Copyright © 2011. G.C. Wells, All Rights Reserved.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies, and provided that the recipient is not asked to waive or limit his right to redistribute copies as allowed by this permission notice.

Permission is granted to copy and distribute modified versions of all or part of this manual or translations into another language, under the conditions above, with the additional requirement that the entire modified work must be covered by a permission notice identical to this permission notice.

Acknowledgements

These notes were developed by George Wells. They are loosely based on some C++ course notes originally written by Peter Wentworth, which were subsequently revised by George Wells and Peter Wentworth over a period of time.

Any errors and omissions at this stage are entirely attributable to George Wells, who would be pleased to hear about them so that they can be corrected in subsequent editions of this document.
Contents

1 Getting Started ................................................................. 1
   1.1 Introduction ..................................................................... 1
   1.2 A First Example ................................................................. 2
       1.2.1 Compiling and Running Java Programs ......................... 3
       1.2.2 Coding Conventions ................................................. 4
   1.3 Further Examples ............................................................... 4
   1.4 About Java .................................................................. 8

2 Types, Values, Variables and Expressions ..................................... 10
   2.1 Fundamental Data Types ..................................................... 10
   2.2 Variables ...................................................................... 11
   2.3 Operators and Precedence .................................................. 12
   2.4 Expression Compatibility ................................................... 13
   2.5 Explicit Type Conversion .................................................... 13
   2.6 Assignments .................................................................. 14
   2.7 Increment and Decrement ................................................... 14
   2.8 Shorthand Assignment ....................................................... 15

3 Flow of Control .................................................................... 16
   3.1 Preliminaries .................................................................. 16
   3.2 The if Statement ................................................................. 16
   3.3 The switch Statement ........................................................ 17
   3.4 The while Statement .......................................................... 18
   3.5 The do Statement ............................................................... 19
   3.6 The for Statement .............................................................. 19
   3.7 The break and continue Statements ...................................... 21
   3.8 Idioms .......................................................................... 21

4 Methods and Arguments .......................................................... 23
   4.1 The Basic Mechanism ........................................................ 23
   4.2 Overloading ................................................................... 24
11.2 Polymorphism ................................................................. 74
  11.2.1 Using Polymorphism in Practice ................................. 75

12 An Example Program ....................................................... 77
  12.1 The Frequency Table ................................................... 77
  12.2 More Flexibility .......................................................... 78

Bibliography ........................................................................ 82

Index .................................................................................... 83
The Java™ platform is a fundamentally new way of computing, based on the power of networks and the idea that the same software should run on many different kinds of computers, consumer gadgets, and other devices. With Java technology, you can use the same application from any kind of machine — a PC, a Macintosh computer, a network computer, or even new technologies like Internet screen phones.

From: What is the Java Platform?
[http://java.sun.com/nav/whatis/index.html]
Chapter 1

Getting Started

Objectives

- To introduce three complete Java programs, and comment on some of the features of the language
- To give a brief history of Java

1.1 Introduction

This is a minimal survival guide/crash course for converting those who already know a high-level programming language to Java. The objectives are to provide enough background to understand some of the principles in Java, and to transfer previously learned skills across to enable us to code simple Java programs. Many of the examples assume a prior knowledge of Python, specifically. The notes also assume the use of a Microsoft Windows operating system.

Java is a big, industrial-strength production language, and has very extensive libraries. We will cover most of the language, but will only scratch the surface of the libraries in these notes. One of the advantages of Java is its widespread use in the computer industry and the Internet, and the fact that it is big. In a course like our own, it allows us to incrementally introduce new techniques (such as object-oriented programming, or writing GUI applications), without having to start again from scratch.

Although Java was designed primarily for industrial use rather than as a minimal teaching language, safety considerations (and other good academic features) have been built into the structure of the language and the associated tools from its inception. In particular, it provides a much more forgiving and safe programming environment than C++, a closely related and very widely used programming language that you will learn more about later in the year.
Exercise 1.1 Programming languages have much in common. Write down a list of at least ten major concepts that you learned in your previous course. (e.g. looping structures, using variables, value and parameter passing ...)

1.2 A First Example

Fragment 1.1 Our first complete program

```java
/* Comment 1: */
/* Written by George Wells -- 6 January 2011 */
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello, world\n");  // Comment 2
    }  // main
}  // class HelloWorld
```

The first program every programmer ever writes in a new programming language simply prints the message: Hello, world. (This is a time-honoured tradition!)

Some points to note:

- The line numbers are not part of the Java program: they are simply provided in these notes to make discussion easier.

- String literals are always enclosed in double quotes. Strings may contain embedded escape sequences (introduced by a backslash) to represent special characters. The string "\n" contains just one character, called newline.

- Braces, {...}, are used for grouping related parts of the program together (rather than using indentation, as in Python). Here we have two such groupings: the class HelloWorld contains just one item (the method called main), and the main method contains a single statement (in general it may contain many statements, and is called a compound statement).

- Semicolons terminate every statement. Unlike Python, there can be any number of statements on a single line, or a single statement can be broken over multiple lines. However, we usually write just one statement per line (as in the example above) for clarity. In general, spaces can be freely used for indenting and separating parts of the program. Clear, consistent indentation of blocks of code is very helpful in making programs easier to read and is recommended for that purpose. However,
it is not compulsory as in Python, and the indentation (or lack of indentation) has absolutely no
impact on the meaning of the program.

- The method `main` is called when the program starts executing (it provides the entry point for the
program). The `main` method in a Java program must appear in a class, and it must always be
declared exactly as it is in line 5 of the example above (i.e. with all of the keywords `public`, `static`
and `void`, and with a parameter that is an array of strings, although this need not be called `args`).

- There are two kinds of comments, both shown in the code above (actually there are three, but we
won’t be considering the third just yet). The first uses (non-nestable) delimiters `/* ... */` which
comment out whole sections, or multiple lines, of code. The double slash form `//` comments out the
rest of the line (in the same way as the `#` form of comment in Python). The `//` form of comment
is particularly useful for marking the ends of compound statements and classes, as has been done
in the example above. While it is not really necessary in such a small program, in larger programs
where the classes and methods may extend over hundreds of lines of code and many pages on screen
this commenting technique can be very helpful.

- All input and output goes via streams. We can visualize a Java program as a black box with
an input stream called `System.in`, and an output stream called `System.out`. When the program
starts executing, the operating system connects these streams to particular devices. In our case, the
default connections are that `System.in` connects to the keyboard, so that anything that the program
reads from `System.in` actually comes from the keyboard. Similarly, `System.out` is connected to
the display screen. Anything that the program writes to `System.out` will be displayed.
Many operating systems will allow the user of the program to redirect the streams.

- As a final observation, you will note that this program is quite a lot more complicated than the
Python equivalent:

```java
print "Hello, world\n"
```

This is largely due to the fact that Python is a scripting language with dynamic typing. As a
compiled language with strong typing, Java has more structure, and hence, for very simple examples
like the one above, slightly more complexity.

### 1.2.1 Compiling and Running Java Programs

In order to run a Java program there are a couple of steps that need to be performed. The program can
be typed up in any text editor. It must be saved in a file with same name as the class it contains and
with a `.java` extension (e.g. the program above must appear in a file called `HelloWorld.java`). The
program can then be compiled by opening a Command window and executing the `javac` (Java Compiler)
program. For the example above we would need to execute the following command:

```bash
javac HelloWorld.java
```

Note that the compiler is very fussy about case-sensitivity, and the name of the file must be identical to
the name of the class which it contains.

The compiler checks the program for errors and, if it is syntactically correct, produces an executable
form of the program. This executable form is not machine code, as is usually the case for most other
languages, and the file produced is a `.class` file (`HelloWorld.class` for the example above), rather than
the usual `.exe` file. The reason for this arises from one of the key features of Java: portability. In an
Internet environment a Java program might be run anywhere on any hardware platform (a Windows PC,
an Apple Mac, a cell phone, etc.) with any operating system (MS Windows, Linux, etc.). The class files
support portability over all these kinds of platforms because they do not contain any specific machine
code or make use of any specific operating system features. Rather the class files contain bytecode. This
CHAPTER 1. GETTING STARTED

### Identifier Type

<table>
<thead>
<tr>
<th>Convention</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes and interfaces</td>
<td>Begin with a capital letter, use mixed case</td>
</tr>
<tr>
<td>Variables and methods</td>
<td>Begin with lower case letters, use mixed case</td>
</tr>
<tr>
<td>Constants</td>
<td>All upper case, with underscores</td>
</tr>
</tbody>
</table>

Table 1.1: Naming Conventions for Identifiers

bytecode is a little like the machine code for an imaginary computer. Since it cannot be executed directly by any computer hardware it has to be interpreted by a program called an interpreter. In the case of Java this interpreter is referred to as the Java Virtual Machine (or JVM for short). This name highlights the fact that the interpreter effectively acts as the imaginary (or virtual) computer on which the bytecodes run.

The interpreter is simply a program called java and can be used to execute our example program as follows (note that no file extension is required here):

```
java HelloWorld
```

Again, the interpreter is fussy about the case-sensitivity of the class name.

#### 1.2.2 Coding Conventions

We have spoken a little about the use of white space, indentation and conventions for the layout of Java programs. There are also well-established conventions for identifier names in Java, with which you should be familiar. These are spelled out in Table 1.1.

### 1.3 Further Examples

Now let’s consider a second, slightly more complicated example.

**Fragment 1.2** An example using declarations, methods and loops

```
1 public class Factorial
2 { public static int fact (int n)
3     //============================================
4     { int result=1;
5         for (int k=1; k<=n; k=k+1)
6             { result = result * k;
7             }
8         return result;
9     } // fact
10
11     public static void main (String a[])  
12     { System.out.print("Factorial of 7 = ");
13         int f = fact(7);
14         System.out.println(f);
15     } // main
16 } // class Factorial
```
Notes:

- Note the slight variations in the parameter for the `main` method (line 11). This is allowed because it has exactly the same meaning as the first form that we saw. The first form is more commonly used.

- The `=` operator denotes assignment (checking for equality is written using `==`).

- Variables must be declared in Java — you cannot simply use a new variable name as you do in Python. For example, on line 4 above, the variable `result` is declared.

- As part of the declaration, the type must be specified for all variables (and parameters, etc.). This cannot be changed, unlike Python. For example, in the program above, the variables `result`, `k` and `f` are all declared as integers (type `int`), and cannot be used to hold any other type of value.

- The type always comes before the things being declared: in the method `fact`, for example, the result type `int` comes before the method name, and the parameter type `int` comes before the parameter name.

- You can provide an initializer as part of the variable declaration, as shown in line 4 (`result=1`). This ensures that the variable always starts its lifetime with a valid, known value (the Java compiler will not allow you to use a variable that may have an undefined value).

- There is a local variable declaration in line 14. Local variables can be declared wherever they are needed. In particular, a variable has scope\(^1\) that extends from its declaration to the end of its enclosing block (denoted by braces). This allows us to declare variables with very small scopes, and so reduce the chance that some other part of the code will corrupt the variable. For example, if we need to swap two elements of an array, we could code it like this:

  ```java
  if (a[i] > a[i+1])
      { int tmp=a[i]; a[i]=a[i+1]; a[i+1]=tmp; }
  ```

  In this case the scope of the variable `tmp` is confined to a short statement block that happens to be no more than a line long.

  Similarly, the loop control variable `k` is declared as part of the `for` loop header and consequently its scope is restricted to the extent of the `for` loop (i.e. lines 5–7).

- Methods cannot be nested in Java (just as functions and procedures cannot be nested in Python, for that matter).

- Java has only one parameter passing mechanism: by value. This means that the code in a method cannot alter the value of the actual parameter that was passed to it. This gets slightly more complicated when we consider how objects are passed as parameters, but we will come to that later.

- The `for` statement in Java has three controlling expressions, separated by semicolons. The first initializes the loop, (in line 5 we declare `k` and assign the value 1 to it). The next expression is the termination test (this can be any boolean expression). The `for` statement is a pretest loop which only executes the loop body if the test expression is true. Finally, the third expression is executed at the end of every iteration (often it simply increments or decrements the loop counter). A more formal presentation is given in Chapter 3, but for now note that the loop on line 5 is equivalent to `for k in range(1, n+1)` in Python.

\(^1\)The scope of a variable is that region of the code in which the variable is “visible”, or can be accessed.
• On line 13 we have used `System.out.print`. This method works in exactly the same way as `System.out.println` to produce output, except that it leaves the cursor at the end of the line so that any subsequent output follows on immediately on the same line.

For comparison, here is the equivalent Python program, which deliberately follows the same design as the Java version above:

**Fragment 1.3 A Python Factorial Program**

```python
def fact(n):
    result = 1
    for k in range(1, n+1):
        result = result * k
    return result

def main():
    print "Factorial of 7 =",
    f = fact(7)
    print f

if __name__ == '__main__':
    main()
```

**Exercise 1.2** Write a Java program which generates three random numbers using `Math.random()`, and prints their sum.

One of the main features of Java is that it is object-oriented, but we have not really seen this aspect in any of the prior examples (although there have been objects lurking quietly in both the programs so far). The next example highlights the use of objects in Java.
CHAPTER 1. GETTING STARTED

Figure 1.1: A reference to an object.

Fragment 1.4 An example showing the use of an object

```java
1 class Factorial
2 { public int fact (int n)
3     //=======================
4         int result=1;
5         for (int k=1; k<=n; k=k+1)
6             { result = result * k;
7                 }
8         return(result);
9     } //fact
10 } // class Factorial
11
12 public class Program3
13 { public static void main (String args[])
14     //=======================
15     { Factorial f = new Factorial();
16         System.out.println("Factorial of 7 = " + f.fact(7));
17     } // main
18 } // class Program3
```

Notes

- In this program we have two classes: one called `Factorial` and one called `Program3`. The `Program3` class simply contains the `main` method, used as the entry point for the program. The important part of this program, from an object-oriented perspective, is in line 15 where we create a new object. We have used the class name `Factorial` as if it was a type, and declared a variable `f` of this type. The `new` keyword then creates a new object from the `Factorial` class. This object is then used in the next line when we call the method `fact` in the `Factorial` class. Note how we need to give the name of a variable that refers to a `Factorial` object (`f` here) in order to do this.

Java uses some specific terminology here which we need to get accustomed to. Firstly, the variable `f` is called a `reference variable`. The reason for this is that it is used to "refer to" an object (specifically a `Factorial` object in his example). In line 15 it is assigned a reference to a specific `Factorial` object (the one created by the `new` operator). However, at other times in a program it may be used to refer to other `Factorial` objects. The object itself is referred to as an instance of the `Factorial` class, and the process of creating a new object is sometimes called `instantiation`.

It sometimes helps to picture what is happening in such cases diagrammatically. For the example above we can picture the situation after line 15 as shown in Figure 1.1.

- In line 16 we also show how multiple values can be printed out using `println`. This is simply a case of using the `+` operator to concatenate the different components of the output string.
• This program should be saved in a file called \texttt{Program3.java} since \texttt{Program3} is the class that contains the \texttt{main} method. You will also notice that the \texttt{Program3} class is declared as \texttt{public} while the \texttt{Factorial} class is not. This is important as there can be only one public class in a file (there may additionally be any number of non-public classes), and the file must be named after the public class. In general, a good rule of thumb for beginners is to put every class in a separate file named with the class name.

1.4 About Java

The Java language was first announced on 23 May 1995, by Sun Microsystems, who developed the language. In a very short space of time it became very widely used and very popular (most other programming languages have taken at least ten years to achieve the kind of success that Java had reached in only two or three). Today most big computer companies have put considerable effort into supporting Java.

Java’s roots go back a little further than 1995 to a project called the Green Project that Sun began in 1991. The Green Project was aimed at producing a sophisticated “set-top box” (something like an interactive DSTV decoder). A programming language called Oak was developed as part of the Green Project. In about 1992/1993 it became apparent that the Green project was far ahead of its time (interactive set-top boxes are only starting to come into use now) and the Green team started to look around for other ways to use the technology that they had developed.

At this time general use of the Internet was starting to take-off and the Green team realised that the work that they had done on the Oak language met many of the requirements of a programming language for use in the Internet environment. With this in mind they began the development of the Java language and the rest, as they say, is history.

A very deliberate decision was taken to base the syntax of Java on that of C++. The reasoning for this was that there were a lot of C++ programmers around, and if the syntax of Java was similar it would be relatively easy for them to learn Java. At the same time, C++ is a language that is full of pitfalls and problems, particularly for beginning programmers, and Sun was careful to address these and to come up with a language that avoids these problems as far as possible.

The first version of Java was 1.0 and had some very “rough edges”. This was partly due to the fact that Sun wanted to ensure that they were the first to get a product onto the market, as to come along second in the computer industry is often fatal. As a result, version 1.1 was released fairly soon and addressed a lot of the problems (many of them quite superficial) in version 1.0. Version 1.1 was around for a long time and went through many sub-versions (ending up at 1.1.7), during which there was a lot of development, particularly on the libraries. Developing Java programs during this period was quite chaotic, as programmers had to keep track of the different versions of the libraries and the development system and which versions of which libraries worked with each other and with which version of the development system!

Fortunately Sun recognised the problem and late in 1998 they released version 1.2 in which they integrated all of the library development that had taken place during the evolution of version 1.1. They also committed themselves to stabilising the development platform from that time on, and did so. In addition to the “Standard Edition” (SE), Sun also released Enterprise Editions (EE) for large businesses, and Micro Editions (ME) for small devices like cell phones. We will only be considering the Standard Edition features in this course. With the release of Java 1.2, some marketing idiots messed up the naming and referred to the product as “Java 2” (to further confuse the issue, versions 1.3–1.5 were still called “Java 2”, not “Java 3/4/5” as you would expect!). Since version 1.6, the naming uses the secondary version number. The current version of Java at the time of writing these notes is “Java SE 6 Update 22”, or just Java 1.6\textsuperscript{2}.

\textsuperscript{2}To prevent any confusion these notes will use the 1.x version numbers where we need to refer to the different versions.
Recently, Sun Microsystems was bought by Oracle Corporation, and so there has been some rebranding of Sun products. So far this has had not had any significant impact on the Java Development Kit.

**Skills**

- You should be able to mimic the simple example programs and write small Java programs involving integer variables and output
- You should be able to work with simple Java objects, and use the correct terminology to describe the use of the object-oriented features of Java
- You should know the history of Java, and understand the relationship between C++ and Java
Chapter 2

Types, Values, Variables and Expressions

Objectives

- To introduce the fundamental building blocks of Java
- To understand the idea of type compatibility in Java
- To learn that Java allows expressions to be used as statements
- To become familiar with some of the shorthand notations in Java

2.1 Fundamental Data Types

As we have already seen, the handling of variables, types, etc. in Java is quite different to the way in which these are handled in Python. Java makes use of what is known as static typing where the type of all variables, parameters, etc. is explicitly declared by the programmer. This allows the compiler to check that these variables are being used consistently and correctly.

The three important types that we begin with in Java are:

- int. Its values are signed integer numbers, such as 123, -27, 1234567. This is similar to the int type in Python.
- double. Its values are floating point numbers that conform to the IEEE standards. (The standards define all sorts of behaviour such as what happens when overflow occurs, how much precision is required, how rounding takes place, how errors are propagated through expressions, and so on. Most scientific or floating-point software adheres to these standards). Some double literals are 3.1415926, -10.0E05, 1.7E-6. This is similar to the float type in Python.
- String. This type is similar to the str type in Python. However it differs from the int and double in that it is an object type, not a primitive type. We’ll restrict ourselves to considering the basic
ASCII character set, but Java does support extended character sets (e.g. Greek, Cyrillic, Kanji, etc.) through the use of Unicode characters. Printable string literals are enclosed in double quotes, as we have seen in the last chapter, and escape sequences may be used to denote non-printable characters. Some of the useful escape sequences that are recognized are:

- \n  Newline
- \t  Tab
- \\  Backslash
- \"  Double quote
- \'  Single quote

Examples: "George"  "s"  "Hello\n"  "C:\Windows"

The statement:

```
System.out.println("He said "That’s rubbish" and quit.");
```

prints:

He said "That’s rubbish" and quit.

Note how the escape sequence \" is necessary in this string to differentiate it from the string delimiters, but we do not need to escape the single quote.

**Exercise 2.1** Show what is printed by:

```
System.out.println("Mary had a little lamb.\nIts fleece was" + " white as snow.");
```

There are actually six other primitive types in Java. There are also integer types byte, short and long, and another floating point type: float. In general you should stick to the use of int for integer values and double for floating point values, unless there are good reasons to do otherwise.

There is a type used for single characters, char, and a type used for logical (true/false) values, boolean (the same as Python’s bool type). Character literals are written using single quotes, and may make use of the same escape sequences as strings (e.g. ‘a’, ‘?’, ‘\n’). The boolean type allows only two literal values: true and false (note the lower-case spelling, unlike Python).

Table 2.1 summarises the properties of the eight primitive types. Unlike many other programming languages, these properties are guaranteed to hold for all Java systems.

### 2.2 Variables

We have already seen how to define variables in the previous examples. The most important points are:

- Like Python, in Java names are case-sensitive. For example, this means that Count and count are considered to be two different variables.

- The list of variable names come after the type. For example, double i, j; long u;
CHAPTER 2. TYPES, VALUES, VARIABLES AND EXPRESSIONS

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bits)</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>-128 – 127</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>-32 768 – 32 767</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>-2 147 483 648 – 2 147 483 647</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>-9 223 372 036 854 775 808 – 9 223 372 036 854 775 807</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>-3.40292347E+38 – 3.40292347E+38</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>-1.79769313486231570E+308 – 1.79769313486231570E+308</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
<tr>
<td>boolean</td>
<td>8</td>
<td>true and false</td>
</tr>
</tbody>
</table>

Table 2.1: Properties of the Primitive Data Types

- Unlike Python, Java uses strong type-checking, and the compiler will object if you assign different types of values to the same variable (see Section 2.4 below).
- Each defined name may have an optional initialization expression. For example,
  ```java
  int i = 7, j = 9;
  String name = "George";
  ```
- Variables can be declared where they are needed. Their scope is from the point of declaration to the end of their enclosing block.

2.3 Operators and Precedence

Java provides a rich set of operators, with a complicated system of different levels of precedence, and some prefix, infix, and postfix grouping rules. We’ll skip most of this and just get a small working set together. The convert from other languages is well advised to make liberal use of parentheses until the differences in precedence and associativity are properly understood.

The basic arithmetic operators are *, /, +, -. They operate on any of the integer or floating types. When applied between two integer operands, the division operator / truncates integer results (e.g. 11 / 4 gives the result 2).

The mod (remainder) operator works only on integer types, and is written as a % b, just like Python.

The six relational operators are ==, !=, >, <, >=, <= (do not forget that = is the assignment operator, not the equality operator).

The logical operators for combining Boolean expressions are && (and), || (or), and ! (not). Short circuit evaluation applies in compound Boolean expressions.

Fragment 2.1 Some examples

```
1 while (x != 100)
2 ...
3 if (a < 0 && b == 'Q')
4 ...
5 if (! s.equals("hello")) // Compare two strings
6 ...
```
2.4 Expression Compatibility

Unlike Python, Java has a fairly strict policy with regard to mixing different types in expressions and assignments. Some conversions are considered “safe” and will be applied automatically. For example, if we mix an int and a double operand in an expression, Java will automatically convert the int operand to a double. During expression evaluation, automatic type conversion (also called type promotion), favours the types with adequate precision. Thus, mixing a short and an int will convert the shorter one to the bigger one, while mixing an int and a double will convert the int rather than the double.

With assignment (or passing a parameter to a method), the Java compiler will automatically handle the conversion if it is a promotion. It will not automatically handle demotion.

As might be expected, converting a floating-point number to an integer will discard the fractional part. This conversion, and any others that go from “big” types to “small” types, can cause problems if the source value is too large to be represented in the target format. The compiler will not handle these cases automatically, but will give an error message. It is possible to force the compiler to do the conversion (see the next section for details of this), in which case you might get unexpected results if the values are not compatible.

2.5 Explicit Type Conversion

In addition to default conversions (promotions), Java provides a mechanism whereby the programmer can force explicitly when and how type conversion is to take place. We call this a cast, or a type-cast, and we write it as:

\[(\text{desired-type}) \text{ expression}\]

This has the same effect as the type conversion functions in Python, which use the different syntax:

\[\text{desired-type(expression)}\]

For example, to convert a double value to an integer:

\[
\begin{align*}
\text{int } k; \text{ double } d; \\
\ldots \\
\text{k = (int)d; // Convert, losing fractional part}
\end{align*}
\]

Exercise 2.2 What is printed by this program?

```java
1 int x, y, sum = 675, num = 10;
2 x = (int)(1.6 + 1.7);
3 y = (int)1.6 + (int)1.7;
4 System.out.println(x + "\n" + y + "\n" + 
5 "average is " + (double)sum / (double)num);
```
2.6 Assignments

Most languages make a clear distinction between statements (which perform actions), and expressions (which yield values). In Java, however, the values of expressions can be used in further expressions, or in some cases can simply be ignored. This blurs the distinction. The most common example of an expression masquerading as a statement is the assignment. In Java, the assignment operator updates a variable (the left operand), giving it the value of the right operand, and finally returns this updated value as the value of the expression as a whole. In most cases this returned value is discarded, but we will find this sort of thing quite often:

Fragment 2.2 Assignments can be chained
1  int i, j, k;
2  i = (j = (k = 0)); // all three variables become 0
3  i = j = k = 0;   // the parentheses above are redundant

2.7 Increment and Decrement

There are a number of shortcuts in Java (most of them inherited from C and C++). Incrementing or decrementing a variable is so common that special operators exist: the increment operator ++ and the decrement operator --. What is more, these come in two flavours, a prefix version, exemplified by ++num, and a postfix version, num++. The idea is that an expression may use the value of a variable (num in this case), and also increment it or decrement it in the same operation. The prefix version says “Increment num, then use its new value”. The postfix version says “Use the old value of num, then increment it”. Incrementing something is sometimes called bumping it.

Fragment 2.3 Examples of the use of ++ and --
1  int i = 5, j, k;
2  i++;  // i is now 6
3  j = i++;  // i is 7, j is 6
4  k = --j;  // j is 5, k is 5

Exercise 2.3 What does this fragment of program print?

```java
int n = 0, m = 10;
while (n < m)
    { m = m - 1;
      System.out.println(n++ + " " + m);
    }
```


\section*{2.8 Shorthand Assignment}

Many assignments in programs are of the form:

\[ \text{lhs} = \text{lhs op rhs} \]

and this form of assignment has a special notation in Java: instead of writing the \text{lhs} out twice, we instead use operators of the form \text{op=}. In the code fragment below, the two statements on each line have the same effect, except that in the shorthand case the expression \text{lhs} is evaluated only once.

<table>
<thead>
<tr>
<th>Fragment 2.4 Matching statements and their shorthand equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 \hspace{1em} \text{i = i + 10;} \hspace{1em} i += 10;</td>
</tr>
<tr>
<td>2 \hspace{1em} \text{i = i * 10;} \hspace{1em} i *= 10;</td>
</tr>
<tr>
<td>3 \hspace{1em} \text{u = u</td>
</tr>
<tr>
<td>4 \hspace{1em} \text{a[n] = a[n]/2;} \hspace{1em} a[n] /= 2;</td>
</tr>
</tbody>
</table>

\section*{Skills}

\begin{itemize}
  \item You should be able to define variables and use literals of the most common Java types
  \item You should be able to use the elementary Java operators, to construct more complex expressions
  \item You should understand what will happen when you mix different types in expressions
  \item You should be able to use a cast to force specific type conversions when you need them
  \item You should be aware of the distinction between a statement and an expression, and be able to take advantage of the fact that Java blurs the difference
  \item You should be able to use the shorthand operators for incrementing, decrementing and common assignments
\end{itemize}
Chapter 3

Flow of Control

Objectives

- To introduce the Java control structures

3.1 Preliminaries

In your previous language we assume you have learned the fundamental use of conditionals, loops and so on. This chapter presents the Java syntax without re-covering those basic ideas.

In Java most of the control structures like `for`, `while`, `if`, and `case` only operate over a single statement. This is not usually good enough — most often we’ll need many statements in the body of our loops, or in one of the alternatives in a conditional statement.

In Java we can use braces to enclose a group of statements, and the resulting construct acts as a single statement. We find it more consistent and ultimately easier if we always use the braces, even in the cases where there is only one statement and the braces could be omitted. The syntax used in the examples below follows this style.

There are also a number of different layout conventions in Java, and every author has a favourite. Because layout carries subtle and important information when reading code, it is a good idea to learn and use one of the widely-used conventions (these notes use my preferred convention throughout).

3.2 The if Statement

The Java `if` statement is directly equivalent to the Python `if` statement.
CHAPTER 3. FLOW OF CONTROL

Fragment 3.1 The if statement

```
  1    if (expr)
  2       { statement; ... 
  3           }
  4    else
  5       { statement; ... 
  6          }
```

No surprises here, (unless we forget those necessary parentheses around the condition). The \texttt{expr} part must be a boolean expression giving a true/false result (unlike C++ where an integer value will do). As in Python, the \texttt{else} part can be omitted entirely if it is not needed. Unlike Python, there is no \texttt{elif} clause, and \texttt{if} statements must be nested to get a similar effect.

### 3.3 The switch Statement

The \texttt{switch} statement allows us to select between many different alternatives. It can sometimes be used in place of multiple \texttt{if} statements. Note that Python has no statement like this.

Fragment 3.2 The switch statement

```
  1    switch (expr)
  2       { case label: ... case label: statements; break;
  3           case label: ... statements; break;
  4           case label: ... statements; break;
  5           default: statements; break;
  6          }
```

A tricky point here is that each switch case has to be explicitly terminated by a \texttt{break} statement, otherwise control simply continues executing the statements in the next leg of the switch. (This is sometimes useful and can be exploited as a feature, but we are more likely to forget the \texttt{break} and create a bug than we are to find a good reason for leaving it out!).

The selecting expression and the case labels can, unfortunately, only be simple integral or character types. It is not possible to use a string or a double expression as a selector. In addition, the labels must be simple literals or constants — they cannot be variables.
CHAPTER 3. FLOW OF CONTROL

Fragment 3.3 Examples of the switch statement

```java
switch (num)
{
    case 0:
        System.out.print("Zero"); break;
    case 1: case 3: case 5: case 7: case 9:
        System.out.print("Odd"); break;
    case 2: case 4: case 6: case 8:
        System.out.print("Even"); break;
    default:
        System.out.print("Not 0-9"); break;
}

switch (ch)
{
    case 'a': case 'A':
        doA(); break;
    case 'b': case 'B':
        doB(); break;
    case 'q': case 'Q':
        quit(); break;
    default:
        System.out.println("Unrecognised option"); break;
}
```

Exercise 3.1 Write Python code fragments equivalent to these examples of Java switch statements (hint: you will need to use if statements).

3.4 The while Statement

Again, this is very similar to the equivalent Python statement.
CHAPTER 3. FLOW OF CONTROL

Fragment 3.4 The while statement

```java
while (expr)
{
    statement; ...
}
```

While the expression is true (again, `expr` must be a boolean value), the loop will re-execute. It is a pretest loop. Note again that the test expression must be parenthesized.

3.5 The do Statement

The `do` statement in Java has no equivalent in Python.

Fragment 3.5 The do statement

```java
do
{
    statement; ...
} while (expr);
```

This is a post-test loop that executes at least once. It terminates when the value of the expression becomes false.

3.6 The for Statement

The Java `for` statement is used in similar ways to the `for` statement in Python. However, it works slightly differently, and it is considerably more flexible.

Fragment 3.6 The for statement

```java
for (expr1; expr2; expr3)
{
    statement; ...
}
```

The `for` loop has three controlling expressions. Note the semicolons that separate them. Any of these can be empty, but the semicolons must always be present.

The first expression (initialization) starts the loop, and is usually a simple assignment statement like `k=0` — but it could be a statement to open a file, or to create a window, or to set up an FTP connection. It is executed exactly once, before any other part of the loop.

If present, the second expression (the test condition) is executed directly after the initialization expression, and then before each iteration of the loop. It must be a boolean expression. If it evaluates to true, the loop body is executed again. (If it is omitted, it is taken to be permanently true and the loop is an infinite loop unless it is terminated in some other way).

At the end of executing the statements in the body of the loop, the third expression is executed. Its purpose is to provide whatever is needed to advance to the next iteration: increment the loop counter; or read the next record from a file; or fetch another file via FTP.

The `for` loop is dynamic — that is, the second and third controlling expressions are re-evaluated afresh on every iteration.
Fragment 3.7 Some examples of the for statement

```java
1 for (int k=0; k < 10; k++) // write out values 0..9
2 { System.out.println(k);
3 }
4
5 sum = 0;
6 for (int k=0; k < 10; k=k+1) // sum elements in array a
7 { sum += a[k];
8 }
9
10 sum = 0;
11 for (int b=1; b <= 64; b=b*2) // sum powers of 2
12 { sum += b;
13 }
```

Exercise 3.2 Write the loops in the code above as while loops which have the same behaviour.

Exercise 3.3 Translate the general form of the for statement,

```java
for (expr1; expr2; expr3)
    { body; ... 
    }
```

into a while loop with the same behaviour.
3.7 The break and continue Statements

The `break` and `continue` statements work in the same way as their Python counterparts. A `break` statement transfers control past the end of the most closely surrounding `for`, `while`, `do` or `switch` statement. Its main use is for forcing an “early exit” from a loop, and for terminating the execution of each leg of a `case` statement. A `continue` statement is used only with the looping constructs. It is similar to `break` in that it disrupts the usual flow of a loop, but not quite as drastic: it simply jumps to the loop test and forces the re-evaluation of whether the loop should continue or not.

In both cases the `break` or `continue` can be used very simply to affect the innermost (or closest enclosing) loop (or `switch`, in the case of `break`) construct. However, it is sometimes necessary to jump out of nested loops, or to re-evaluate the outer loop condition from within nested loops. To allow this to be done, Java allows the looping statements to be `labelled`, and the `break` and `continue` statements can optionally specify the label of the loop which is to be affected. A label is simply an identifier name followed by a colon, appearing before the loop.

```
Fragment 3.8 The use of break and continue
1    for (...) 
2       { statements; ... 
3           if (...) 
4               break; // Exit loop, carry on from line 7 
5               statements; ... 
6           } // for
7    while (...) 
8       { statements; ... 
9           if (...) 
10              continue; // Re-evaluate loop, go back to line 1 immediately 
11             statements; ... 
12       } // while
13    myLabel: for (...) 
14       { statements; ... 
15           while (...) 
16              { statements; ... 
17                  switch (...) 
18                  { ... 
19                      break myLabel; // Exit for loop, go to line 13 
20                  ... 
21                 } // switch 
22             statements; ... 
23         } // while 
24       } // for
```

3.8 Idioms

Over the years certain patterns of usage in C and C++ became well established. A number of these have become widely used in Java too. The meanings of some of these constructs are not always obvious to the beginner, but any experienced Java programmer immediately knows what they mean and what they do.
These fragments are called *idioms*, and one of the tasks in learning Java is to become familiar with its idioms.

We sometimes hear the complaint that Java is cryptic and hard to read. Fluency in any language (e.g. French, music notation, Java) does not come from reading word-at-a-time or note-at-a-time. It comes from practice and exposure, until we can read phrases or sentences without having to analyse the individual components. Experienced programmers understand the idiomatic usage of the language at the level of “phrases” rather than by understanding the individual “words”.

We present two popular idioms for middle-exit loops in Java. (The language has no explicit middle-exit looping construct).

**Fragment 3.9 Two common idioms**

```java
1   while (true) // the preferable way of doing a middle exit.
2       { ...
3       if (e) break;
4       ...
5     }
6
7   for (;;) // ugly, but seen quite often.
8       { ...
9       if (e) break;
10      ...
11    }
```

The second form above works because the test part of a `for` loop is assumed to always be true if it is omitted.

**Skills**

- You should be able to use `if`, `switch`, `while`, `do`, `for`, `break` and `continue` constructs in Java
- You should understand the idea of a language idiom
Chapter 4

Methods and Arguments

Objectives

- To introduce the Java method mechanism
- To review call-by-value argument passing
- To understand the role of \texttt{void}
- To understand overloading of methods

4.1 The Basic Mechanism

Java allows the programmer to define new methods\footnote{In Java, all “functions” are called methods: there is no distinction between “methods” and “functions” as in Python.} that can be called from other sections of the program. A method heading specifies some formal parameters and a \textit{return type}: this allows us to pass arguments from the \textit{caller} to the \textit{callee}, and to pass results back from the method to the caller. Note that there is no keyword equivalent to \texttt{def} in Python.

All methods in Java \textit{must} appear inside a class. However, for simplicity, many of the examples below will not show this.

As a programming tool, methods are invaluable. They allow us to break our program into smaller, manageable sections of code, each of which does a very specific task. They are also an essential feature of any object-oriented language.

Our first example function calculates the number of pixels in an image which is $h$ pixels high and $w$ pixels wide. Notice in particular how the two arguments are declared and separated by a comma, how the return type of the method is given, and how the result is returned to the caller.
CHAPTER 4. METHODS AND ARGUMENTS

Fragment 4.1 A simple method with two arguments

```java
1 int numPixels (int h, int w)
2 { return (h * w);
3 } // numPixels
```

This is equivalent to the following Python function:

Fragment 4.2 A simple method in Python

```python
def numPixels (h, w):
    return (h * w)
```

Many other languages have separate mechanisms for functions and procedures. Java uses just the one mechanism, and we have a special keyword called `void` which can be used in place of the usual return type. We call such a method a `void method`, and it takes the place of a procedure in other languages. In a `void` method the `return` is not necessary, but if one is provided, it should not specify any return value. Here is a `void` method that prints the time in a fancy format:

Fragment 4.3 Example of a void function

```java
1 void printTime (int hrs, int mins)
2 { System.out.print("The time is " + hrs + ":" + mins);
3 } // printTime
```

At this stage we note that:

- Java methods may *not* be nested within one another
- Java methods may call themselves (recursion) or other methods
- Arguments in Java can only be passed by value

4.2 Overloading

*Overloading* is the term used to describe a name or operator that can have more than one meaning, depending on context. Most languages make use of this idea — for example, the operator `+` can mean “add two integers” or “add two reals”, or even something like “take the union of two sets” or “concatenate two strings”.

One of the powerful features of Java is that the overloading is not hard-wired into the language. The programmer can create new overulings. More than one method can have the same name but with different argument types, and the system will choose whichever version of the method is appropriate. For example, we might want to have a method to find the maximum of two `double` values, or two `long` values. By re-using the most appropriate name `max` for each of these our code becomes more readable, and we reduce the so-called “name-space clutter”:
Fragment 4.4 Overloading allows many functions with the same name

```c
1 int max (int u, int v) // Find maximum of two ints
2 { if (u > v)
3     return u;
4     return v;
5 } // max

7 double max (double u, double v) // Find maximum of two doubles
8 { if (u > v)
9     return u;
10    return v;
11 } // max

14 char max (char u, char v) // Find maximum of two chars
15 { if (u > v)
16     return u;
17     return v;
18 } // max
```

Resolving the overloading (i.e. working out which of the many methods with this name should be called) takes place according to the types and number of the actual arguments. If the number and types of the actual arguments in the call exactly match one of the candidates, that method is used. Otherwise the system uses its automatic conversion rules: if after conversions there is only one possible match then the overloading is resolved and that method is called. If more than one method matches after conversions (or if none do) it is an error. Note that the return type of the method is not considered at all when resolving the overloading.

For example, assume we call a method \( f \) with two integer parameters (e.g. \( f(i, j) \)). If there were a method like \( \text{void } f \ (\text{int } x, \text{int } y) \ {...} \) available then it would match exactly, and would be called. If that method were not available, but there were other ones in scope, the system would try to convert the actual arguments. If a method like \( \text{void } f \ (\text{long } m, \text{long } n) \ {...} \) was in scope it would match successfully since an \( \text{int} \) can convert automatically to a \( \text{long} \) value.
Exercise 4.1 Write Java methods that allow the display of a row of n repeated characters followed by a newline using System.out. By default, the characters should be asterisks (*), but we should be able to supply another character if we want that one printed instead.

Skills

- You should be able to write Java methods with arguments
- You should be able to use overloading to provide easy-to-use intuitive methods
Chapter 5

Constants

Objectives

- To introduce the final modifier for variables and explain its usage

The keyword final can be added to the declaration of an entity to indicate that the entity may not be modified.

```java
1 final double PI = 3.1415926;
2 final int[] DAYS_IN_MONTH = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
3 int sumArray (final int n, final int[] a) {...}
```

The fact that the formal parameters in line 3 are final guarantees that they will not be modified in the method. This is a little pointless in Java, as the call-by-value parameter passing mechanism means that such changes are limited to the method in any case. Also, even although the array parameter `a` is marked as final, the contents of the array can still be altered — it is only the reference itself that is being protected.

An attempt to modify final entities is an error. Java allows the programmer to separate the declaration of the final variable from the point where it is given a value, but usually this is done in a single step as in the examples above.

Note that Python has no equivalent feature.

Skills

- You should be able to make use of constants in your programs
Chapter 6
Classes and Encapsulation

Objectives

- To cover the concept of a class as a mechanism for:
  - creating new data types: groupings of related data and operations,
  - creating an Abstract Data Type (ADT) in which the client's facilities are isolated from the implementor's mechanisms.

6.1 The Class: Definition and Use

An absolutely essential requirement of a good programming language is the ability to combine a number of variables into a bigger structure that can be treated as a new type. For example, we want to be able to build new types like student, or car, or scroll bar, or menu, or bank account. Each of these is an example of an aggregate data type\(^1\). The particular variables that will need to be stored will depend on the application. In Java, these variables are called data members, although the terminology attributes, or properties, or fields is common too.

In Java, we use a class to define a new type which consists of one or more different data members. In these notes we use a simple form of UML diagram with three sections to describe the classes:

```
<table>
<thead>
<tr>
<th>ClassName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Members</td>
</tr>
<tr>
<td>Methods</td>
</tr>
</tbody>
</table>
```

You should ignore the third (greyed-out) section of the class diagram for the present.

\(^{1}\)aggregate means the combination of a number of separate parts into a new whole.
CHAPTER 6. CLASSES AND ENCAPSULATION

Exercise 6.1 What are some of the obvious data members that are likely to make up a student record at a typical university?

Exercise 6.2 A program might use one or more windows. What are some of the data members that are likely to be represented within the class for a GUI window? Draw a class diagram for the class.

We make an important distinction now: when we extend our programming language by defining new types and operations on those types, we are implementors of the new type. We distinguish this activity from using the new types within a program. In fact, we want this distinction to be as clear as possible, so we introduce a new term to refer to someone (or some part of a program) that uses a new type. We call the latter a client of the type.

6.1.1 Defining a New Class: The Implementor’s Side of Things

A simple push button control such as those used in many Windows programs has attributes such as its current state (up or down), the caption displayed on the button, the position of the button, its height and width, and a visibility component determining whether the button is currently visible or not. (There are many more attributes too, such as colour and font, but we ignore those for the moment). Here are the class diagram and definition for that button. (For the moment, pay no attention to the public specifier in the code: it will be explained shortly).

```
Button
state
caption
x, y
w, h
visible
```
CHAPTER 6. CLASSES AND ENCAPSULATION

Fragment 6.1 A simple class definition

```java
public class Button
{
    public final static int UP = 0;
    public final static int DOWN = 1;

    public int state;
    public String caption;
    public int x, y, // position
            w, h; // size
    public boolean visible;
}
```

6.1.2 Using a Class: The Client’s Side of Things

A class, whether provided as part of a language or defined by some implementor, provides the means to create (or instantiate) objects of that class. From a single class or type definition one can define many variables or objects. It is important to realize that a class definition alone does not create any objects: it merely describes a new type.

So the first thing a client has to do is to create a few objects of the desired type. The syntax for instantiating new objects and using reference variables is precisely the same as we met before.

```java
Button b1 = new Button(),
        b2 = new Button(); // Create two button objects: b1 and b2
```

Now the client might need to access the attributes of the objects, which is easy: the member selector notation is the same in Python and in most other languages.

```java
b1.visible = true; // Make button visible
...
if (b2.state = Button.DOWN) { ... }
```

6.2 Types Are More Than Just Data

One of the important ideas in programming is that certain types of data or objects have related operations that are unique to those objects. For example, if we have a window, we might want to resize it, iconize it, move it to some other position on the screen, or bring it to the front of the desktop. A pack of playing cards would have related operations like shuffle or deal.

We could write some functions to do each of these operations on a window or pack-of-cards type: in fact, this was how programming was done in the old days. But this approach really treats the data parts of the objects and the operations separately. Quite early in the history of Computer Science we realized that a type is more than just its data: the current technique is to bundle the data and its associated operations together into a single “thing”, usually called an object.

The process of bundling together these two aspects is called encapsulation. Java uses the class mechanism for encapsulation of types.
6.3 Adding Methods to a Class

Two operations that we would naturally want to have on a button would be to press and to release the button.

If you have watched carefully when using your GUI, you will have noticed that the three-dimensional effect of pressing the button is achieved by having two different pictures (bitmaps) for the button. In Windows, for example, a light source is assumed to come from the top left, and all the three-dimensional icons and their shading and shadows are designed accordingly. When the button is up, it is brightly lit, especially at the top and left edges. When the button is pressed, the 3-D effect is achieved by putting shadows onto the top and left sides, highlights onto the bottom and right sides, and the face of the button becomes darker. Any text or picture on the button also moves to the right and down by one pixel, to reinforce the illusion that the button moves “into” the screen. Of course we’re not becoming graphical designers in this course, but the implementor needs at least two bitmaps to achieve the desired visual effect.

We’ll now add the two bitmaps to our class definition, and we’ll add methods to display the bitmap, and to press and release the button. In our diagrams, the methods (or function members) are given in the third section of the class:

```
Button

state
caption
x, y
w, h
visible
upMap, downMap

displayBitmap
press
release
```

Here is an improved Button class showing the new features.


**Fragment 6.2** Adding methods to the class

```java
public class Button {
    public final static int UP = 0;
    public final static int DOWN = 1;

    public int state;
    public String caption;
    public int x, y, // position
            w, h; // size
    public boolean visible;
    public Bitmap upMap, downMap;

    public void displayBitmap ()
    { if (state == UP)
        ... // display upMap
    else
        ... // display downMap
    } // displayBitmap

    public void press ()
    { state = DOWN;
        displayBitmap();
    } // press

    public void release ()
    { state = UP;
        displayBitmap();
    } // release
}

// class Button
```

Note that the methods do not require an explicit “self” parameter in Java, and the data members can be referred to directly in the methods — for example, as `state` (in Python this would have to be `self.state`). In Java we can write `this.state`, but we do not have to do so — `this` in Java is equivalent to `self` in Python, and is a reference to the object for which the method is being called, but it is handled automatically and transparently by the compiler, rather than explicitly.

### 6.4 Abstract Data Types via Access Control

Now we need an important extension. The bitmap members and the method which displays the appropriate bitmap are aspects of the *internal* implementation of the type, whereas the operations to press or release a button are intended for the client.

There are some important advantages to be had by distinguishing between the external interface of a type (the facilities that a client may use), and its internal representation. The main one is that the client operates via the interface, and needs no knowledge of the internal mechanisms that implement the features. For example, the client of a pack-of-cards type might be able to shuffle the cards, display a card, or determine whether two cards belong to the same suit. But these operations tell us nothing about the *internal implementation* of the type. One implementor may elect to represent a pack of cards as an array of integer values; another implementor may use a very different internal structure.
This provides the freedom for the implementor and client to develop their sections of the program independently of each other. Organizing our program via interfaces like this is also a natural way to break a big complex task into manageable subtasks that can be tackled separately.

A data type which provides to the client a new type together with appropriate operations, but isolates the client from the internal workings of the type is called an *Abstract Data Type*.

Each member in a Java class can be given either public accessibility, or private accessibility. Publicly accessible members (whether data members or methods) can be used by the client or by other methods within the class itself. Private members are not accessible by the client: they can only be used within the class itself, and constitute part of the internal implementation. Contrast this with Python where all attributes are public — there is no way to restrict clients from accessing the internal workings of an object.

In our class diagrams, we will use a grey background to distinguish the private members from the public ones, thus:

![Class diagram example](image)

---

2By contrast, if we told the client precisely how the information was organized, and allowed the client to access and change the data members directly, we would probably call the type a concrete data type. Many languages have a record structuring mechanism for providing concrete data types.

3There are two other kinds of accessibility, protected and a default, available in Java, but they are not considered here.
Fragment 6.3 Adding access control to the class to produce an ADT

```java
public class Button {
    public final static int UP = 0;
    public final static int DOWN = 1;

    private int state;
    public String caption;
    public int x, y, // position
        w, h; // size
    public boolean visible;
    private Bitmap upMap, downMap;

    private void displayBitmap () {
        if (state == UP)
            ... // display upMap
        else
            ... // display downMap
    } // displayBitmap

    public void press () {
        state = DOWN;
        displayBitmap();
    } // press

    public void release () {
        state = UP;
        displayBitmap();
    } // release

} // class Button
```

Notice here how the client cannot access the bitmap data members or the method displayBitmap() as they are now private. Similarly, a client cannot alter the state of the button except through the press() and release() methods.

Exercise 6.3 Show how the Button might be used in a client program.

6.5 Constructors

A class may have special methods which are called whenever a new object is created (called constructors). These serve the same purpose as the __init__ initialisers in Python classes.
Constructors are especially useful if, when the object is created, we need to initialize the data members or do some other processing. For example, when we create a button using the class above, we would like to ensure that the button starts its life in the “up” state, and that the appropriate bitmap is displayed. The constructors usually have public accessibility. Additionally, their syntax is slightly different from other methods in that:

- the constructor name is the same as the class name, e.g. Button,
- and the constructor does not have a return type (all other methods must have a return type, even if it is just void).
Fragment 6.4 Adding a simple constructor to the class

```java
public class Button {
    public final static int UP = 0;
    public final static int DOWN = 1;

    private int state;
    public String caption;
    public int x, y, // position
            w, h; // size
    public boolean visible;
    private Bitmap upMap, downMap;

    public Button () // Constructor
        { release(); // Start with button up
        } // Constructor

    private void displayBitmap ()
        { if (state == UP)
            ... // display upMap
        else
            ... // display downMap
        } // displayBitmap

    public void press ()
        { state = DOWN;
            displayBitmap();
        } // press

    public void release ()
        { state = UP;
            displayBitmap();
        } // release

    } // class Button
```

Like any other methods, constructors can take arguments. Overloading is permitted too, so a class may have many constructors with different types or numbers of arguments. The usual rules for resolving the overloading are applied.

As an example, we'll modify our button class: when an object is created, the client can (optionally) specify its initial state as UP, or DOWN.

First, the implementation: we assume a default creation state of UP, and add the following constructor to the class, in addition to the previous one:

```java
public Button (String caption, int x, int y, int w, int h, boolean visible) { // Constructor
    this.caption = caption;
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
    this.visible = visible;
    this.state = UP;

    displayBitmap();
}
```
From the client's perspective, the definition of some new buttons could now take an (optional) argument, as in the following example. Note how the parameters are passed to the constructor.

One can go on indefinitely adding new features to the `Button` class. To give another example of a class with overloaded constructors, we extend the `Button` class so that the text caption can be given when the button is created. Firstly, clients can now do all that they could before, plus more...

Here is the program, with all of the constructors shown, but none of the other members:
Fragment 6.8 Demonstration of more than one overloaded constructor

```java
public class Button {
    ...
    public Button () // Constructor
        { release(); // Start with button up
        } // Constructor

    public Button (int initState) // Constructor
        { if (initState == UP)
            { release();
                }
        else
            { press();
                }
        } // Constructor

    public Button (String initCaption) // Constructor
        { caption = initCaption;
            release();
        } // Constructor

    public Button (String initCaption, int initState) // Constructor
        { caption = initCaption;
            if (initState == UP)
                { release();
                    }
        else
            { press();
                }
        } // Constructor
    ...
} // class Button
```
Exercise 6.4 Complete the class definition and draw a class diagram for this class:

```java
public class OddsAndEvens {
...
}
```

```java
public static void main(String[] args) {
    OddsAndEvens x = new OddsAndEvens();
    int num;
    while (...) {
        // Read in next value of num
        x.addIn(num);
    }
    System.out.println("The sum of all odd numbers = "+x.oddSum());
    System.out.println("The sum of all even numbers = "+x.evenSum());
} // main
```
Exercise 6.5 Complete the two class definitions in this program:

```java
public class Point // A point object at position (x, y). {
}

} // class Point

public class Circle
// A circle object with the centre at position (0, 0).
{

}

} // class Circle

public static void main (String[] args)
{
    Circle c = new Circle(5.0), // outer circle radius = 5 units
d = new Circle(2.0); // inner circle radius = 2 units
    double x, y;
    // Read in values for x and y
    Point p = new Point(x, y);
    System.out.print("The point falls ");
    if (d.contains(p))
    {
        System.out.println("within the smaller circle");
    } else
    if (! c.contains(p))
    {
        System.out.println("outside both circles");
    } else
    {
        System.out.println("between the two circles");
    }
} // main

Hint: A point falls within a circle if the distance from the center to the point is less than the radius of the circle. The distance of a point from the origin is given by \( d = \sqrt{x^2 + y^2} \) (from Pythagoras' theorem).
```

Exercise 6.6 How would the circle class in the previous exercise need to be modified if the midpoints of the circles could be at any coordinate position?
Exercise 6.7 Should information such as the position of a point or the radius of a circle be public or private members? Justify your answer.

6.6 Static Class Members

We have seen the `static` keyword used with a number of methods in the example programs (particularly with the `main` method in all our programs so far) and with the constants used in the program above. What does `static` do and why is it needed?

The methods and data members that we have been considering in the examples in this chapter are almost all `instance members`. This means that the variable or method “belongs to” an instance of the class (i.e. a specific object). If we refer to `b1.visible` or to `b1.press()` there is no doubt as to which object’s data or methods we are using — it is clearly `b1`. This is usually exactly what is required for most methods and data members of most classes.

However, in some cases we want data or methods to be shared by all of the objects belonging to the class (or possibly to be used even when there are no such objects in existence). This is what static class members are used for. As a simple example, consider a situation where we want to create objects that have a unique “serial number”. Each object will require a normal instance variable to hold its own serial number, but somewhere in the program we need to keep track of the next available serial number. A static variable is perfect for this as shown in the class below. Note also how the variable `serialNumber` has been “protected” here. It is `private` so it cannot be accessed by any code outside of the `MyObject` class, but its value can be “seen” by using the `getSerialNumber` method. This is a common practice in designing ADTs.

Note that static variables in Java are equivalent to `class attributes` in Python.

Fragment 6.9 A simple example of a static variable

```java
public class MyObject {
    private static int nextNumber = 1;
    private int serialNumber;

    public MyObject () {
        serialNumber = nextNumber;
        nextNumber = nextNumber + 1;
    } // constructor

    public int getSerialNumber () {
        return serialNumber;
    } // getSerialNumber

} // class MyObject
```

The static variable `nextNumber` is shared by all the `MyObject` objects that are created by the program.
CHAPTER 6. CLASSES AND ENCAPSULATION

42

Figure 6.1: Example of Static Data Members.

The first MyObject object will be allocated a serial number of 1, the second will be allocated a serial number of 2, and so on. Static variables are sometimes called class variables because of this behavior: they belong to the class and not to a specific instance of the class (object).

Figure 6.1 shows the state of the program after the following code segment has been executed:

```java
MyObject o = new MyObject(),
o2 = new MyObject(),
x = new MyObject();
```

This also explains why we made the constants UP and DOWN static in the earlier example. As constants they cannot be changed and so there is no need for every object to have its own copy of these values — they can be shared safely by all of the objects created from the Button class.

Just as variables can be static, so can methods. This has a very similar meaning in that such a method “belongs to” the class and not to a specific instance of the class. This is very useful for methods that are intended as general utility functions and do not need to work with a specific object’s data members in order to perform their intended task. Very good examples of such methods are the common mathematical operations of finding square roots, sines, cosines, etc. Here is a simple example of how we might do this:

```java
class MathematicalOps
{
    public static double squareRoot (double x)
    {
        // Calculate and return the square root of x
        ...
    } // squareRoot

    public static double sin (double x)
    {
        // Calculate and return the sine of x
        ...
    } // sin

    ...
} // class MathematicalOps
```

Because these methods are not associated with an instance of this class, but “belong to” the class itself, we can make use of them without creating any objects of this class. This is done by using the class name as if it was an object reference:
Fragment 6.11 Using static methods

```java
class MyMathsProgram {
    public static void main (String args[]) {
        double x, y, angle, hypotenuse;
        // Read values for x, y and angle
        hypotenuse = MathematicalOps.squareRoot(x * x + y * y);
        double s = MathematicalOps.sin(angle);
        ...
    } // main
} // class MyMathsProgram
```

At this point you may be relieved to know that: (1) the Java class library provides a class with all of the common mathematical operations as static methods, and (2) its name is just Math! The important point to note from this program is that we have used the methods of the MathematicalOps class without creating any MathematicalOps objects (nowhere in the program is there any code like MathematicalOps m = new MathematicalOps();).

Another important point about static methods is that they cannot make use of non-static methods or data members. This makes sense if you think about it. The static method does not “belong to” an instance of the class, but the non-static members do. If the static method did refer to a non-static member, which of the many possible instance members is it referring to? In fact, as in the last example program, there may be no instances of the class in existence — so how could we refer to an instance member?!

Exercise 6.8 Look up the Math class in the on-line documentation and see what facilities it offers.

6.7 Final Thoughts

As we leave this chapter we can return to a bit of “magic” that we haven’t explained yet, but are now in a position to understand. That is the modifiers that are required for the definition of the main method of a program. As we have seen in many examples now, the main method must be declared as:

```
public static void main (String[] args)
```

We now know what the public and static modifiers mean. Why does main require them?

The answer to this lies in the way that the JVM executes our Java programs. When you type in a command like:

```
java MyProgram
```

to execute the main method in the class MyProgram, the JVM has to load this class and call the main method. If the JVM, which is not part of your program, is going to do this then main must be public so that it is visible to code outside the class (in this case the code in the JVM).

Similarly, the JVM does not want to create an instance of the MyProgram class. For starters it has no idea what constructor should be called and what parameters the constructor might require. So it does not create an object of the program class! However, this means that the main method must be static so that it can be called without instantiating any object(s) from the program class (in the same way that we made use of the squareRoot method in the last section without creating a MathematicalOps object).
Skills

- You should have a clear idea of the separation between the roles of the clients and the implementor of an ADT.
- As a client...
  - you should be able to define objects of the new ADT, and use the members provided by the implementor
- As an implementor...
  - you should understand the notions of a class and its data members and methods/function members; private and public accessibility; and the role of the class constructor
  - you should understand and be able to use simple UML class diagrams
  - you should be able to define your own simple classes
  - given a fragment of a client program that uses a class, you should be able to determine what the purpose and methods of the class must be, and define a suitable class that satisfies the needs of that client
  - you should be aware that there are many more aspects of the Java class mechanism that have not been covered here
- You should understand how the static keyword modifies the nature of a data member or method.
Chapter 7
Arrays

Objectives
- To familiarize ourselves with the basic concepts for using the built-in array mechanism in Java

7.1 Concepts

Arrays in Java are reasonably straightforward and work a lot like Python lists: they comprise a finite number of elements of the same type, indexed from 0 to \( n - 1 \). Subscripts in Java are always integers. Arrays are declared as follows:

\[
\text{type identifier[]};
\]
or:

\[
\text{type[]} \text{ identifier;}
\]

(There is a subtle difference between these two forms, which we will come back to shortly — for simple cases they can be used interchangeably). For example:

\[
\text{int[]} \text{ myArray;}
\]

This declares the array, but does not actually create it. The reason for this is that arrays in Java are objects. As usual, in order to create an object we need to use the \( \text{new} \) operator:

\[
\text{myArray} = \text{new int[10]};
\]

This creates an array object that can contain ten integers and assigns the reference to the new array object to the variable \( \text{myArray} \). One potential trip-up point is that this defines an array with a maximum subscript value of 9, not 10.
Arrays are indexed using the subscript operator [], e.g. myArray[3].

One of the nice things is that an array can have an initializer expression (just like any other definition). This greatly simplifies setting up arrays of values, as we will see later in this chapter.

Another nice feature of Java arrays is that they are checked for subscript errors (unlike C and C++ where this is the programmer's job).

A method can have formal parameters of array type. A method with an array parameter might be:

```java
1 int largest (int[] list)
2 { ...
3 } // largest
```

(Again, the square brackets may come after the parameter name, or after the parameter type, as shown above).

It may seem that there is a problem here, as there doesn’t seem to be any way for the method to determine the number of elements in the actual parameter array. In fact there is, since all array objects have a data member called length (this is similar to the len function used in Python).

### Fragment 7.1 Find the largest of the elements of a

```java
1 int largest (int[] a)
2 //===================
3 // Find the largest element in a
4 { int biggest = a[0]; // Initial assumption
5     for (int k = 1; k < a.length; k++)
6         if (a[k] > biggest)
7             biggest = a[k];
8     }
9 }
10 return biggest;
11 } // largest
```

In the above code, take special note that the last valid element of the array is at position a.length-1. Note too the coding of the for loop, to ensure that the start and termination conditions do not introduce off-by-one errors.

We also need to consider very carefully how arrays are passed to methods in Java. Remember that the only parameter passing mechanism in Java is call-by-value. However, when it comes to objects this gets a little more complicated. As we noted in chapter one, the variables we use when working with objects are reference variables, which can refer to different objects at different times. When we pass an array as a parameter (or, in general, any object) what we are really passing is a copy of the reference. The method may use this reference to change the contents of the object referred to, but it cannot change the reference itself. So, we still have call-by-value parameter passing, but a method may be able to make changes to an object that will be detected by the rest of the program.

Before we move on, here is the Python function equivalent to the Java method above:
CHAPTER 7. ARRAYS

Fragment 7.2 Python function to find the largest element of a list

```python
def largest (a):
    # Find the largest element in a
    biggest = a[0]  # Initial assumption
    for k in range(1, len(a)):
        if a[k] > biggest:
            biggest = a[k]
    return biggest
```

Earlier we said that we would return to the two different forms of declaration for reference variables and parameters for arrays. As we noted, this makes no difference for simple examples. The following two declarations are completely equivalent:

```python
int myArray[]; int[] myArray;
```

Where this does start to make a difference is when we declare a number of variables in a list:

```python
double a[], b, c; double[] a, b, c;
```

In the first form, only `a` is an array; `b` and `c` are simple integer variables. In the second form, the square brackets apply to all the variables in the list, and so all of `a`, `b` and `c` are arrays. In general, the second form is to be preferred.

7.2 Multi-Dimensional Arrays

Multi-dimensional arrays are simply arrays of arrays (this is similar to the use of nested lists in Python to represent matrices). The statement below defines a three-dimensional array of integers, with rank $3 \times 5 \times 7$.

```python
int x[][][] = new int[3][5][7];
```

Any of the terms `x`, `x[i]`, `x[i][j]`, `x[i][j][k]`, may then reasonably appear in an expression. A question that arises in this context is: “what is the meaning of the length field?” The answer is easily found by considering the multi-dimensional array as an array of arrays. For the example above we have: `x.length` is 3 (i.e. the number of “two-dimensional” arrays); `x[k].length` is 5, for $0 \leq k \leq 2$; and `x[k][j].length` is 7, for $0 \leq k \leq 2$ and $0 \leq j \leq 4$.

7.3 An Example

The following program illustrates these points for a cricket example. It uses an array that is initialized to contain the scores that some batsman has made in his last twelve innings.

The program prints the array, sorts it, and then prints the sorted list.
Fragment 7.3 Printing an array

```java
1 // George Wells - 14 January 2011
2 public class Cricket
3 { static void print (int[] a)
4     //=========================  
5     // Print the elements of 'a'
6     { for (int i=0; i < a.length; i++)
7         { System.out.print(a[i] + " ");
8         }
9     System.out.println();
10 } // print
```

Fragment 7.4 Sorting the array

```java
12 static void bubblesort (int[] a)
13     //==============================
14     // No-frills sort of the elements of 'a'
15     { for (int k=0; k < a.length-1; k++)
16         { for (int j=0; j < a.length-1-k; j++)
17             { if (a[j] > a[j+1])  // Swap them
18                 { int tmp=a[j]; a[j]=a[j+1]; a[j+1]=tmp;
19                 }
20             }
21         }
22 } // bubblesort
```

Fragment 7.5 Completing the program

```java
24 public static void main (String[] args)
25     //================================================
26     { int scores[] = {31,17,12,9,19,2,31,90,28,5,14,11};
27     print(scores);
28     bubblesort(scores);
29     print(scores);
30 } // main
31 } // class Cricket
```

- The sort changes the elements in the array: the reason that these changes are visible in `main` is that, while the parameter `scores` is passed by value, the contents of the array object can be changed by the `bubblesort` method, as discussed in section 7.1.
- The initializer syntax (line 26) for the array is very convenient, as is the ability to find the length of an array automatically.

For comparison, the same program in Python is as follows (note that `print` is a keyword in Python, so cannot be used as a function name):
Fragment 7.6 The Cricket program in Python

# George Wells - 14 January 2011

def prnt (a):
    # Print the elements of 'a'
    for i in range(len(a)):
        print a[i],
    print

def bubblesort (a):
    # No-frills sort of the elements of 'a'
    for k in range(len(a)-1):
        for j in range(len(a)-1-k):
            if (a[j] > a[j+1]): # Swap them
                tmp=a[j]
                a[j]=a[j+1]
                a[j+1]=tmp;

def main ()�
    # =========
    scores = [31,17,12,9,19,2,31,90,28,5,14,11]
    prnt(scores)
    bubblesort(scores)
    prnt(scores)

    if __name__ == '__main__':
        main()

Exercise 7.1 Write a function which searches the first $n$ unsorted cricket scores to determine whether a batsman has made exactly key runs in any innings.
Exercise 7.2 Write a function which searches the first $n$ sorted cricket scores to determine whether a batsman has made exactly $key$ runs in any innings. Use a binary search algorithm.

One important point to note is that array sizes in Java are fixed — this means that an array cannot change its size once its storage has been allocated using `new`. However, unlike many other languages, the actual number of elements in the array does not have to be known to the compiler at compile time: we can delay determining the size of the array until run time.

Fragment 7.7 A legal fragment of program

```java
1  int n;
2  // Read in a value for n
3  int a[] = new int[n]; // Legal in Java
4  ...
```

7.4 Final Thoughts

We can now fill in the last missing fact about the way in which the `main` method for all Java programs is declared: the “args” parameter. As you should be able to guess by now, this is an array of strings. But what is being passed to the `main` method, as it is being called by the JVM? The answer to this is that the interpreter picks up any command-line arguments that are typed after the class name and passes these to the program for it to use if necessary. If a Java program is executed from the command-line as follows:

```
java MyProgram George 27 CS2
```

then the `main` method will be passed an array containing three strings: "George", "27" and "CS2". These are referred to as the “command-line arguments”, hence the convention of calling the parameter `args`. In general, this feature can be very useful for giving the program information such as filenames for it to work with, and for specifying options that control the program’s behaviour.
Skills

- You should be able to define arrays of any type, and be comfortable with the subscript notation.
- You should understand the consequences of out-of-range subscripts.
- You should understand the way in which multi-dimensional arrays work in Java.
- You should understand the mechanism for passing arrays as parameters.
- You should understand the meaning of the parameter notation:
  
  ```
  void foo(<type>[] a)
  ```
- You should be able to use the initializer syntax for arrays.
- You should be able to use command-line parameters in Java.
Chapter 8

String Handling

Objectives

- To introduce the string handling mechanisms in Java
- To reinforce some of the object concepts via examples
- To become aware of some of the facilities in the String class

8.1 Concepts

As we have already commented, strings in Java are objects. As with any object, we work with strings by means of reference variables. However, strings are a little unusual in that the compiler “knows” more about strings than other types of objects and allows us to do some things with strings that are not possible with other objects. One simple example of this is shown in the following code fragment:

Fragment 8.1 Setting up a string

```java
1 String str = "Hello";
2 System.out.println(str); // Output the string
```

In line 1 we are creating a new object, but we haven’t used the new operator. The reason for this is that the compiler automatically creates the necessary String objects to represent the string literals used in a program (like "Hello" here). What we have after line 1 is the usual situation with references to objects, which we can picture as shown in Figure 8.1.

If we now execute the statement:

```java
3 str = "Wells";
```

we get the picture as shown in Figure 8.2.
The question is now: "what happens to the original string (containing "Hello")?". The answer is that it is now out of the reach of our program, as there is no longer any reference variable that refers to it. As such it is referred to as "garbage": memory that our program was using to store an object but which is no longer required. Many languages that support dynamic memory allocation like this require that the programmer keeps track of the memory being used. Thus it is the programmer’s responsibility to ensure that the program deallocates the memory used by an object once it is no longer required. This is a notorious source of programming errors in many programming languages, not least C++. Fortunately for us, Java has a very powerful and useful feature called automatic garbage collection. The Java run-time system keeps track of the references to the objects that the program is using and will automatically reclaim any memory that is no longer being used by the program (such as that being used by the "Hello" object in our example).

Another important fact to bear in mind with Java strings is that they are immutable. This means that the contents of the object cannot be changed after the object has been created. To put it another way: it is impossible to edit or modify a string object in Java. Of course, it is possible to make a String reference variable refer to another object with different contents, as our example has shown, but the original object can never be changed.

Since they are objects, strings have various methods (in fact, quite a lot of methods) that we can use. These are listed in the on-line documentation for the String class. A simple one is the length method, which returns the number of characters in a string. For example, str.length() will return 5 for our example above (note that this is a method and should not be confused with the length data member of an array). There is also a method to extract the characters from a string one by one:

```
for (int k = 0; k < str.length(); k++)
    System.out.print(str.charAt(k));
System.out.println();
// The above is equivalent to: System.out.println(str);
```

Figure 8.1: A reference to a String object.

Figure 8.2: A reference to a String object after assignment.
Exercise 8.1  Children often encrypt messages by substituting letters by others which fall some constant distance further in the alphabet. If they get to the end of the alphabet, they start again from ‘a’. For example, substituting letters 3 places further translates “hello, yacht” into the secret message “khoor, bdfkw”.

1. Write a function which takes a string, and puts its encrypted version into into a second char array parameter.

2. What particular choice of constant would allow us to use the same function for either encrypting or decrypting a message?

8.2 The String Class

Here we mention a few of the more useful methods in the String class.

Fragment 8.3 Some methods in the String class

```java
1  int    length ()
2     // Returns the length of this string.
3
4  String concat (String str)
5     // Concatenates the specified string to the end of this string.
6
7  int compareTo (String anotherString)
8     // Compares two strings lexicographically. See below for meaning.
9
10 int compareToIgnoreCase (String str)
11     // Compares two strings lexicographically, ignoring case.
12     // Returns 0 if this == str
13     // < 0 if this < str
14     // > 0 if this > str
15
16 int indexOf (int ch)
17     // Searches for ch in this string, returns the index of its position, or -1 if it does not occur.
18
19 int indexOf (String str)
20     // Returns the index within this string of the first occurrence
21     // of the specified substring, or -1 if it does not occur.
```

There is an important point that needs some care with string processing (or any object handling for that matter): be sure you understand the difference between working with or comparing the references, and handling the strings themselves. Study the following example carefully, and make sure you understand exactly what is happening...

```java
String s = "a"; String p = "hello"; String q = "bye";
```
p = q;
p = "Seven";
p = s;
s = concat(s,"goodbye");
if (p == s) . . .
if (p.compareTo(s) == 0) . . .

Skills

- You understand how strings are handled in Java
- You know how individual characters can be accessed
- You know how Java handles automatic garbage collection for objects
- You understand the difference between working with a reference variable and working with an object
- You are aware of some of the main functions in the String class
Chapter 9

Exception Handling

Objectives

- To introduce the idea of exceptions in Java for handling error conditions

9.1 Concepts

Java has a very powerful mechanism for handling errors, called exceptions. Exceptions are used for almost all error-handling in Java, from simple programming errors (like array subscripting errors) through user errors (such as invalid file names) to fatal system errors (such as the JVM running out of memory).

As a simple example, consider the following program extract, which contains an obvious error:

```java
Fragment 9.1 A program with an error
46   int[] a = new int[3];
47   System.out.println(a[5]);
```

If this program is executed, the result is as shown below:

```
java.lang.ArrayIndexOutOfBoundsException
at Test.main(Test.java:47)
Exception in thread "main"
```

This shows the full name of the exception (i.e. `java.lang.ArrayIndexOutOfBoundsException`), which helps explain why the error arose. It also gives the name of the class (`Test`) and the method (`main`), and even the line number (47) where the exception was generated.
9.1.1 Types of Exception

As with many other aspects of the Java language, exceptions are handled by means of objects. These exception objects are “thrown” when an error is detected. In general, a program must then decide what it is going to do with the exception: either catch it and deal with it, or throw it along for some other part of the program to deal with.

The exception classes form a hierarchy as illustrated in Figure 9.1. This hierarchy classifies the different types of exception into three categories:

Errors These are generally very serious conditions from which a program cannot realistically hope to recover (such as the JVM running out of memory). As a consequence of this they may be (and usually are) ignored by programs.

Runtime Exceptions This class of exceptions is used for common programming errors which can be prevented by careful programming (such as division by zero, or subscripting errors). The compiler is lenient about these, and programs do not have to check for them (a reasonable assumption if the program is well written!). The ArrayIndexOutOfBoundsException thrown by the program at the start of this chapter is an example of such an exception.

Other Exceptions This category serves for all the remaining exceptions. Common examples are I/O errors, network errors, security errors, etc. In general these types of errors may be situations from which a program can recover and so the Java compiler forces the programmer to handle the exception.
CHAPTER 9. EXCEPTION HANDLING

To summarise: a programmer may choose to ignore errors and run-time exceptions, but is forced to deal with all other types of exception. This can be done in two ways, as discussed in the next two sections.

9.2 Catching Exceptions

The first, and often most appropriate, way of dealing with an exception is to catch it. For example, if we chose to catch the subscripting error shown in the example program at the start of this chapter, we could do so as follows:

**Fragment 9.2 Catching an exception**

```
1    int[] a = new int[3];
2    try
3        System.out.println(a[5]);
4    }
5    catch (ArrayIndexOutOfBoundsException e)
6        System.err.println("ERROR: "+ e);
7 }   
```

Notes:

- Since the `ArrayIndexOutOfBoundsException` is an example of a run-time exception, we do not have to do anything about it. This example simply serves to illustrate the exception catching mechanisms in Java for a very easy case.

- The syntax of the exception catching mechanism is that the code which may cause the exception is enclosed in a `try` block. This is followed by one or more `catch` clauses, specifying what is to be done when a particular exception, or category of exceptions, occurs.

- The syntax of the `catch` clause is that it specifies the exception very much like a parameter to a method. Here we have used `e` as the name of the exception which we may catch. This is the exception object itself and it may contain useful information about the exact nature of the error.

- Having caught the exception it is completely up to the programmer to deal with it. It is not uncommon to see programmers ignoring exceptions (this is not recommended!) as follows:

```
    try
    {
    ... 
    }
    catch (SomeException e)
    {}
```

In the example shown in Fragment 9.2, we have chosen simply to print an error message and carry on. The output from this program will now be:

```
ERROR: java.lang.ArrayIndexOutOfBoundsException
```

Note how we can simply print the exception object (`e` in the example) and this results in the name of the exception being displayed.

Quite often the best course of action is to print out an informative error message and then terminate the program. This can be done using a method called `System.exit`, as shown in the following example:
try
{
...
}
catch (SomeException e)
{
 System.err.println("ERROR: " + e);
 System.exit(1);
}

- We have printed out the error messages in the examples above using System.err. This is usually equivalent to using System.out, but is preferred for error messages (some operating systems may allow error messages and normal output to be sent to different devices or files).

- Note that the Java try/catch statement is very similar to the equivalent statement in Python.

As stated above, we can catch more than just one type of exception from a single try block. The following example shows a fairly common situation when dealing with I/O exceptions.

Fragment 9.3 Catching more than one exception

```java
try
{
...
}
catch (FileNotFoundException e)
{
 System.err.println("File "mydata.txt" not found");
}
catch (IOException e)
{
 System.err.println("I/O ERROR: " + e);
}
```

When catching multiple exceptions like this the order of the catch blocks is important. The more specific exceptions must come before the more general ones (as in this example: FileNotFoundException is a specific type of IOException). The compiler will complain if you get this wrong.

### 9.3 Throwing Exceptions

The second approach to handling exceptions is to pass them along, rather than catching them. This works by passing the exception back along the chain of methods that have been called in the program up to the point where the exception occurred. This is done automatically for errors and run-time exceptions, but must be handled explicitly by the programmer for the other exception types. If a method may generate an exception, but does not wish to catch the exception, then it must declare that it throws the exception.

In Java this is known as the “handle or declare” rule: a method must either handle the exception (using try/catch), or declare that it throws the exception.

A method declares that it throws an exception using the throws keyword in the method header:

Fragment 9.4 Throwing an exception

```java
void readData () throws IOException
{
 // Do all sorts of error-prone I/O, without try/catch:
...
} // readData
```
In this case, rather than having to handle the exception in the `readData` method the exception gets thrown back to the point in the program where the `readData` method was called from. At this point the exception must be caught or that method must declare that it throws the exception back to where it was called from, and so on. Eventually the exception must end up being caught by a `try/catch`.

Skills

- You should understand the principles behind the error-handling mechanisms in Java
- You should know the three main categories of errors, and the differences between them
- You should know how to catch an exception in a Java program and deal with it appropriately
- You should know how to declare that a method throws an exception and understand how this is handled

---

1 This is not strictly true. The `main` method may also declare that it throws exceptions, in which case the program effectively throws the exception out to the JVM, and will then be terminated with a standard error message.
Chapter 10

Input and Output Streams

Objectives

- To familiarize ourselves with the basic ideas of reading and writing data in Java.

10.1 Introduction

In Chapter 1 we saw how to produce output from our programs using System.out, and we mentioned that we can read data from the System.in stream. We need to explore these ideas in a little more detail, and, in particular, to find out exactly how input works (unfortunately this is a little more complicated than output in Java, which is why we have put it off to now!).

Before we look at the stream mechanisms in more detail, it should be noted that many of the methods used for input and output may throw exceptions (due to premature end-of-file, invalid data, invalid file names, etc.). For simplicity the exception-handling code (i.e. try/catch or throws) is not shown in the example programs in this chapter. It should also be noted that the I/O classes are all in a package called java.io, and this needs to be included using an import statement at the top of the program as shown in the following program extract:

Fragment 10.1 Importing the I/O classes from the java.io package

```java
import java.io.*;
public class MyProgram {
    // ...
}
```

A package in Java simply serves to group a number of related classes together (it is very similar to the idea of a module in Python). The java.lang package is automatically included in all Java programs by the compiler, which means that the classes it contains (i.e. very widely used classes like String) are automatically available for use. All other packages must be included explicitly, as shown in the example above.
The notation `import java.io.*` means import all of the classes from the `java.io` package. The asterisk can be replaced by a class name if only one class is required to be imported (e.g. `import java.util.Scanner` will import only the `Scanner` class from the `java.util` package).

10.2 Reading from a Stream

The basic concept of stream input/output (I/O) is that there is a stream of information flowing into a program and another stream of information flowing out. These streams have a source or a destination, and can optionally flow through various processing stages, called filters. For example, a typical program might have the I/O configuration shown in Figure 10.1.

In Java, the source or destination nodes in such a stream, together with any filtering stages are constructed from stream objects. The source or destination stream objects are referred to as node streams and the filtering stages as filter streams.

To slightly complicate matters, the initial release of the Java development system did not always handle textual I/O correctly for foreign alphabets. To correct this, version 1.1 introduced the parallel concept of readers and writers for text I/O. Again, these are simply Java objects. A reader can very easily be connected to an input stream, and a writer to an output stream, providing the necessary bridging between the two approaches. We will restrict ourselves in this chapter to considering some of the simpler uses of streams and text readers/writers. More details can be found in most Java reference books.

10.2.1 Reading Text

In order to read text into a Java program we need to connect a reader to the `System.in` input stream. There are a number of readers available to us (see the on-line documentation for full details), but the most useful of these is the `BufferedReader` class that allows us to read text a line at a time. However, a `BufferedReader` cannot be connected directly to an input stream and so we need to make use of an `InputStreamReader` (a filter stream) as a bridge. The code to do all this is as follows:
Figure 10.2: A BufferedReader connected to System.in.

**Fragment 10.2 Connecting a BufferedReader to System.in**

```java
1 InputStreamReader isr = new InputStreamReader(System.in);
2 BufferedReader in = new BufferedReader(isr);
```

Or, equivalently:

```java
1 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

We can picture this situation as shown in Figure 10.2.

Having set up the streams and readers we can read the text typed by the user on the keyboard as follows:

**Fragment 10.3 Reading from a BufferedReader**

```java
1 String s = in.readLine();
2 while (s != null)
3 { // Process the line
4     ...
5     s = in.readLine(); // Read the next line
6 }
```

Note how we check for `null` to tell when we have reached the end of the input. If the standard input stream is redirected to take its input from a disk file or a remote source, the system will automatically know when the stream ends. If, however, the input stream is attached to the user’s keyboard, the user will have to type some special character to say “Hey, I’m finished with input.” Under Unix, this is usually the Control-D character, while under Windows, it is usually Control-Z.

Here is a very simple fragment of program which counts how many lines there are in an input stream. Note the middle-exit loop:

**Fragment 10.4 Counting the lines in an input stream**

```java
1 int n = 0;
2 while (true)
3 { String s = in.readLine(); // try to read a line
4     if (s == null) break; // exit loop if it fails
5     n++;
6     } // process the data
7 System.out.println("There were " + n + " lines of input");
```

Take careful note that if there are only ten lines in our input stream, line 3 is executed 11 times — we must try to read before we can tell that we’ve run out of input data.

There are some Java shortcuts that allow us to write this in a more compact way — this is a fairly
common Java idiom that needs to be understood and recognized. Firstly, we can put the assignment into a condition (this arises from the way in which assignments in Java are expressions, as discussed in Section 2.6):

### Fragment 10.5 A small variation

```java
int n = 0;
String s;
while (true)
    { if ((s = in.readLine()) == null) // read and exit if it fails
        break;
    n++; // process the data
    }
System.out.println("There were " + n + " lines of input");
```

Note that the parentheses in line 4 of the code above are essential. If they are omitted (i.e. if we had written `if (s = in.readLine() == null)`) the compiler would interpret this as assigning the result of the expression `in.readLine() == null` (a boolean) to the variable `s` (a `String` reference variable), giving an error.

We can still take this program a step further and combine lines 3-5 above into one, using the controlling expression for the `while` loop. Here is the final idiomatic form (note how the logic of the condition needs to be changed around):

### Fragment 10.6 The idiomatic style

```java
int n = 0;
String s;
while ((s = in.readLine()) != null)
    { n++; // process the data
    }
System.out.println("There were " + n + " lines of input");
```

This is compact and very sensible: you can read line 3 as “While you are still able to successfully input from the reader to `s`, execute the body of the loop sol".

### 10.3 Reading Numeric Information

This is all very well, but how can we read numbers, rather than strings, into our programs? If the numeric information is in a textual format (which is usually the case when we read from the keyboard or from a text file) then we need to read the data as a `String` still, but then convert it to a numeric form that our program can use. This is done using the facilities provided by a number of classes called the **wrapper classes**. The wrapper classes provide us with a way of handling the primitive data types (e.g. `int`, `double`, `char`, etc.) as objects. Thus, each of the eight primitive data types has an equivalent wrapper class. These usually have the same name as the primitive data type, except that the wrapper class name is capitalised. For example, the wrapper class for `double` values is `Double`. There are two exceptions: the `int` wrapper class is `Integer`, and the the `char` wrapper class is `Character`.

In addition to allowing us to work with primitive values as objects, the wrapper classes in Java also provide a number of utility methods, including the ability to convert strings into the primitive types. For example, to convert a string to an integer you use the `Integer.parseInt()` method, which takes the string as a parameter.
As an example, here is a method which reads a list of numbers (one per line) into an array, and returns a count of how many numbers have been inserted into the array.

**Fragment 10.7 Reading into an array of integers**

```java
1 int readIntList (BufferedReader in, int[] list)
2 { int num, count = 0;
3     String s;
4     while ((s = in.readLine()) != null)
5         { num = Integer.parseInt(s);
6             list[count++] = num;
7         }
8     return count;
9 } // readIntList
```

Note that after executing the `while` loop, `count` reflects the number of items in `list`, and the last element is stored at `list[count-1]`. In this case, the client, or caller, is responsible for allocating the array, and making sure it is big enough, or else a subscript bounds exception will be thrown.

**Exercise 10.1** A better version of this method might test the value of `count` against `list.length` and return as soon as `count` became equal to that maximum. Make this change to the method above.

### 10.4 Reading from a File

How do we create our own node streams? Perhaps we do not want to use `System.in`, or need to be able to read data from a file and interact with the user in the same program. Here is a simple program that shows how to define new file streams. Having defined a file stream, we must open it (by attaching it to a file), and can then use it just like we have used the `System.in` stream. Note that this is a complete, working program with all the exception-handling and other necessary features.
CHAPTER 10. INPUT AND OUTPUT STREAMS

Fragment 10.8 Sum all numbers in the file called “mydata.txt”

```java
import java.io.*;

public class SumData {
    public static void main(String[] args) {
        try {
            BufferedReader in = new BufferedReader(new FileReader("mydata.txt"));
            String s;
            int total = 0;
            while ((s = in.readLine()) != null) {
                total += Integer.parseInt(s);
            }
            in.close();
            System.out.println("Total = " + total);
        } catch (IOException e) {
            System.err.println("I/O Exception caught: " + e);
        }
    }
} // class SumData
```

10.5 Easier Input

In Java 1.5 a new I/O feature was introduced which greatly simplified the common case of reading data from the standard input channel. This is the `Scanner` class. The following complete example program shows how one might read in a set of numbers and calculate the total. Note that the `Scanner` class is part of the `java.util` package rather than `java.io` package.

Fragment 10.9 Reading and summing integers

```java
import java.util.*; // Scanner is not in java.io

public class Summer {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num, total = 0;
        while (in.hasNextInt()) {
            num = in.nextInt();
            total += num;
        }
        System.out.println("The total is " + total);
    }
} // class Summer
```

Note that, in this case, the numbers may be typed in with several on each line (separated by spaces) and/or on separate lines. For example:

```
C:\> java Summer
1
2
```
In addition to being able to read integers, the Scanner class provides methods that allow all the primitive types and words (or “tokens”) to be read. See the documentation for the class for more details.

10.6 Closing Remarks

In addition to the basic I/O mechanisms that we have explored in this chapter there is a lot more available in Java. For example, we have not touched on the use of data files (binary files), or random access facilities. Java even provides a very powerful I/O feature called serialization that allows objects to be read from or written to streams very easily. There are also very powerful networking features that integrate very neatly into the basic stream mechanisms making the transmission of data (or even objects) across a network very simple.

Skills

- You should be familiar with the concepts of streams and readers and writers, and why we have these two slightly different I/O mechanisms in Java
- You should be able to read textual data as strings, and detect end-of-stream properly
- You should be able to read data into an array
- You should be able to convert textual data into numeric formats
- You should be able to create different input readers which are attached to disk files
- You should be able to use the Scanner class
- You should be aware that there is a lot more to I/O in Java than has been covered in this chapter
Chapter 11

More Classes and Objects

Objectives

- To introduce the concepts of inheritance and method overriding
- To define polymorphism and see examples of its use

11.1 Inheritance

One of the most important and powerful features of object-oriented programming is the ability for classes to inherit properties and methods from other classes. As a simple example of this, consider the kinds of accounts that a typical bank might offer. All accounts have some basic things in common: there will be a current balance, an interest rate, and ways of depositing and withdrawing money. A normal savings account might only have these features, but other types of account are likely to need to have further information. For example, a current (cheque) account usually has an overdraft limit, and an associated interest rate for when the account is overdrawn; a fixed deposit will have a termination date when the money will be paid out; etc.

In an object-oriented programming language we can use a hierarchy of classes to represent these kinds of relationships. We usually start by considering the most fundamental type of object — a generic bank account in our example. Such a class might look something like this:

```
Account

<table>
<thead>
<tr>
<th>balance, accountNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>interestRate</td>
</tr>
<tr>
<td>deposit, withdraw</td>
</tr>
</tbody>
</table>
```

The (very simplistic) Java code for this might be:
A generic bank account class

```java
public class Account {
    public double balance;
    public String accountNumber;
    public double interestRate;

    public void withdraw (double amount)
    { balance = balance - amount;
    } // withdraw

    public void deposit (double amount)
    { balance = balance + amount;
    } // deposit
} // class Account
```

Note that, for simplicity, we have made all of the members of this class public, but in practice one would want to protect at least the balance field from being altered, except through the withdraw and deposit methods. The methods are also very simple, particularly the withdraw method, which should really check to make sure that there is enough money in the account. We will come back to this point a little later.

If we now consider a cheque account, this will need exactly the same information as the generic Account class, plus additional data members for the overdraft limit and overdraft interest rate:

A ChequeAccount class

```java
public class ChequeAccount {
    public double balance;
    public String accountNumber;
    public double interestRate;
    public double overdraftLimit, overdraftIntRate;

    public void withdraw (double amount)
    { balance = balance - amount;
    } // withdraw

    public void deposit (double amount)
    { balance = balance + amount;
    } // deposit
} // class ChequeAccount
```
If you look at this carefully, you will see that we have ended up with a lot of duplication here: the two classes have several data members in common, and the methods are identical in both cases. If we go on to consider fixed deposits, credit card accounts, special savings accounts, etc., etc. we will find that this duplication just gets worse. So, how can an object-oriented approach help us? The answer lies in the fact that an object-oriented programming language allows one class to inherit the properties of another.

In this example, we can inherit all the properties of the `Account` class in the `ChequeAccount` class. All we then need to put in the `ChequeAccount` class are the properties that are unique to it. So, the `Account` class does not change at all, and the `ChequeAccount` class becomes:

```
ChequeAccount
  overdraftLimit
  overdraftIntRate
```

Notice how we have only specified the unique properties of the `ChequeAccount` class. The complete Java code for the new `ChequeAccount` class is as follows:

```
Fragment 11.3 The ChequeAccount class using inheritance

public class ChequeAccount extends Account
{
  public double overdraftLimit, overdraftIntRate;
} // class ChequeAccount
```

The phrase `extends Account` in the class header is what introduces the inheritance here. This means that the `ChequeAccount` automatically inherits all of the data members and methods of the `Account` class. So a `ChequeAccount` object has a balance, an account number and interest rate, and methods for withdrawals and deposits, but we haven’t had to write them in the `ChequeAccount` class. Neat!

We often use UML diagrams to show the relationships between classes that are related by inheritance. For our simple example, the inheritance hierarchy would be as shown in Figure 11.1.

There is a little terminology that we need to be familiar with when considering such inheritance hierarchies. The class which is inherited from is called the superclass (`Account` is the superclass in our example). The class which is inheriting is called the subclass (`ChequeAccount` is the subclass in our example). We sometimes refer to inheritance as subclassing (e.g., “`ChequeAccount` subclasses `Account`”). We also sometimes call the superclass the parent class and the subclass the child class. In Java, a class can only have one superclass — this is referred to as single inheritance.

In general, an inheritance hierarchy can be far more complicated than shown by this example. A class may inherit from a class that in turn inherits from another class, and so on. A class may also have many subclasses. For a bank system we might end up with an inheritance hierarchy something like that in Figure 11.2. (We also saw a more complex inheritance hierarchy when we looked at the different categories of exceptions in section 9.1.1, although we didn’t talk about it as an inheritance hierarchy then).

---

1. The type of inheritance that we are talking about here is more like genetic inheritance of parental characteristics, than legal inheritance arising from a will.
2. In Python we would declare this class as: `class ChequeAccount(Account)`
3. Some other object-oriented programming languages allow a class to inherit from more than one superclass (i.e. multiple inheritance), but this tends to get very complicated.
CHAPTER 11. MORE CLASSES AND OBJECTS

Figure 11.1: The Class Hierarchy for Account and ChequeAccount.

Exercise 11.1 Write a few of the classes suggested by the inheritance hierarchy in Figure 11.2. You will need to use some common sense and imagination in designing the properties of the classes.

11.1.1 Overriding Methods

We made the point a little earlier that the withdraw method was very simple and had no checks to make sure that there was enough money in the account. We can fix this easily enough:

Fragment 11.4 A better withdraw method

```java
public class Account {
    ...
    public void withdraw (double amount)
    {
        if (amount <= balance)
            balance = balance - amount;
        else
            System.err.println("Insufficient funds for withdrawal");
    } // withdraw
    ...
} // class Account
```

At first it might seem that this has introduced a problem, since the ChequeAccount class has different criteria for checking whether a withdrawal is allowed. When the ChequeAccount class inherits this new withdraw method it will not permit the bank’s customers to withdraw up to their overdraft limit, but only up to their balance. Fortunately, there is a way to get around this. A class that is inheriting from another class is permitted to override the inherited methods. This is not the same as overloading (see section 4.2). When a class overrides an inherited method it provides an identical method that effectively replaces the inherited method.
Figure 11.2: A Possible Class Hierarchy for a Banking System.
For our problem we need to override the `withdraw` method in the `ChequeAccount` class because it needs to work slightly differently. The `deposit` method can still be inherited in its original form.

### Fragment 11.5 Overriding the `withdraw` method in `ChequeAccount`

```java
public class ChequeAccount extends Account {
    public double overdraftLimit, overdraftIntRate;

    public void withdraw(double amount) // Override withdraw in Account
    { // Allow for overdraft
        if (amount <= balance + overdraftLimit) // Allow for overdraft
            balance = balance - amount;
        else
            System.err.println("Insufficient funds for withdrawal");
    } // withdraw
} // class ChequeAccount
```

#### 11.1.2 The `Object` Class

Without making a fuss about it, we have been using inheritance in every Java class that we have ever seen or written. The reason for this is that the Java compiler automatically forces classes to use inheritance, even if they do not state that they extend some other class. When we write:

```java
public class SomeClass {
    ...
} // class SomeClass
```

in our programs, the compiler effectively changes it automatically to:

```java
public class SomeClass extends Object {
    ...
} // class SomeClass
```

The `Object` class is thus the ultimate superclass (or ancestor) of all Java classes. This means that all Java classes inherit all the methods and properties of the `Object` class. These are usually not very interesting or important to us (at least not at this stage). One potentially useful method that is defined in the `Object` class is `toString`. This method converts an object into a `String` representation. The default for this is not very pretty, but it does mean that all Java objects can be converted to a string, and the compiler can rely on this. For example:

```java
ChequeAccount cq = new ChequeAccount();
System.out.println(cq.toString());
// Or, equivalently:
System.out.println(cq); // The compiler calls toString() automatically
```

This program will print out something like:

```
ChequeAccount@3fbdb0
```

which is not very helpful! However we are free to override the inherited `toString` method and provide our own more useful version. For example:
### 11.2 Polymorphism

Until now whenever we have been working with objects we have written code much like the following to declare and initialise reference variables:

```java
MyClass x = new MyClass();
```

When we have an inheritance hierarchy like the one in the previous section we can also work with related objects in the following ways:

1. `ChequeAccount c = new ChequeAccount(); // No surprise`
2. `Account a = new ChequeAccount();        // Surprise!`
3. `Account fd = new FixedDepositAcct();`
4. `CreditCardAcct cc = new GoldCardAcct();`

What are we doing here, and why does it work (it does work — trust me!)? The answer is that we are making use of another very powerful feature of object-oriented programming called *polymorphism*. That might seem like a big, complicated word, but it is quite simple in fact: *poly* comes from a Greek word meaning “many”, and *morph* refers to the “shape” or the “form” of something. So we’re talking about objects that can take on many forms, which is exactly what we have in the code above. In line 1 we’re dealing with a `ChequeAccount` object and treating it as if it had the form of a `ChequeAccount` object (again, there should be no great surprise in that). In line 2 we’re dealing with a `ChequeAccount` object again, but now we’re treating it as if it had the form of an `Account` object. We’re allowed to do this because a `ChequeAccount` object has got all the properties of an `Account` object — it inherited them from the `Account` class.

Similarly, in line 3 we can treat a `FixedDepositAcct` object as if it had the form of an `Account` object, and in line 4 we can treat a `GoldCardAcct` object as if it had the form of a `CreditCardAcct` object. In each case you can see from the inheritance hierarchy in Figure 11.2 that this is not an unreasonable thing to do as the objects we are dealing with have all inherited the necessary properties to take on the form of one of their superclasses.

Some important points:

- An object can be treated as having the form of any of its superclasses, no matter how many intervening levels there may be in the inheritance hierarchy.
- We can only treat a subclass as having the form of a superclass. The following example is *not* correct:
ChequeAccount x = new Account(); // WRONG!
as the Account object cannot have the form of a ChequeAccount object. This should not be a
surprise, as it doesn’t have any of the necessary overdraft information.

- When we refer to an object, the type of the reference variable determines what we can do with
  the object that it refers to. So, although in the example above the variable fd actually refers to a
  FixedDepositAcct object, the fact that the type of the variable is Account restricts us to using
  the methods and accessing the data members defined in the Account class. Effectively we have lost
  access to the more specialised properties of the FixedDepositAcct class by using polymorphism.

- When we call an overridden method in a polymorphic situation, it is the version of method in the
  actual object that is called. For example, if we called a.withdraw(250.00) we would actually be
  calling the version of the withdraw method defined in the ChequeAccount, which is the type of the
  object that a refers to, even though the type of the reference variable a is Account.

11.2.1 Using Polymorphism in Practice

While the examples in the previous section serve to illustrate how polymorphism works, they are not
very good examples! In practice we hardly ever use polymorphism in the way shown above.
Polymorphism is extremely useful in several situations. The first of these is in parameter passing. For
example, imagine that the bank that we have been considering wanted to write the balances of all
the accounts to a log file at the end of the day. Without polymorphism they would have to write
different methods to deal with cheque accounts, credit card accounts, savings accounts, etc., etc. Using
polymorphism they can simply write one method:

---

Fragment 11.7 Polymorphic parameter passing

```java
public void writeBalance (Account acc)
{ logFile.println(acc.accountNumber + ": R" + acc.balance);
} // writeBalance
```

---

This method can take an Account object as the actual parameter, but, thanks to polymorphism, it can
also take ChequeAccount objects, or CreditCardAcct objects, or objects belonging to any other subclass
of the Account class. In the writeBalance method we only rely on the account number and balance
information, which all the subclasses have got through inheritance, so polymorphism is perfect for this
purpose.

The other situation where polymorphism is very useful is when we need to deal with lists of related
objects. Again, considering our banking example, the bank is very unlikely to use a collection of variables
like c, a, fd and cc, as we had in the earlier example. They are far more likely to work with an array
of accounts. Polymorphism allows us to create an array that can store all the different types of account
that we need to deal with:
A polymorphic list

```java
Account list[] = new Account[120000]; // Space for 120 000 accounts
list[0] = new ChequeAccount();
list[1] = new FixedDepositAcct();
list[2] = new GoldCardAcct();
list[3] = new ChequeAccount();
...
// Now find total balance
double total = 0.0;
for (int k = 0; k < list.length; k++)
    total = total + list[k].balance;
```

Skills

- You should be able to use inheritance for simple classes
- You should be able to override inherited methods in subclasses
- You should understand the significance of the `Object` class in Java
- You should be able to make use of polymorphism, particularly for parameter passing and generic lists
- You should be familiar with the object-oriented terminology used in this chapter
Chapter 12

An Example Program

Objectives

- This chapter reinforces the material in the previous chapters by working through a more substantial example of a full program that implements and uses an ADT.

12.1 The Frequency Table

Students write tests and exams, and one important measure of class performance is a frequency table showing the spread of marks. In order to keep our problem simple, we’ll begin by assuming that all marks are integer percentages.

Your course supervisor wants to know how many students got marks lower than or equal to 9%, 10–19%, 20–29%, … or 90% and above. Associated with each of these ranges is a frequency bin, (a simple counter for values within that range), and the collection of all the bins is called the frequency table.

We’ll start with a class diagram that shows only the public member functions, and a simple client program which will provide the essential requirements for the class:
CHAPTER 12. AN EXAMPLE PROGRAM

Fragment 12.1 A client which uses our frequency table...

```java
import java.io.*;

public class MarkAnalysis {
    public static void main(String[] args) throws IOException {
        FreqTable cs2 = new FreqTable();
        BufferedReader in = new BufferedReader(new FileReader("marks.txt"));
        String s;
        int total = 0;
        while ((s = in.readLine()) != null) {
            cs2.addEntry(Integer.parseInt(s));
        }
        for (int k = 0; k < 10; k++) {
            System.out.println("The number of hits in bin " + k + " is " +
                    cs2.hits(k));
        }
    } // main
} // class MarkAnalysis
```

In passing, notice how this program uses a rather lazy alternative approach to handling the potential I/O exceptions.

The `FreqTable` class itself is very simple:

Fragment 12.2 A frequency table ADT

```java
public class FreqTable {
    private int[] bin = new int[10];

    public void addEntry(int entry) {
        int binNumber = entry / 10;
        if (binNumber < 0)
            binNumber = 0;
        if (binNumber >= 10)
            binNumber = 9;
        bin[binNumber]++;
    } // addEntry

    public int hits(int binNumber) {
        return bin[binNumber];
    } // hits
} // class FreqTable
```

12.2 More Flexibility

Let us extend this class now. In another situation we might like the bins to each be 5 units wide, perhaps over a different range of values. We might even want to capture daily maximum temperatures where we’d like a range such as 6–40 degrees in steps of 2 degrees.

At the time the table is created, we must now specify the lower bound of the first bin, the width of each bin, and the number of bins. Our frequency table of marks would thus be declared as:
FreqTable cs2 = new FreqTable(0, 10, 10);
while the daily temperature definition would be

FreqTable dayTemps = new FreqTable(6, 2, 17);

When we come to implement this, there is a fairly tricky bit: previously, mapping a mark like 47 into bin number 4 was quite easy: we simply divided by 10. Now the mapping is a little more complicated.

Exercise 12.1 Given daily temperature measurements of 9, 17, 24, 32 and 15, calculate which bin number each measurement should fall into.

Exercise 12.2 In general, given that the lower bound of the frequency table is $lb$, and the width of each bin is $n$, give a formula which calculates which bin a measurement $x$ must fall into. Check that your formula agrees with the hand calculations done in the previous exercise.

We begin with a class diagram showing some of the private members, and a little more detail about the constructor arguments that we will need.
Fragment 12.3 The improved FreqTable class

```java
public class FreqTable {
    private int[] bin;
    private int lb; // Lower bound of first bin
    private int n; // number of bins
    private int binSize; // Number of values in each bin

    public FreqTable (int lowerBound, int width, int numBins) {
        bin = new int[numBins];
        lb = lowerBound;
        n = numBins;
        binSize = width;
    }

    public void addEntry (int entry) {
        int binNumber = (entry - lb) / binSize;
        if (binNumber < 0) binNumber = 0;
        if (binNumber >= n) binNumber = n-1;
        bin[binNumber]++;
    }

    public int hits (int binNumber) {
        return bin[binNumber];
    }
}
```

Satisfy yourself as to how this works.

For comparison, the equivalent Python program is shown in Fragment 12.4 below. Notice the differences, particularly the explicit use of `self` in the Python methods, and the explicit initialisation required for the list `bin`. 
Fragment 12.4 A Python frequency table used to analyse class marks

```python
class FreqTable:
    def __init__(self, lowerBound, width, numBins):
        # Initialiser
        self.bin = [0] * numBins
        self.lb = lowerBound  # Lower bound of first bin
        self.n = numBins     # number of bins
        self.binSize = width  # Number of values in each bin
    
    def addEntry(self, entry):
        binNumber = (entry - self.lb) / self.binSize
        if binNumber < 0:
            binNumber = 0
        if binNumber >= self.n:
            binNumber = self.n-1
        self.bin[binNumber] += 1

    def hits(self, binNumber):
        return self.bin[binNumber]

def main():
    cs2 = FreqTable(0, 10, 10)
    inFile = open("marks.txt", "r")
    total = 0
    s = inFile.readline()
    while s != ":
        cs2.addEntry(int(s))
        s = inFile.readline()
    for k in range(10):
        print "The number of hits in bin", k, "is", cs2.hits(k)

if __name__ == '__main__':
    main()
```

Skills

- You should have more confidence about building a non-trivial class
Bibliography


[13] Sun Microsystems. The official Java homepage. URL: [http://java.sun.com/](http://java.sun.com/). This is Sun’s main Java page and has links to all sorts of useful on-line resources.

Index

++, 13
--, 13

Abstract Data Type, 26, 31
accessibility, 31
aggregate data type, 26
arrays, 43
    passed as parameters, 44
automatic garbage collection, 50

boolean, 11
break, 16, 19
byte, 11
bytecode, 3

cast, 12
char, 11
class, 26
class, 26
class variables, 40
coding conventions, 4
command-line arguments, 47
comments, 3
compiling programs, 3
constructors, 32
continue, 19
data members, 26
do, 17
double, 9, 11

encapsulation, 28
escape sequences, 2, 9
exceptions, 53
    handle or declare rule, 56
executing programs, 3
extends, 66

factorial, 4, 6
filter streams, 59
final, 25
float, 11
for, 17, 18
garbage collection, 50

idioms, 19, 20
if, 15
import, 58
inheritance, 64
initializers
    arrays, 44
instance, 7
instance members, 39
instantiation, 28
int, 9, 11
interpreter, 3

Java Virtual Machine, 3
JVM, 3

literals
    strings, 2, 9
long, 11

methods, 21, 29
    nesting, 5
    overloading, 22
    overriding, 67

node streams, 59

Object, 69
operator
    ++, 13
    --, 13
    assignment, 13
overloading, 22
overriding, 67

package, 58
polymorphism, 70
    arrays, 71
    parameter passing, 71
private, 31
protected, 31
public, 27, 31
public class, 7

readers, 59
INDEX

reference, 7
reference variable, 7

scope, 5
serialization, 63
short, 11
short circuit evaluation, 11
shorthand assignment operator, 14
single inheritance, 66
static, 39
stream, 3
stream input, 58
String, 9, 49
string functions
  charAt, 50
  compareTo, 51
  concat, 51
  equals, 11
  indexOf, 51
  length, 50, 51
subclass, 66
subclassing, 66
superclass, 66
switch, 16
System.out, 24

type promotion, 12
type-cast, 12

variable
  case-sensitivity, 10
  initialization, 11
  local, 5
  scope, 5, 11
void method, 22

while, 17
wrapper classes, 61
writers, 59