to cover programming, i.e. the construction of small- to medium-sized systems.

The principal tool for implementation is the programming language, with one example, Java, being the subject of this book. The design of a language like Java is based on principles that are the result of both many years of research and of the practical use of earlier generations of programming languages. The research addresses not only the best ways of making the computer behave as we want it to, but also how best to avoid the errors that human beings, being imperfect, introduce into the systems they are developing. The practical experience of using programming languages highlight those features that provide useful solutions to real problems and work effectively in the social environment of the development team.

Thus, the programming language and the development tools used for constructing programs try to prevent the programmer making errors in the first place and, if errors are introduced, help finding and eradicating them quickly and efficiently. The features of Java and the tools for developing Java programs support these principles as we will show in this and the following chapters.

Underlying all of this, we need an understanding of how a program is executed† — the *execution model* — and how the source code of the program is structured so as to exploit this model effectively, whilst accurately reflecting the design of the program. This knowledge is essential so that we can reason about how our programs will execute once compiled. This mental execution of the program is an integral part of the way in which people construct programs and hence is an important skill any programmer requires.

So what model does Java have? Using the vocabulary of programming language design, Java can be categorized as an *imperative object-oriented programming language*. The underlying model of computation in Java is a set of interacting objects. Therefore, we can use *object-oriented analysis and design methods* to determine how to structure the implementation of a Java program, allowing the source code of that Java program to directly reflect the intended design. Then, when the program is executed, the execution model is, in turn, object-oriented and built on top of fundamental ideas such as instruction sequences and state.

But what does all this jargon really mean? This and the other chapters of this Part of the book will explore these ideas and show how they are supported by Java. Let us start with some of the most basic ideas.

2.2 Imperative Programming

A program written using an imperative programming language is executed by following an *ordered sequence* of instructions: in programming languages such as Java these instructions are usually termed *statements*. Based on this idea of an ordered sequence of

†Note that the Concise Oxford Dictionary defines execute (*definition truncated*): 1. carry out a sentence of death 2. carry into effect, perform 3. make (a legal document) valid by signing. In computing the term is used with meaning 2!

statements, a program may be designed by decomposing a problem into a sequence of statements and then having them executed in the order they are written down.

As an example of imperative statements let us consider moving around a two-dimensional space. In particular, let us consider moving around a on chess board.

From a given square, we can go `Forward`, `Left`, `Right` or `Backward` (we ignore diagonal movement for the purpose of this example). In this context, the following is a reasonable program†:

```
Forward ; Left ; Forward ; Right ; Forward ;
```

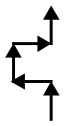
In order to be able to decide the outcome of the execution of this program, we need to know a number of things: what do the instructions mean (what are the *semantics*) and where are we currently.

Determining the semantics of the instructions, rests on two factors:

1. How many squares constitute a single move.
2. Are the directions fixed relative to the board or relative to the direction of travel.

If we supply definite answers to these questions we have specified the execution model of this machine. Since it is the most obvious solution, and indeed the best, let us assume we move one square per move. The only variable left is how directions are specified.

If the directions are fixed relative to the board, then, assuming we start from a square somewhere in the middle of the board, the above program gives the following behaviour:



In fact, with fixed directions, any ordering of a given set of statements will take us through different squares but always leave us at the same place; a different sequence of instructions results in different behaviour but the same result. For example, the sequence:

```
Forward ; Forward ; Right ; Forward ; Left ;
```

looks like:

We have taken a different route to the same place, the square 3 forward — in effect the left and right moves cancel each other out.

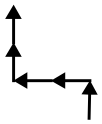
---

†The semi-colon is being used to mark the end of the statement, separating it from the next statement.

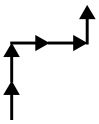
Exercise 2.1

How many different orderings of the five statements in the two programs considered so far are there?

If the directions are determined with respect to the direction of travel, we have a very different situation. The original program looks like:



whereas the second program looks like:



Not only is the sequence of squares visited different but the destination square is different. With this execution model, the order of statements matters even more than it did with the previous model.

A completely different execution model using the same language is where `Forward` is the only movement statement, with `Left` and `Right` being orientation changing statements only. With these semantics, the first program represents the behaviour:

whereas the second program represents the behaviour:

Again the different ordering of the statement leads to a different behaviour (squares visited) and a different final square.

In summary, for an imperative language, not only does changing the order of statements change the behaviour of a program, it can, and usually does, mean that the final result is different.

These ideas of statements and statement sequences constitute a useful starting point but writing programs consisting of one huge list of statements is not only uninteresting, it

also severely limits the kinds of program that can be created. In order to do better, two more concepts are required:

1. *control structures* in order to control the order in which statements are executed; and
2. the idea of *state* to hold values that can be manipulated by the statements.

2.2.1 Controlling Execution

To provide control over the order of execution of statements, imperative languages provide *selection* and *iteration* statements.

Selection allows one of two or more sequences of statements to be selected and executed. This provides a way of testing a condition and then deciding which set of statements to execute based on the value of the condition.

The *if statement* is used to select one of two choices (*binary selection*). For example:

```
if (space-to-the-right)
    Right ; Forward ;
else
    Left ; Forward ;
```

`space-to-the-right` is a *condition* which tests whether there is a valid location for the `Right ; Forward` statement sequence† and executes the right turn and move forwards if there is, otherwise (else) a left turn and then a forward movement are executed. The condition is a *boolean expression* — an expression that has a value of either true or false.

Iteration allows sequences of statements to be repeated some number of times. The repetition may either be bounded (repeated a fixed number of times), or unbounded (repeated some arbitrary number of times). For example:

```
while (space-to-the-front)
    Forward ;
```

This *while loop* will repeatedly execute the `Forward` statement while there is `space-to-the-front`. As with `space-to-the-right`, `space-to-the-front` is a boolean condition, this time to determine how long to keep repeating the loop.

As we will see in later chapters, the Java programming language provides a number of different kinds of *control flow statements*, including the ones above, allowing iteration and selection to be expressed. The written appearance of such statements is defined by the Java language *syntax* which provides a *keyword* for each kind of statement. For example, both '**if**' and '**while**' are keywords used when writing down if or while statements. A list of Java keywords can be found in the Appendix.

2.2.2 State and Variables

In general, statements on their own are still not very useful unless they are able to manipulate *values* and especially *stored values*. Think of trying to bake a cake using a recipe that did not state how much of each ingredient to put into the mix. To follow a baking recipe, we need the values of the amount of each ingredient, the value of the time to cook for, the value of the temperature to cook at and the value of the amount of time

†Actually, it is an assumption that this is what this condition tests for based on the name and the context. We are assuming that programmers use sensible and descriptive names in their programs.

the baking has already gone on for. This last feature is an instance of *state*: the cooking device 'remembers' the temperature set and using sensors and control circuitry in the oven keeps the temperature at the set level.

Imperative programming languages have this idea of state, where the state provides a set of values that can be manipulated by statements.

The execution of an imperative program can be described in terms of a *state machine model*, where each statement causes the program to move from one state to the next. The initial state holds the set of starting values (the input) while the final state holds the set of result values (the output). The statements in the program are responsible for transforming the initial state to the final state.

The state of a Java program is represented by:

- the Java Virtual Machine that defines how Java programs are executed; and
- a collection of *variables* to hold the values that are manipulated by the program.

A variable is a *container* which is able to hold a representation that denotes a *value*. For example:

the square box represents a variable (the container), holding a binary number (the representation) which denotes the value one.

Values, for example the integer value one, are abstract things which cannot be manipulated directly — how can you pick up "the value one"? In order to be manipulated, we must use a representation of the value. The same abstract value can be represented in many different ways, e.g. one can be represented by '1', 'i', 'I', "one", "ONE", etc. on paper. Different representations have different properties. The obvious example here is the roman and arabic representation of numbers. The number nineteen hundred and ninety seven can be represented in words (as we just have), in its roman representation, MCMCXXXXVII, or in its arabic representation, 1997. Doing arithmetic with the arabic representation is relatively easy, using the roman representation makes it very hard. The arabic representation systems is therefore the one we normally make use of.

Computer hardware represents integer values in a binary representation[†]. The Java Virtual Machine makes use of these directly when a Java program executes. Fortunately, programmers do not have to worry about this representation since the Java programming language shields us from it. In a Java program, we can make use of representations employing arabic style numbers.

As we shall see, it is crucially important to distinguish between the variable, the representation it holds and the value that is represented. A variable is just a container. The representation, on the other hand, is some representation of an *abstract* value. This is not an artificial distinction. Consider integers — there are an infinite number of them. If we

†Most machines use a representation called "2's complement binary". We are not going to present the details of the representation in this book, the interested reader is referred to any book on computer architecture.

wished to store *any* integer value we would need to have a variable that could hold a representation of infinite size. This is not very practical (to say the least!), so the representation is limited to a range of integers within a fixed maximum and minimum value; a finite range. As a programmer you need to know what that range is and be aware that integers outside that range cannot be represented even though they are otherwise normal values.

Variable are given names, allowing statements in a program to refer to them. For example, the variable holding the result of a calculation can be called ‘result’. Naming a variable is equivalent to sticking a name onto the variable container.

2.2.3 Assignment

Variables are *updatable*: a variable is a container whose contents can be overwritten, by replacing the current representation with a new one. The execution of an imperative program relies on variables being updated in order to produce the end result. A typical Java program will have lots of variables which are constantly updated as the program executes.

The act of changing a variable is known as *assignment* — a variable may be assigned a new value, meaning that the contents of the container is changed. Assignment does not change the name of a variable, only the representation held in the variable container.

An example of assignment is the Java statement:

```
x = 1 ;
```

The `=` symbol is the assignment operator in Java; the semi-colon marks the end of the statement. The result of the execution of this statement is that the variable named `x` is assigned a representation of the value one. When programmers talk about assignment they often use expressions like “assign the variable `x` the value one”. The representation and value are both referred to as “the value”. However, as noted above, it is important to distinguish the two; something which can really only be done by appreciating the context in which the term is being used.

Interestingly, assignment allows you to write the statement:

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

This is a valid expression in an imperative language like Java — it means fetch the value in `x`, add one to it, and save the result back into `x`. If we were to think of this as a mathematical expression, it would only be valid for the value infinity (∞) or in a very strange algebra! It is fortunate that this is a Java statement.

2.2.4 Variable Types and Type Checking

An abstract value has a *type*. A type defines a set of possible values and the set of operations that can be applied to the values that are members of that type. For example, in mathematics the value one is of type integer, has a representation such as ‘1’ and can be manipulated using the mathematical operations normally associated with integer numbers: addition, subtraction, multiplication, etc.

As abstract values can be given types, so can representations of them and also the variables that hold the representations. This allows the programmer to specify that a

variable can hold values of a particular type and no others. In Java, this is done by specifying a type when a variable is first named — or *declared*. For example:

```
int j ;
```

declares a variable called `j` of type `int`[†].

Java is a *strongly typed language*, meaning that each variable *must* have a type associated with it determining what kind of representation (and hence value) the variable is able to hold.

Types are an important way of accurately specifying how a program is to work and of verifying that a program is properly constructed. The Java compiler uses types to do *type checking* which essentially means that it checks that any value that is assigned to a variable is of the right type to put in that variable. For example, if the Java programmer writes:

```
int j ;
j = 3.141 ;
```

the compiler will notice that an attempt is being made to assign a real number (3.141) to an `int` variable. As the types don't match, the assignment is not allowed and an error is reported. The error prevents an executable program from being created, forcing the programmer to correct the problem.

Strong typing should not be seen as something the programmer must fight with, it is there as a support tool for avoiding avoidable errors. The principle being used here is that if the programmer is required to specify exactly the types of all the representations and variables that they are using at all times throughout the program, then any violation of strong type checking indicates either a trivial error on the part of the programmer (which can be corrected trivially) or that there is a more serious error of analysis or design. In all cases, the programmer, in correcting the problem, must reason carefully about their program. Strong typing is a support tool to help the programmer in this reasoning activity.

Exercise 2.2

1. Write down a sequence of statements that describes making a telephone call. Can it be done without using control statements?
2. Think about how a floating point number could be stored in a variable. How does the fixed size of a variable affect the storage of floating point numbers?
3. Compare the idea of types with units used for measuring size or weight. Do such units have types?

2.3 Moving On — Giving a Program Structure

The preceding sections have introduced the core ideas of imperative programming, on which Java is built. We have introduced the ideas of sequence, iteration, selection and updatable variables. Using these features a programmer can express *algorithms* — step

†Java has its roots in C++ and C; these languages used the symbol `int` as a label for the integer type.

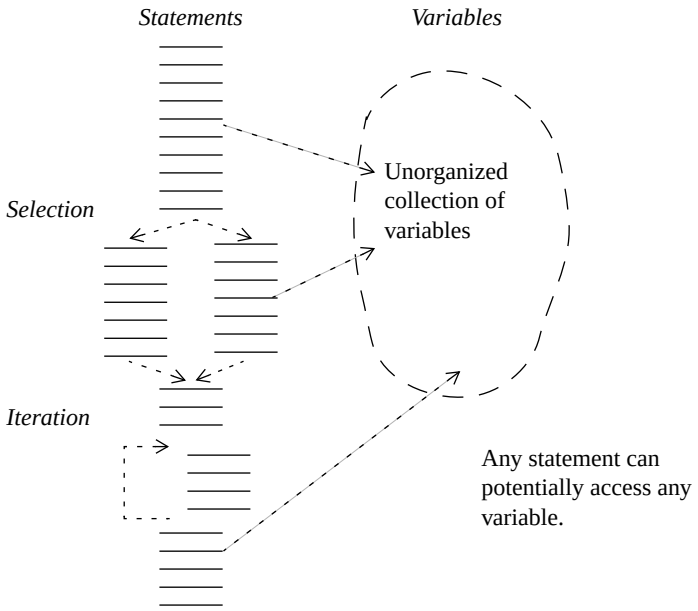


Figure 2.1 A program with no real structure.

by step descriptions of how go about things; for example cooking recipes are algorithms for turning raw ingredients into more interesting forms of food. With these basic building blocks it is possible, but tedious, to build useful programs. We therefore need further infrastructure in order to allow arbitrarily complex programs to be written in such a way that we can understand and reason about them.

Let us consider the problems of using only sequence, iteration and selection in a little more detail in order to give ourselves an insight into the new features that we need. With only these features, then:

- There is little, if any, structure to a program. Even with iteration and selection a program still essentially consists of one long list of instructions. Execution starts at the top and continues to the end, with statements placed according to their execution order, not according to a well planned and implemented structure for the program. Further, all the variables used by a program exist in an unstructured collection, into which statements dive for access to the representation. These ideas are shown diagrammatically in Figure 2.1.
- Everything has to be built from the basic elements. Statements are able to manipulate variables, control iteration and make selections but there is no way of creating larger building blocks. We need a way to group related blocks of statements together so they only need to be written once and used many times.
- It is very difficult for people to read and maintain such programs. Just like reading a book with no structure, reading a long list of instructions is difficult and frustrating. Mistakes can be easily made.

We need to start imposing some further order and structure on our programs. To be able to do this, we must examine some additional basic principles. First we need to consider *abstraction* and *encapsulation* of instructions and data, and then ideas about *scope* and *lifetime*.

2.3.1 Abstraction & Encapsulation

Abstraction is the process of capturing the essential or distinguishing features of something while suppressing or ignoring the detail. In doing so, it provides a crucial mechanism to enable people to understand and reason about complex systems, for example computer programs. Without abstraction, the level of detail required to understand a system becomes so overwhelming that the programmer is unable to build a mental model of how it is structured and how it works. In fact, abstraction is a mechanism that people use all the time to understand and interact with the real world, typically without really noticing that it occurs. For example, a conversation about how to drive from A to B can take place without ever having to discuss all the details of how a car works or how every road on the route is constructed: we abstract away the details of road surface construction, the internal combustion engine and all the rest of the technology discussing only the abstractions of turning right, travelling one block, etc.

Thus, as dealing with long complicated sequences of statements is hard for people, even programmers, exploiting abstraction is essential if any non-trivial program is to be constructed. We want to be able to hide away the full detail of parts of a program behind a simpler, more abstract, interface: we want to be able to collect parts of the statement sequence together into abstractions of the statement sequence. This will allow the programmer to conceptualise the program in terms of a smaller number of abstractions, rather than a much larger number of individual statements.

Abstraction in programming is intimately associated with the concept of *encapsulation*. Encapsulation is a mechanism for ensuring that things have a definite inside and outside. The inside is protected from anything on the outside, offering a guarantee that some outside agent cannot arbitrarily change anything on the inside.

Once protected by encapsulation an abstraction can only be accessed via a well defined interface visible from the outside. The inner workings are hidden and do not have to be known about in order to make use of the object. Hence, the outside view and the interface present the abstraction, hiding away the internal details. Encapsulation is often referred to as being an *information hiding* mechanism for fairly obvious reasons.

Defining and using the various kinds of abstractions that provide encapsulation is a key part of programming using any modern programming language and is most especially important in Java programming. Java provides abstractions over state (variables and types) as well as abstractions over statements. The abstractions over state are the objects that give rise to the label object-oriented. First though we must look at abstractions over statements.

2.3.2 Procedural Abstraction

In order to reason sensibly about even relatively small programs, we need to break up the long lists of statements into smaller chunks, each chunk being given a name, and thence

referred to and used without having to worry what the list of statements in the chunk actually are. Abstraction allows us to focus on the name of the chunk and what it does, without needing the detail of how it does something.

In languages preceding Java, this idea is known as *procedural abstraction*. A procedure consists of a sequence of statements (known as a *compound statement*) and has a name. For example†:

```
procedure move
{
    forward ;
    left ;
    forward ;
    right ;
}
```

The procedure is called `move`, while the procedure body consists of a compound statement (bounded by braces) containing statements. To invoke the procedure (or as it is usually termed "call the procedure") and execute its statements we just use its name, followed by '`()`' — the parentheses act as an operator to call the procedure:

```
move() ;
forward ;
move() ;
right ;
move() ;
```

This statement sequence will call the `move` procedure three times, each time executing the statements in the procedure body. The statements in the procedure only have to be written once but can be used anywhere in a program by calling the procedure.

Based on this we could sub-divide a program in to a series of procedures and call each procedure in the order we want things done. Each procedure should be *cohesive* — that is it should focus on doing one well-defined chunk of behaviour needed by the program.

We should also note that a procedure can call other procedures, allowing procedures to be defined in terms of other procedures. Also procedures are able to call themselves. This activity, a procedure calling itself, is such an important special case that it has a name, *recursion*. Another special case is where a procedure calls another procedure which calls the first procedure. This sort of indirect recursion is called *mutual recursion*.

As well as acting as an abstraction over instructions, procedures may additionally return a value. The sequence of instructions in the procedure may be designed to compute a value which the calling procedure can make use of. To avoid any problems with employing variables to communicate the computed value, there is a special mechanism for procedures to return values. A value-returning procedure can be called (again by using its name) and the value returned used in another expression. For example:

```
i = f() ;
```

Here the procedure `f` is called and the result, which must be an `int` value, is assigned to the `int` variable `i`. This mechanism can be used to construct all sorts of useful value-returning procedures, for example mathematical function such as square root or factorial. Indeed, value-returning procedures are often called functions.

---

†This is not Java syntax and hence not a valid Java program. See page 18 for the proper syntax of procedure definition.

2.3.3 Procedure Arguments

Like a mathematical function, a procedure (whether value-returning or not) needs the potential to be able to use input values in order to start the computation with different initial conditions. Consider square root or factorial for example; these functions would be impossible to implement were it not possible to give it an initial value to compute the square root or factorial of. Programming languages provide simple mechanisms to enable procedures to be given these input values or *arguments* as they are often called.

Thus, we can construct procedures to be called with different arguments in order to give different results:

```
double d ;
d = squareRoot(20.0) ;
int i ;
i = factorial(10) ;
```

However, we must be careful not to assume that procedures are actually mathematical functions. Mathematical functions always return the same value given the same argument, this is termed *referential transparency*. Procedures that implement mathematical functions must clearly exhibit referential transparency in order to be proper implementations. However, not all procedures implement mathematical functions; these other procedures may, or may not, be referentially transparent — it is possible to call some procedures with the same arguments several times but get different results each time. It is crucial to remember this very important difference between mathematical functions and procedures!

We do not always want our procedures to have only a single argument and indeed a procedure can take any number of arguments, provided as a list with each parameter separated by a comma. For example:

```
int i ;
i = f(1,3.1,2) ;
```

It is, however, not a good idea to have more than about six to eight arguments unless absolutely necessary since this usually leads to bulky and less comprehensible source code.

Being able to call procedures is fine but we must be able to construct them! In Java, the factorial function would look something like:

```
int factorial (int x)
{
    Statements
}
```

The `int` preceding the name of the procedure is the type of the value returned by the procedure (the *return type*). The procedure name, `factorial`, is followed by a single *parameter variable*, named `x`, preceded by its type, `int`. `x` behaves like a *local variable*†, except that its initial value is set to the argument value before the procedure starts executing. This activity is often known as *parameter passing* and procedure arguments are often referred to as parameters.

---

†The term local variable refers to the scope properties of the variable. Scope is covered in Section 2.4.1

Like all variables, parameter variables must have a type. Whenever a Java program is compiled, strong type checking of the match between arguments and parameter variables is undertaken to make sure that a procedure is always called with the right number and types of arguments. The type system requires the programmer to explicitly use functions in the way that they were defined to be used. This is not constraining of the programmer but is supporting the "not making errors" philosophy — a defined function cannot work as expected if it is given unexpected types of data!

The need to do type checking has a further implication for procedures. Rather than having two kinds of procedure, one of which specifies return type and the other which does not, Java requires that all procedures have a return type. This necessitates having a type indicating that there is no return value from the procedure. Java has the type `void` which is this type; `void` is a place holder for strong type checking consistency. A procedure declared as:

```
void f()
```

is a procedure with no return value. Type `void` is a perfectly valid type but there are no values of the type and, hence, no operations on them. It is not possible to have variables of type `void` as there is no representation that can be stored in them: the type only makes sense associated with functions where returning `void` simply means that a procedure will be returning no value.

2.3.4 Procedural Decomposition and Structured Programming

We now have the basis of strategy that we could employ to design and structure programs, known as *procedural decomposition* or *top-down refinement*.

When a program is constructed to solve a problem, the entire problem is first cast as a single procedure. This top-level procedure is then defined in terms of a collection of calls to other procedures, which are in turn defined in terms of other procedures, creating a hierarchy of procedures. This process continues until a collection of procedures is reached that do not need to be refined further since they are constructed entirely in terms of statements in the base language.

At this point we have a collection of procedures defining a complete program. The program is executed by calling the top-level procedure. Figure 2.2 shows an example situation in which the program Start has been decomposed into procedures P1, P2 and P3 in which Start calls P2 then P3, P2 calls P1.

As well as the code itself, we can construct what is termed a *call graph* of the program. A call graph is used to show which procedures are called by which other procedures. Figure 2.2 shows the call graph for Start. The usefulness of call graphs is that they show the structure of the source code in an immediate way that is not obtainable from perusing the source code itself. They enable the programmer to reason about their program behaviour and structure at an abstract level.

Top-down refinement using procedures is the basis of what is known as *Structured Programming*. This style of programming became very popular in the late sixties and early seventies and is exemplified in programming languages like Pascal and C. The development of structured programming was a quite natural progression of programming technology and was quite rightly popular. However, since then technology has moved on,

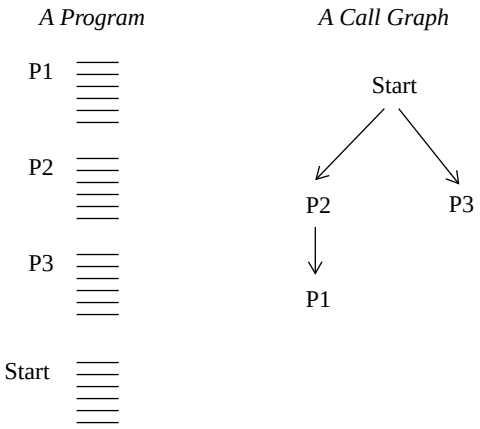


Figure 2.2 A program constructed from procedures with a possible call graph.

first by exploiting the concept of *abstract data types* and then moving to *object-oriented programming*. This led to programming languages like Smalltalk, C++, and Ada and most recently Java. Java is very much an object-oriented programming language.

Whilst it is possible to program in the structured programming way using Java, the language is really not designed to support this style. Java only really works when the full object-oriented approach is followed.

Like all object-oriented languages, Java does have the procedure construct but it is not intended for use in a procedural decomposition. To reinforce this altered use of the idea of procedural abstraction (which is still very important in object-oriented languages) procedural abstractions are called *methods*.

Before being able to uncover how object-oriented languages make use of methods, we must investigate how to organize variables.

2.4 Organizing Variables

Having considered how to structure the statements using procedural abstraction, we must now consider how to structure the variables: so far, we simply have a big collection of variables that can be accessed by any statement in any procedure.

With this unstructured collection of variables, we have to assume that all the variables come into existence when the program starts and go away when the program ends. This raises some serious problems:

- The programmer has to make, and enforce, all decisions over how variables are used and accessed. Choosing a sensible name for a variable is essential (the name should reflect the purpose of the variable). However, any variable can still be changed from any part of the program, whether or not it makes sense to do so.
- The programming language cannot properly enforce rules over the way variables are used. Human beings have limited memories and make mistakes; for any large program it becomes impossible to remember what each variable is used for. If the

programming language compiler can help enforce rules over how a variable is to be used, it leads to safer and more reliable programs.

- The number of variables is fixed. As yet, we have no way of creating variables on demand. The programmer has to know exactly how many variables will be needed by the program at compilation time.

2.4.1 Scope and Lifetime

To address these obstacles, we need two new core ideas and some mechanisms within the programming language to support them. These ideas are not limited to Java or other object-oriented programming languages, but derive from the earlier structured programming languages. The ideas we need are:

- *Scope* — defining which parts of a program can access a variable.
- *Lifetime* — defining when a variable is created and destroyed.

Every variable has a scope; that part of a program where it is named and where it can be used. Program statements in the same scope can access the variable, while statements outside the scope cannot.

Procedures and compound statements define a *local scope*. A variable declared in a local scope can only be accessed by statements in that scope, i.e. the procedure or compound statement. Thus a procedure can have *local variables* that only statements within it can access. Being declared within the procedure scope, procedure parameter variables are also local variables.

Scopes can be *nested*, so that one scope can exist inside another scope. Typically this occurs when compound statements are used within a procedure body. The inner or nested scope can access any variables declared in the enclosing scope but variables declared in the inner scope can only be accessed within that scope (or scopes nested inside it).

Using nested scopes is a way of minimising the region of a program in which a particular variable can be used. This reduces the risk of the variable being used inappropriately in part of a program that was not really meant to have access to it, so scope is a useful way of exploiting encapsulation. A good programming principle to follow is to always limit the scope of a variable to the minimum possible.

Since variables are only accessible when they are in scope, they need only exist when they are in scope. Because of this property, the language can ensure that variables only get created[†] when they are needed. Once out of scope, they can be destroyed and the memory used for the variable reclaimed. For local variables, their *lifetime* is the same as their scope.

This way of controlling resources is very important for the implementation of the idea of recursion: every time a given procedure is executed, it is a new scope and new instances of the local variables are created. When the procedure ends, that scope no longer exists and all the local variables are destroyed.

---

†Whilst we say that variables are created, what we mean is that a location in the computers memory is allocated as the place where the variable is kept. This allocation of memory for a variable is handled by the programming language not by the programmer.

2.4.2 Names and Scope

The idea of scope is not limited to variables. In fact, all names in a program have scopes — this includes variable and procedure names, as well as other kinds of names yet to be introduced.

All names used in the program must have been declared, in either the current scope or an enclosing scope, before it can be used. If this is not the case then it is a compile time error. Declaring a name associates it with a type as well as determining what kind of thing is being named.

In addition to being declared, names have to be unique within a scope†. This means that the same name cannot be declared twice in the same scope, as it won't, in general, be possible to tell which named thing is being referred to. It is possible for a name to be declared in a nested scope that is the same as one declared in an enclosing scope. In this case the nested declaration is said to hide or override the one in the enclosing scope. So for example:

```
void f()
{
    int i = 1 ;
    int i = 2 ;
    ...
    {
        int i = 3 ;
    }
}
```

The line `int i = 2 ;` is an error and the compiler will report the statement as such, refusing to compile further. The line `int i = 3;` is not an error since the declaration occurs within an enclosing compound statement and hence in a new scope. However, code of this sort is usually considered to be in poor style since the inner `i` hides the outer `i` introducing potential confusion to the reader of the code.

2.4.3 Initialization

When a variable is declared within a scope it is important to think about what value it should initially have. Given a declaration like:

```
int i ;
```

what value can the programmer expect `i` to have? With some languages the variable would actually have a random, unpredictable value. Fortunately Java has rules for initializing variables and in this case `i` would be given the initial value zero. If that is not what the programmer wants it is possible to initialize a variable at the point it is declared as follows:

```
int i = 10 ;
```

This creates the variable in the current scope and sets its value to 10. Using this approach there is no danger of the initialization being forgotten, for example if it had to be done by a separate assignment after the variable had been declared. So whilst we had been using code such as:

---

†As will be seen later, determining if a Java method name is unique in a scope also needs to take into account the argument types.

```
int i ;
i = 10 ;
```

previously, this should be considered very poor programming style. Such code should be completely avoided in favour of initialized declarations in fact. Any variable can be initialised by providing an expression following the equals sign that yields a value of the right type. Good practice dictates that variables should always be explicitly initialized whenever possible, giving a guarantee that they always start with a known and correct value.

2.5 Comments and Documentation

At the same time as considering ways of giving a program structure it is also worth introducing ways of documenting what the program does. There are two kinds of documentation beyond the actual source code of a program; that included as part of the program text, and that stored separately. The later will be discussed in Chapter 8 but it is worth a quick look at the former now.

Documentation is needed as the source code of the program (the statements) may need additional information to properly convey its meaning to those who read it. For example, it may be useful to provide a short explanation of what a variable is used for, rather than relying on just its name or trying to infer its use by the way it is manipulated by statements. It is also often useful to provide a short description of what a procedure does, what arguments it expects and what result it returns.

Documentation can be embedded in the text of a program using comments. For example:

```
int r ; // This variable will hold the result of the calculation
```

The comment is written in plain text following the `//` characters. Any text following `//` on the same line will be considered part of the comment and not part of the program source code, allowing the comment to contain any characters or words.

Java actually provides two different types of comment structure: as well as the above, the bracketing pair `/*` ... `*/` can be used to surround a comment:

```
int r ; /* This variable will hold the result of the calculation.
           This comment is a rather longer one describing this.*/
```

the comment starts with the `/*` and finishes only when a `*/` is reached. Java uses a special form of this sort of comment. Any comment starting with `/**` and finishing with `*/` is a *documentation comment*. All Java environments have a tool, `javadoc`, which supports automatic documentation generation from Java code. This tool reads through the code that it is presented with and generates World Wide Web (WWW) pages (HTML code). Documentation comments are the way of inserting documentation information about the Java code into the generated WWW pages: all other types of comment are simply ignored. Thus:

```
/**
* This is a documentation comment
*
* @see Integer
* @version 1.1
* @author Graham
*/
```

is a valid documentation comment. The first sentence of the comment is assumed to be a summary and the rest of the comment the main body of information. Within these comments you can embed HTML tags, for example to add emphasis or other typesetting features. The `@see`, `@version` and `@author` tags are special `javadoc` tags. `@see` in particular helps support the construction of inter-linked hypertext.

2.6 Summary

This chapter has outlined what a program is and, more importantly, begun to show how a program can be given structure using procedures. This structuring exploits the principles of abstraction, whereby a program is broken down into named units that allow details to be suppressed in favour of a higher level of abstraction, but more understandable description. Abstraction, in turn, enables mechanisms such as encapsulation, scope and lifetime to operate. The net result is that programs become more understandable by people, while the additional structure allows tools such as compilers to do more checking about the correctness and good behaviour of a program.