THE LITTLE BOOK OF JAVA

by Huw Collingbourne
(Bitwise Courses)

http://bitwisecourses.com/

written by
Huw Collingbourne

First edition: February 2015

Chapter 1 – Getting Started

Before you can start programming Java, you need to understand what Java is and what tools you need in order to write Java programs.

WHAT IS JAVA?

Java is a cross-platform Object Oriented language. You write Java programs in high-level human-readable programming code in files ending with the extension `.java`. The Java compiler then translates this text into a compressed non-human-readable code called bytecode which can be run by a program called the Java Virtual Machine (JVM). Any operating system that can run a Java Virtual Machine (and these days most can) is able to run your Java bytecode. This is why Java programs are highly portable across different hardware platforms and operating systems.

Java was developed in the mid '90s by programmers at Sun MicroSystems. In 2010 Sun was acquired by Oracle Corporation and Oracle is now the focus of development and documentation of the Java language. However, Java – the language and its principal tools – are free to use without having to pay Oracle royalties.

WHAT WILL YOU LEARN?

This course assumes no prior knowledge of programming in general or the Java language in particular. It teaches you everything you need to get started with writing, compiling and running Java programs on a PC, Mac or (in principle) any other system capable of running Java. It will teach you the fundamentals of programming, including Object Oriented programming. It will explain how to create objects, calls methods, test conditions, handle data and deal with files on disk. If you already have some programming experience, feel free to skip over the early sections of the course. Most lessons are accompanied by ready-to-run code samples provided in the downloadable source-code archive. There are also some more substantial example programs that implement a simple exploring-style text adventure game to illustrate many of the features and techniques described throughout this course in a more complete application.

Editors and IDEs

You will need an editor or IDE (Integrated Development Environment) to run Java programs. While several alternatives exist (see Appendix), this course uses the NetBeans IDE (refer to the next section on Installation to find out where to download NetBeans).

Installation

You need to install the Java JDK for your operating system. You may obtain the JDK from the Oracle web site here:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Optionally, from this page, you can download a version of the Java JDK that includes the NetBeans development environment. If you need both the JDK and a Java programming environment, this would be the simplest option. If you already have the JDK installed and you want to install NetBeans, you can obtain the software here: https://netbeans.org/downloads/index.html

On Windows

Windows users may now want to add your Java installation to your computer's search path so that the Java compiler and tools may be run from any directory or folder. This is how to do this:

Click *Start*, then *Control Panel*, then *System*.
Click *Advanced*, then *Environment Variables*.
Add the location of the *\bin* folder of the JDK installation for the `PATH` variable in *System Variables*. The following is a typical value for the `PATH` variable (But be sure to use the actual path to your JDK):

```
C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Java\jdk1.8.0_5\bin
```

For more complete instruction on installation for Windows, Mac or Linux, be sure to read Oracle's detailed setup guides and tutorials here:
http://docs.oracle.com/javase/8/

Get the source code

The source code of the projects described in this course is provided in the form of a downloadable Zip archive (available on the course Home Page). You will need to unzip the archive using an UnZip tool before using the code.

Naming Conventions

When you give names to variables, classes and other programming elements it is normally neater and clearer if you adopt a consistent 'naming convention'. So, for example, you may decide to name private variables such as `description`, all in lowercase like this:

```
private String description;
```

You might name methods in mixed case with the first letter in lowercase and any other words capitalized like this:

```
public String getDescription()

public int movePlayerTo(Direction dir)
```

I won't be insisting a strict set of naming conventions in this course. In real-world programming, the naming (and code formatting) conventions are often defined by the team-leader of a specific project. However, if you want to adopt a fairly typical set of Java naming conventions, you can refer to (a rather out-of-date) guide on the Oracle site here:

http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

In this course I will mention important conventions as I encounter them – for example, the capitalization of class names and constants.

MAKING SENSE OF THE TEXT

In **The Little Book Of Java**, any Java source code is written like this:

```
public void setThings(ThingList things) {
    this.things = things;
}
```

Any output that you may expect to see on screen when a program is run is shown like this:

```
The result of that calculation is 24!
```

When there is a sample program to accompany the code, the program name is shown in a little box like this:

Adventure4

When an important name or concept is introduced, it may be highlighted in the left margin like this:

FUNCTIONS

Explanatory notes (which generally provide some hints or give a more in-depth explanation of some point mentioned in the text) are shown in a shaded box like this:

This is an explanatory note. You can skip it if you like – but if you do so, you may miss something of interest…!

About the Author

Huw Collingbourne has been a programmer for more than 30 years. He is a top-selling online programming instructor with successful courses on C, C#, Object Pascal, Ruby, JavaScript and other topics. For a full list of available courses be sure to visit the Bitwise Courses web site: http://bitwisecourses.com/

He is author of *The Book Of Ruby* from No Starch Press and he holds the position of Director of Technology at SapphireSteel Software, makers of the *Ruby In Steel*, *Sapphire* and *Amethyst* programming environments for Microsoft Visual Studio (http://www.sapphiresteel.com/).

He is a well-known technology writer in the UK and has written numerous opinion and programming columns (including tutorials on Java, C#, C++, Delphi, Smalltalk, ActionScript and Ruby) for a number of computer magazines, such as *Computer Shopper*, *Flash & Flex Developer's Magazine*, *PC Pro*, and *PC Plus*. He is author of the free eBook *The Little Book of Ruby* and is the editor of the online computing magazine Bitwise (http://www.bitwisemag.com/).

In the 1980s he was a pop music journalist and interviewed most of the New Romantic stars, such as Duran Duran, Spandau Ballet, Adam Ant, Boy George, and Depeche Mode. He is now writing a series of New Romantic murder mystery novels.

At various times Huw has been a magazine publisher, editor, and TV broadcaster. He has an MA in English from the University of Cambridge and holds a 2nd dan black belt in aikido, a martial art which he teaches in North Devon, UK (http://www.hartlandaikido.co.uk/). The aikido comes in useful when trying to keep his Pyrenean Mountain Dogs under some semblance of control.

Chapter 2 – First Steps in Java

In this chapter we shall discover the basic features of the Java language and we shall write a few short Java programs.

Hello World

By tradition, the first program many people like to create when learning a new programming language is one that simply displays "Hello world". If you are using NetBeans, select *File | New Project*. In the *New Project* dialog, make sure the *Java* category is selected in the left-hand pane and the *Java Application Project* type is selected in the right-hand pane. Click *Next*. Give the project a name such as *HelloWorld* and make sure that the *Create Main Class* option is ticked. You might also want to browse to a project location to select a specific directory for this project. Then click *Finish*.

NetBeans will create a source code file called *HelloWorld.java*. This includes a `main` function which will be run when the program itself is run. Edit this `main` function by adding the following code between the curly brackets:

```
System.out.println("Hello world");
```

The code in the file should now look like this:

HelloWorld.java

```
package helloworld;

public class HelloWorld {

    public static void main(String[] args) {
        // Display Hello world to output
        System.out.println("Hello world");
    }

}
```

In fact, there may also be some 'comments' – that is, text placed between `/*` and `*/` or placed after `//` characters. Comments are 'inline documentation' and they are ignored by the Java compiler. As long as the actual Java code is the same as is shown above, you are all ready to go.

COMMENTS

It is a good idea to add comments to your programs to describe what each section is supposed to do. Java lets you insert multi-line comments between pairs of `/*` and `*/` delimiters, like this:

```
/* This program displays any
   arguments that were passed to it */
```

In addition to these multi-line comments, you may also add 'line comments' that begin with two slash characters `//` and extend to the end of the current line. Line comments may either comment out an entire line or any part of a line which may include code before the `//` characters. These are examples of line comments:

```
// This is a full-line comment
if (args.length == 0){ // this is a part-line comment
```

The easiest way to run this program within NetBeans is by clicking the green arrow '*Run Project*' button or by selecting the *Run Project* item from the *Run* menu. All being well, you will now see the output from the program – that is, *Hello world* – displayed in the NetBeans *Output* window.

If you want to run a Java program from the system prompt, you will need to open a command window or Terminal in the *classes* directory. This is the directory that contains the *helloworld* subdirectory (you can see in my code that *helloworld* is the name of the Java 'package' – I'll have more to say about packages later. To run the Java virtual machine – the tool that runs your Java programs, you need to enter `java` followed by the name of the package and class file (minus its extension). Here my class file name is *HelloWorld.class* (that's the compiled file that has been placed into the *helloworld* subdirectory), so I have to enter this command:

```
java helloworld/HelloWorld
```

And this is the output I see:

```
Hello world
```

Troubleshooting

In some circumstances, instead of the expected output, Java may show an error message such as:

```
Error: Could not find or load main class
```

There are a number of possible reasons for this. For example, this message might appear if the *HelloWorld.class* file does not exist (maybe you haven't compiled your program yet); or you might have entered the wrong name or used the wrong-mix of upper and lowercase letters (Java is 'case sensitive' so if your class is called *HelloWorld* with a capital *H* and *W* but you enter *java helloworld/Helloworld* (with a lowercase *w*) the class will not be found. Or maybe you are in the wrong directory. Make sure you are in the *\classes* directory. Or possibly you didn't compile the main class in the specified package. You can check this by highlighting the *HelloWorld* project in NetBeans and selecting *File | Project Properties*. Then select the *Run* node in the dialog. Check that the *Main Class* is shown as *helloworld.HelloWorld* where *helloworld* is the package name and *HelloWorld* is the class name.

Even if you have problems running from the command prompt, don't worry. In this course we will not normally run programs in this way. If you are using an IDE (and for the purposes of this course, I strongly recommend that you use NetBeans), you will be able to compile and run programs within the IDE itself rather than from the system prompt.

Running Programs At The Commandline

If you plan to compile and run programs from the system prompt, you may want to read Oracle's documentation:

http://docs.oracle.com/javase/tutorial/getStarted/cupojava/index.html

Oracle also has a problem-solving guide to help you identify and fix common errors:

http://docs.oracle.com/javase/tutorial/information/run-examples.html

INTRODUCTION TO JAVA CODE

Let's now turn to the code of this simple program and try to identify its essential parts.

The program begins with this:

```
/*
 * Bitwise Courses - sample Java code
 * http://www.bitwisecourses.com
 */
```

As explained earlier, any text placed between the characters `/*` and `*/` is treated as a comment and it is ignored by the Java compiler. A comment can be used to document your code or to provide copyright or licensing details.

The first line of real Java code occurs here:

```
package helloworld;
```

Here `package` is a keyword – that is, a predefined word that has a special meaning in Java code, and `helloworld` is a name or 'identifier' which I have chosen for the current package. The statement is terminated by a semicolon. Semicolons are widely used in Java to mark the end of a statement.

KEYWORDS

Java defines a number of 'reserved' words or keywords that have special meanings in the language. For example, when we start testing values, we will use the `if` and `else` keywords. The following keywords are found in our simple 'Hello world' program: `package`, `public`, `class`, `static`, `void`. For now you need not be concerned about what these all mean. The most important Java keywords will be explained later in this course.

A Java 'package' defines a named region (technically called a 'namespace') in which related Java code can be grouped together. The standard libraries of Java code (the "Application Programming Interface" or API) contain numerous named packages. For example, the `java.io` package contains code for dealing with IO (Input/Output) operations, while the `java.math` package contains code for doing arithmetic. In terms of organising your code, packages fulfil a similar role to the named folders on your computer's hard disk. Your disk folders may group together related data files. Your Java packages group together related code. In fact, Java packages are often (but not invariably) placed into disk folders that have the same names as the packages. This has the advantage of making the organisation of packages clear.

Now we come to this:

```
public class HelloWorld {
```

CLASS

If you have programmed an object oriented language before you will be familiar with the concept of a class. If you haven't, it is enough for now to know that a class defines a 'block' of code that may contain some data and some 'subroutines' or 'functions'. This is a class called *HelloWorld*. The contents of the class are contained between a pair of curly brackets. In fact this class only contains a single function called `main()`:

```
public class HelloWorld {

    public static void main(String[] args) {
        // Display Hello world to output
        System.out.println("Hello world");
    }

}
```

In Java, the name `main` is treated specially. The `main()` function of the main class in a Java program is run first when the program starts. Incidentally, while this very simple program contains just one class, real-world Java programs will usually contain many classes. In NetBeans, I can specify the main class in the *Run* page of the *Project | Properties* dialog.

METHODS ARE OBJECT-ORIENTED FUNCTIONS

The names of functions (or 'methods' as they are often called in Object Oriented languages) must be followed by a pair of parentheses like this:

```
public void myfunction()
```

If you want to be able to pass some kind of data to a function, you can add one or more 'arguments' between the parentheses. The `main()` function defines a list or 'array' of String arguments called `args`:

```
main(String[] args)
```

Just as the class is delimited by a pair of curly brackets, so too is the function. Note also that the function contains a line-comment that documents what the function does:

```
// Display Hello world to output
```

Finally, we come to this single line of code:

```
System.out.println("Hello world");
```

Here `System` is the name of a class supplied by the Java API and `out` is the name of an item or 'field' that is found inside the `System` class. In fact, `out` is the name of a `PrintStream` object and it contains a function or 'method' called `println()` which displays (that is, it 'prints') a string.

A *string* is a sequence of characters delimited by double-quotes.

In order to call the `println()` method I have to specify the full 'path' in the form of the `System` class name, the `out` field name and the `println()` function name, all joined together by dots: `System.out.println`. Then I have to pass the string, "Hello world" to `println()` by placing it between a pair of parentheses. The end result is that, when I run this program, "Hello world" is displayed.

PASSING COMMANDLINE ARGUMENTS

Sometimes it is useful to pass an argument to your program when it is loaded and run. For example, if you have written a Java game or a database program you might want to tell it to restore a saved version of your game or reload a specific data file. To do this you can pass one or more strings to your program as 'arguments'. You can see how to do this in the *HelloArgs.java* sample program.

HelloArgs.java

```
public class HelloArgs {

  public static void main(String[] args) {
    System.out.println("Hello.");
    if (args.length == 0){
      System.out.println("You didn't enter any commandline arguments.");
    }else for (String arg : args) {
      System.out.println(arg);
    }
  }

}
```

If you run this program from the command prompt or Terminal you would pass arguments to it simply by entering bits of text separated by spaces, like this:

```
java helloargs/HelloArgs one two three
```

And the program would then display this:

```
Hello.
one
two
three
```

This is how it works. The commandline arguments are passed in the form of an 'array' – a sequential list – of strings: "one", "two" and "three". This array is assigned to the `args` argument in `main()`.The code tests the length of `args`. If it is 0 there are no arguments and the message "You didn't enter any commandline arguments." is displayed. If there are some arguments, however, this bit of code runs:

```
for (String arg : args) {
    System.out.println(arg);
}
```

Here the `for` keyword starts running a loop that processes each item in the list of `args` one at a time and prints each string on a separate line – hence the output shown earlier.

If you are running the program direct from the NetBeans IDE, you can tell NetBeans which arguments to pass to your program. Load the *Project | Properties* dialog and enter the arguments into the *Arguments* field on the *Run* page. The output will now be shown in the Output window.

COMPILING JAVA PROGRAMS – FROM SOURCE CODE TO BYTECODE

Java programs are written in a human-readable form – the Java source code that you write in an editor. But before the program can be run, that source code has to be translated into a different format called 'bytecode'. The translation from source code to bytecode is done by the Java compiler.

JAVA VIRTUAL MACHINE AND JAVA RUNTIME ENVIRONMENT

In order to run a Java program, the bytecode has to be loaded into the Java Virtual Machine or JVM. You can think of the JVM as a special environment for running Java programs and it is, in fact, the core element of the Java Runtime Environment (JRE). In addition to the JVM, the JRE also includes various essential Java class libraries.

The JVM is the secret to Java's ability to run programs on many different operating systems. Some programming languages such as C and Pascal compile source code into a format called machine code which is run by your computer's hardware. Machine code is fast and efficient but it isn't portable – programs compiled to machine code can generally only be run on the hardware and operating system on which they were compiled. A program compiled to machine code on a Windows PC won't run on a Mac, for example. But the Java bytecode format is run inside the JVM. That means that if you have a JVM installed on a Windows PC and you also have a JVM installed on a Mac or on Linux, you can compile your programs to bytecode on one of those platforms (Windows, Mac OS X or Linux) and then run those compiled programs using the JVM on another operating system or hardware platform.

BACK TO THE SYSTEM PROMPT…

Java source code files end with the extension `.java`. Java bytecode files end with the extension `.class`. To translate source code to bytecode you must pass the `.java` file to the Java compiler. The Java compiler has the name `javac`. This is the command I would enter in order to compile a source code file called *HelloWorld.java* at the system prompt:

```
javac HelloWorld.java
```

NOTE: There are many options available for use with the `javac` compiler. For example, it is possible to compile multiple source code files and to place the compiled `.class` files into a specific subdirectory. To see the available options, just enter the command `javac` at the system prompt. More detailed documentation can be found here:
http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html

The Java compiler would create a bytecode file called *HelloWorld.class*. In order to run that file I have to pass the file name (without the `.class` extension) to the Java interpreter. This is the program that runs the bytecode inside the Java Virtual Machine. The Java interpreter is named `java`. So, in order to run the compiled file *HelloWorld.class* this is the command I would need to enter:

```
java HelloWorld
```

More information on the `java` command can be found here:
http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html

While it is useful to understand the basics of compiling and running Java programs from the system prompt, in this course we will generally use the NetBeans IDE which allows us to compile and run program simply by making selections from menus.

CREATING A 'VISUAL' APPLICATION

Most of the applications that we run on our computers have some sort of 'visual' user interface. Everything from a calculator to a word processor pops up in its own self-contained window and contains a number of 'components' such as buttons and text fields. It turns out that Java (especially when used from NetBeans) makes it very easy to create visual applications and, in fact, most of the programs in this course will be 'visual' rather than having to be run from the commandline. Here I want to guide you step-by-step through the creation of a visual version of our 'Hello world' program – one that pops up in a window and displays 'Hello world' in a text field when a button is clicked. Although this is a very simple program, the essential tasks will be similar no matter what type of visual Java program, you create.

- In Netbeans select *File |New Project |Java | Java Application*.
- Click *Next* (optionally browse to a project location) and give the project a name.
- *Un*check *Create Main Class* (make sure the item is *not* ticked). Click *Finish*.
- Right-click the main node of your project in the Project panel.
- From the popup menu, select *New | JFrame Form*. Click *Finish*.
- From the Palette drag and drop two components: a Button and a Text Field.
- Use the mouse to and move and size the components on the form.
- Select the Text Field. In the Projects panel, find the *text* property. Delete the text.
- Select the Button. In the projects panel, find the *text* property. Change text to *Click Me!*
- Right-click button, select *Change Variable Name*. Enter a descriptive name such as: *myButton*
- Right-click text field, select *Change Variable Name*. Enter a descriptive name such as: *outputField*
- Double-click the button. This will create this method in the code editor:

```
private void myButtonActionPerformed(java.awt.event.ActionEvent evt)
{
        // TODO add your handling code here:
}
```

- Edit this method to the following:

```
private void myButtonActionPerformed(java.awt.event.ActionEvent evt)
{
    outputField.setText("Hello world");
}
```

- Select *File* | *Save All*.
- Select *Run* | *Run Main Project*.
- A dialog prompts you to select the main class. Accept the default *NewJFrame* object and click *OK*.
- And that's it!

DISPLAYING TEXT

There are several functions that can be used to display information when your Java programs run. Here I'll look at some important ways of displaying text in both visual (form-based) and non-visual (commandline) applications.

I - PRINTING TO THE COMMANDLINE

When you want to 'print' some text from the system prompt you can use `print()`, `println()` or `printf()`. All these functions are supplied (as I explained earlier) as methods of `System.out`. To display strings on new lines you can use `println()` which automatically appends a newline character at the end of each string. So if you were to write this code:

```
System.out.println("Hello world");
System.out.println("Hello world");
System.out.println("Hello world");
```

This is what would be displayed:

```
Hello world
Hello world
Hello world
```

The `print()` function is similar to `println()` but it does not append a newline character. If you were to write this code:

```
System.out.print("Hello world");
System.out.print("Hello world");
System.out.print("Hello world");
```

This is what would be displayed:

```
Hello worldHello worldHello world
```

You can embed newline (`"\n"`) and tab (`"\t"`) characters into strings displayed by `print()` and `println()`. For example, if you wrote this code:

PrintText.java

```
System.out.print("\tHello ");
System.out.print("world\n");
System.out.println("Hello\tworld");
```

This would be displayed:

```
        Hello world
Hello   world
```

The `printf()` function allows you to embed 'format specifiers' into a string. A format specifier begins with a `%` and is followed by a letter: `%s` specifies a string; `%d` specifies a decimal or integer. When format specifiers occur in the string, the string must be followed a comma-delimited list of values. These values will replace the specifiers in the string. The programmer must take care that the values in the list exactly match the types and the number of the format specifiers in the string. Here is an example:

```
System.out.printf("There are %d bottles standing on the %s.\n", 20,"wall");
```

When run, the code produces the following output:

```
There are 20 bottles standing on the wall.
```

There are many other format specifiers available for use with strings displayed by `printf()`. For guidance, refer to Oracle's documentation of the `Formatter` class:

http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

2 – DISPLAYING TEXT IN VISUAL CONTROLS

When you want to display strings in a visual Java program, you need to use the methods provided by the visual components. The `getText()` and `setText()` methods are used with many types of control such as text fields and buttons. The `getText()` method returns or 'gets' the text which is currently being displayed by the control. The `setText()` method assigns or 'sets' some new text to be displayed by the control.

GetTextSetText

```
String someText;
someText = textEnteredField.getText();
upperTextField.setText(someText.toUpperCase());
myButton.setText("Thank you!");
```

To get or set the title of a `JFrame` (the main window of your application), use `getTitle()` and `setTitle()`. In my sample program I call these methods using the keyword `this` to indicate that I want to call them on the current object which happens to be the `JFrame`:

```
this.setTitle(this.getTitle()+'!');
```

Chapter 3 – The Fundamentals of Java Programming

In this chapter we'll look at how different types of data are categorised and how we can work with data items by assigning them to named variables.

VARIABLES AND TYPES

When your programs do calculations or display some text, they use data. The data items each have a data *type*. For example, to do calculations you may use integer numbers such as 10 or floating point number such as 10.5. In a program you can assign values to named variables. Each variable must be declared with the appropriate data type. This is how to declare a floating-point variable named **myDouble** with the *double* data-type:

```
double myDouble;
```

You can now assign a floating-point value to that variable:

```
myDouble = 100.75;
```

Alternatively, you can assign a value at the same time you declare the variable:

```
double myDouble = 100.75;
```

CONSTANTS

The value of a variable can be changed. So, for example, if you were to write a tax calculator that allowed the user to enter a subtotal and then calculated both the tax and also the grand total (that is, the subtotal plus the tax), you might have three floating point (`double`) variables named `subtotal`, `tax` and `grandtotal`. The values of these variables would be determined when the program is run and they could be assigned different values each time a new calculation is done. You may also change the values in your code, like this:

```
subtotal = 10.0;
subtotal = 20.2;
subtotal = 105.6;
```

But sometimes you may want to make sure that a value *cannot* be changed. For example, if the tax rate is 20% you might declare a variable like this:

```
double TAXRATE = 0.2;
```

But now someone might accidentally set a different tax rate (this is not a problem in a very small program – but this sort of error can easily crop up in a real-world program that may contains tens or hundreds of thousands of lines of Java code). For example, it would be permissible to assign a new value of 30% like this:

```
double TAXRATE = 0.3;
```

If you want to make sure that this cannot be done, you need to make `TAXRATE` a constant rather than a variable. A constant is an identifier whose value can only be assigned once. In Java, I can create a constant by placing the `final` keyword before the declaration like this:

```
final double TAXRATE = 0.2;
```

Now it would be an error (my program would not compile) if I tried to assign a new value to `TAXRATE` in my code. This, then is the finished version of my calculation method in my simple tax calculator:

TaxCalc

```
private void calcBtnActionPerformed(java.awt.event.ActionEvent evt) {
        final double TAXRATE = 0.2;
        double subtotal;
        double tax;
        double grandtotal;

        subtotal = Double.parseDouble(subtotalTF.getText());
        tax = subtotal * TAXRATE;
        grandtotal = subtotal + tax;

        taxTF.setText(Double.toString(tax));
        grandtotalTF.setText(Double.toString(grandtotal));
   }
```

MORE ON CONSTANTS

Some programming languages use the `const` keyword to define constants. While the `const` keyword is defined in Java it is not used (though it may be used in a future version). Constants that are declared outside methods – that is, in the body of a Java class – are usually defined using both the `static` and `final` keywords, in addition to the keyword `public` or `private`. So, for example, the value of `PI` (in Java's `java.lang.Math` package) is declared like this:

```
public static final double PI = 3.141592653589793;
```

The keywords `public`, `private` and `static` will be explained later in this course.

PACKAGES

We've already seen packages a few times in previous projects. For example, in the Hello World program from Chapter 2 we created a code file called *HelloWorld.java* and saved it into a sub directory named *helloworld*. At the top of the code I put this:

```
package helloworld;
```

This states that the code in *HelloWorld.java* exists in the package named *helloworld*. So what exactly is a 'package'? Put simply, a package is a named group of code files. In the *HelloWorld* program there was just one code file in the *helloworld* package. Often, however, a package will contain many code files. Typically the code files in a package will be related in some way – a package called *calc* might contain a number of source code files that do calculations; a package called *fileops* might contain numerous files that do some sort of file handling operations, and so on. In other words, packages defined named libraries of reusable code.

Adventure1

For a better example of how packages can be used to create logical groups of code files, load up the *Adventure1* project. This is my first attempt at writing a text-based exploring game in Java. While most of the sample projects in this course are very short, this project will gradually grow in size as we go through the course. It provides an example of a more complex Java programming project which will use a broad variety of coding techniques. Don't worry too much about the details of the code at the moment – I'll explain how it all works later on. For now it is sufficient to know that the project contains four packages. The default (unnamed) package contains the visual design code (the main form of my project). Then there are three named packages: *game*, *gameobjects* and *globals*. Notice that the *gameobjects* package contains three code files: *Actor.java*, *Room.java* and *Thing.java*. These files are all related – they implement three 'types' of object in the game.

IMPORTING

The code inside a named package cannot be used by code in other packages unless it is explicitly imported. If you look in the *Game.java* file you'll see that I have imported the `Actor` and `Room` classes like this:

```
import gameobjects.Actor;
import gameobjects.Room;
```

Since a package may contain many different classes, it is often useful to import only the specific classes that your code requires. Alternatively, you may use an asterisk to import all the classes in a single operation, like this:

```
import gameobjects.*;
```

The standard Java class library contains many classes and you will often need to import these by adding the appropriate import statement (this statement may sometimes be added 'automatically' by an IDE such as NetBeans). For example, in the *Game.java* file you will see that I have imported everything from Java's `util` package with this statement:

```
import java.util.*;
```

TYPE CONVERSION

There are many times when you will want to convert a piece of data to some other data-type. In fact, we have already seen a number of examples of converting between a numeric type and a string. In the tax calculator sample program, for instance, the string that the user entered into the subtotal text field, `subtotalTF`, was accessed by calling the `getText()` method. This string was then converted to a floating-point `double` value by passing it as an argument to `Double.parseDouble()` and this returned a `double` value which was assigned to the variable `subtotal`, like this:

```
subtotal = Double.parseDouble(subtotalTF.getText());
```

Later, when I wanted to display the value of the `double` number `grandtotal` in the `grandtotalTF` text field, I passed that floating-point value as an argument to `Double.toString()` which returned the string representation of `grandtotal` and then I called `grandtotalTF.setText()` to display that string value like this:

```
grandtotalTF.setText(Double.toString(grandtotal));
```

To understand why these conversions are needed you have to be clear on the difference between numeric and string values. A floating point number such as 10.5 can be used in calculations. But a string such as "10.5" cannot. That's because the `double` value 10.5 is treated as a number by the computer. But the string "10.5" is treated as a list of four characters '1','0','.' and '5'. It may *look* like a number to you and me. But to the computer it looks like four characters strung together. In order to display a number in a text field that number must be converted to its string representation. In order to do a calculation using a string value such as "10.5" that string must be converted to its numeric equivalent.

The `toString()` method that converts a number to a string, is provided by the `Double` class for a `double` value and by the `Integer` class for an `int` value. These classes also provide the `parseDouble()` and `parseInt()` methods to convert strings to `double`s and `int`s. You can find more examples of converting `double` and `int` numbers to and from strings in the *DataTypes* project.

```
// convert double to string
double myDouble;
myDouble = 100.75;
myTextArea.setText(Double.toString(myDouble));


// convert int to string
int myInt;
int myOtherInt = 10;
myInt = 20;
myTextArea.setText(Integer.toString(myInt + myOtherInt));

String d;
String i;
double total;
d = "15.6";
i = "100";
total = Double.parseDouble(d) + Integer.parseInt(i);
```

Be careful when converting data. It is your responsibility to ensure that it is possible for the data to be converted successfully. If, for example, you try to convert a string such as "Hello World" to a `double`, the operation will fail and, unless you take specific measures to recover from the error (we'll look at ways of doing that in chapter 9) your program may crash!

But what exactly is the difference between `double` with a lowercase initial `d` and `Double` with an uppercase initial `D`, or between `int` and `Integer`? Let's consider that next.

Primitives and Wrappers

Java is a high-level object-oriented language. That means it uses data-types and programming techniques which are convenient to the programmer but which mean nothing to the computer hardware. For an extreme example of a high-level data-type, think of the buttons we've added to our visual designs. `JButton` is a recognised data-type. It implements the appearance and the behaviour of a button object. Other high-level data-types include String and Integer. A string object, just like a button object, wraps up some data and some behaviour. The behaviour takes the form of methods such as the `setText()` method of a button or the `toLowerCase()` method of a String:

ObjectsAndMethods

```
String myString;
myButton.setText("Clicked!");
myString = "Hello World";
myTextArea.setText(myString.toLowerCase());
```

But these complex data-types – objects with built-in methods – are created by Java and they are not meaningful to the computer hardware. The computer only recognizes simple data types.

Simple data types are known as 'primitives'. There are precisely eight primitive data types available to you in Java. Four of them are integers (`byte`, `short`, `int`, `long`), two are floating point numbers (`float` and `double`), one is an alphanumeric character (`char`) and one is a *true* or *false* value (`boolean`).

The various numeric data-types have different ranges of possible values. These are shown below. When creating a variable of a specific type you must be sure that the value falls within the supported range:

TYPE	SIZE	RANGE
Integers		
`byte`	8 bits	-128 to 127
`short`	16 bits	-32,768 to 32,767
`int`	32 bits	-2,147,483,648 to 2,147,483,647
`long`	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Floating Points		
`float`	32 bits, single precision	
`double`	64 bits, double precision	
Character		
`char`	16 bits of precision, unsigned	
Boolean		
`boolean`	true or false	

Since primitives have no methods, Java needs a way of wrapping up primitives inside something that *does* have methods which can be used upon the specific primitive data type. You can think of `Integer` as a wrapper for an `int` primitive and `Double` with a capital `D` as a wrapper for a `double` primitive. In fact, these wrappers are Java classes. I'll have a lot more to say about classes in the next chapter. It is these wrapper classes that supply methods such as `Integer.toString()` to convert an `int` to its string representation.

PRIMITIVES AND WRAPPER CLASSES

Primitive	*Wrapper*
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`boolean`	`Boolean`
`char`	`Character`
`double`	`Double`
`float`	`Float`

TYPE CASTS

Sometimes it may be useful to tell the Java compiler to treat one data type as another data type, assuming that the two data types are compatible (that is, it actually makes sense to treat one type as the other type). You can accomplish this by 'type casting'. For example, you may have declared a variable called `doubleVar` which is a `double` but you now want to assign its value to another variable, `floatVar`, which is a `float`. Both `double` and `float` are floating point numbers so they are compatible with one another apart from the fact that the two data types have different ranges (maximum and minimum values).

With floating point numbers, the 'larger' `double` data type is able to represent a floating point value more precisely than the 'smaller' `float` data type – that is, it may have more digits after the point.

The compiler will object if you attempt to assign the larger (`double`) value to the smaller (`float`) variable, like this:

```
floatVar = doubleVar;
```

In order to perform this conversion, you could 'cast' the `double` value to a `float` by placing the target type name (`float`) between parentheses in front of the source (`double`) value like this:

```
floatVar = (float) doubleVar;
```

But be careful! Remember that a `float` has a smaller range than a `double` so if the `double` value assigned is greater than the range of a `float`, that value will be truncated. Look at this code:

```
float floatVar;
double doubleVar;
doubleVar = 1.765987009800000052;
floatVar = (float) doubleVar;
outputTA.setText("doubleVar = "+ doubleVar +", floatVar = "+floatVar);
```

When this is run, this is the messages that is displayed:

```
doubleVar = 1.7659870098000001, floatVar = 1.765987
```

It is best to avoid type casting whenever possible. There are often neater and safer ways of handling type-conversions in Java.

Numerical class types such as `Integer` and `Double` have their own built-in methods to convert between compatible data types. For example, variables of the `Double` type can use the `floatValue()` to return a `float` value. As with a simple cast, the actual value of the returned `float` may differ from the original `double` value due to rounding. Here is a simple example showing a conversion from a `double d1`, to an `integer i`, using the `intValue()` method of the `Double` variable, and a conversion of the `int` value `i` back to a `double` using the `Integer` variable's `doubleValue()` method.

```
Integer i;
Double d1;
Double d2;
d1 = 105.123;
i = d1.intValue();
d2 = i.doubleValue();
outputTA.setText("i = " + i + ", d1 = " + d1 + ", d2 = " + d2);
```

This is the output:

```
i = 105, d1 = 105.123, d2 = 105.0
```

Notice that the final values of `d1` and `d2` are different because `intValue()` created an integer and the floating-point data was lost in the process. When a new `double` value is created from this integer value, a 0 is placed after the point.

NUMERIC LITERALS

When you enter a number such as 10 or 100.5 it is called a literal. In Java numeric literals are automatically assigned specific data types. An integer literal is assigned the `int` data type whereas a floating-point literal is assigned the `double` data type. That means that if you try to assign numeric literals to variables of other numeric types the Java compiler may complain. If, for example, you try to assign to a `long` variable a numeric integer value that is bigger than the maximum value of an `int`, the compiler (or your IDE, such as NetBeans) will complain that the integer number is too large. Look at this example:

```
long l;
l = 9223372036854477580;
```

Even though a `long` (such as the variable `l`) is capable of holding the value 9223372036854477580, this assignment is an error because an integer literal is treated as an `int` – and an `int` has a maximum value of 2147483647. I therefore need to tell the compiler that this numeric literal is a `long` value. I do that by appending an `L` to the number like this:

```
l = 9223372036854477580L;
```

Similarly, if you need to assign a floating point literal (which is by default treated as a `double`) to a `float` variable, you can append an `F` to the literal like this:

```
float f;
f = 3.4028235E38F;
```

If you are unsure of the minimum and maximum ranges of numeric types, the constants `MIN_VALUE` and `MAX_VALUE` provided by the wrapper classes for each type return these values. The *Primitives* sample program contains an example of this:

```
Byte.MIN_VALUE
Byte.MAX_VALUE
Short.MIN_VALUE
Short.MAX_VALUE
Integer.MIN_VALUE
Integer.MAX_VALUE
Long.MIN_VALUE
Long.MAX_VALUE
Float.MIN_VALUE
Float.MAX_VALUE
Double.MIN_VALUE
Double.MAX_VALUE
```

AUTOMATIC STRING CONVERSIONS

I said earlier that when I want to convert numeric values to strings, in order to display them in text fields, I need to call the `toString()` method using the appropriate numeric wrapper class, like this:

DataTypes

```
myTextArea.setText(Integer.toString(myInt + myOtherInt));
```

However, if you've been paying close attention to my code you may notice that I don't always follow this rule. In the *DataTypes* project, for example, I display the value of the `double` variable `total` without explicitly converting it to a string:

```
myTextArea.setText("Total = " + total);
```

How can this be? Let's try an experiment. You'll find the above line of code in the `parseStringsBtnActionPerformed()` method in the *DataTypes* project. Try editing it to get rid of the string `"Total = "` and the concatenation operator `+` so that the code now reads:

```
      myTextArea.setText(total);
```

If you now try to rebuild and run the project you will see an error message warning you that a `double` cannot be converted to a string. So how is it that my original version worked without problems by this new version does not? It turns out that when you concatenate strings, Java automatically calls the `toString()` method for each item that is not already a string. That means that the two lines of code shown below are equivalent:

```
myTextArea.setText("Total = " + total);
myTextArea.setText("Total = " + Double.toString(total));
```

BOXING/UNBOXING

Even though we have to use primitives in our programs, we frequently need to use the methods provided by their corresponding class wrappers. For that reason, Java provides a shorthand way of converting number *primitives* into number *objects*.

A number *object* is constructed from a number *class* and has access to numerous built-in methods.

When you assign a primitive to a variable with the type of the corresponding wrapper class, an object is created which contains the primitive value. That is called *autoboxing*. When you assign a number object to a variable of the corresponding primitive type, the primitive value which is contained by the number object is assigned to the primitive variable. That is called *unboxing*. You can find a simple example of autoboxing and unboxing in the *Primitives* project:

Primitives

```
Integer boxedInt;
int unboxedInt;

Double boxedDouble;
double unboxedDouble;

// autobox primitives into Integer and Double objects
boxedInt = 500;          // auto-boxing
boxedDouble = 200.25;

unboxedInt = boxedInt;   // auto-unboxing
unboxedDouble = boxedDouble;
```

STRINGS

We use strings all the time in our programs so to conclude this chapter, I will explain, in simple terms, what a string actually is. In Java, a string represents a series of characters. For example, the string "Hello" represents the five characters 'H','e','l','l','o'. A string can be entered as a series of characters delimited by double quotes (this is called a *string literal*) and the data type is `String`. A single character has the `char` data type. A `char` literal is delimited by single quotes – for example 'H'.

A Java string is, in fact, an object which contains both data (some characters) and various methods. String methods include functions to find characters and substrings, to return a copy of the string in upper or lower case and to return the length of the string. The first character in a string has the index 0. The last character in the string has an index given by the length of the string -1.

Assuming the string `s` contains the text, "Hello world". The expression, `s.length()`, would return a value of 11. This is how my code creates this string and assigns its length to the `int` variable `sLen`:

Strings

```
String s;
int sLen;
s = "Hello world";
sLen = s.length();
```

Here you can see I have passed the index values `0` and `sLen-1` to the `charAt()`method (which returns a character at a given index) in order to find the first and last characters in the string:

```
char c;
char c2;
s = "Hello world";
sLen = s.length();
c = s.charAt(0);           // c is now 'H'
c2 = s.charAt(sLen-1);     // c2 is now 'd'
```

If I want to use a lowercase or uppercase version of the string, I need to call the `toLowerCase()` or `toUpperCase()` methods on the string object. These methods return a new string with all the characters changed to the specified case.

```
lowcaseS = s.toLowerCase();
upcaseS = s.toUpperCase();
```

NOTE: You cannot *modify* a string object. For example, when I call the `toUpperCase()` method on the string object `s`, that string object remains unchanged (it still contains "Hello world"). The *modified* string ("HELLO WORLD") is returned by the `toUpperCase()` method and, in my code, I assign this returned value to a *new* string variable, `upcaseS`.

ESCAPE CHARACTERS

Notice that when I display the output in the *Strings* sample program I use a slash character, `\`, in front of the double-quote character in order to display a double-quote.

```
"\nIn lowercase, the string is \"" + lowcaseS + "\""
```

If I don't do that, Java thinks that the double-quote character is intended to be the end of the string literal. The slash character `\` tells Java that the character that follows it (such as `"` or `n` should be treated specially): `\n` is treated as a newline character, `\"` is a double-quote character. These are the escape characters recognised by Java strings:

Escape Character	Description
`\t`	Insert a tab in the text at this point.
`\b`	Insert a backspace in the text at this point.
`\n`	Insert a newline in the text at this point.
`\r`	Insert a carriage return in the text at this point.
`\f`	Insert a formfeed in the text at this point.
`\'`	Insert a single quote character in the text at this point.
`\"`	Insert a double quote character in the text at this point.
`\\`	Insert a backslash character in the text at this point.

Chapter 4 – Object Orientation

Java is an object oriented programming language. What this means is that almost everything you work with in Java – from a string such as "Hello world" to a file on disk – is wrapped up inside an object that contains the data itself (for example, the characters in a string) and the functions or 'methods' that can be used to manipulate that data – such as, for example, a string object's `toUpperCase()` method.

As we've already seen, primitives are exceptions to the rule that 'everything in Java is an object'. Primitives are simple pieces of data; they are *not* Java objects. However, primitives may be 'wrapped up' inside Java objects so an `int` primitive may be wrapped up inside an `Integer` object. This was explained in *Chapter 3*.

Classes and objects

Each actual object that you use is created from a 'class'. You can think of a class as a blueprint that defines the structure (the data) and the behaviour (the methods) of an object.

Let's look at an example of a very simple class definition. You define a class by using the `class` keyword. Here I have decided to call the class `Thing`. Notice that the initial letter of the class name is capitalized. This is the normal convention in Java and in many other programming languages. The class contains two String variables, `name` and `description`.

I've made these variables private by prefacing their declarations with the `private` keyword:

```
private String name;
private String description;
```

When variables inside a class are private, any code outside the class is unable to access them. In order to provide access to my private variables I have written these accessor methods:

```
public String getName() {
    return name;
}

public void setName(String aName) {
    this.name = aName;
}

public String getDescription() {
    return description;
}

public void setDescription(String aDescription) {
    this.description = aDescription;
}
```

In Java, when you want to give a new value to a private variable, you need to write a *set* accessor which, by tradition, uses the capitalized name of the variable preceded by `set`. So a new value can be assigned to the `name` variable, for example, using the `setName()` method.

When you want to retrieve the value of a private variable, you need to write a *get* accessor which, by tradition, uses the capitalized name of the variable preceded by `get`. So the value of the `name` variable is returned by the `getName()` method.

Each class should be created in a separate code file with the same name as the class. So, for example, I have written the `Thing` class in a file named *Thing.java*.

It is generally good practice to make variables private. By using methods to access data you are able to control how much access is permitted to your data and you may also write code in the methods to test that the data is valid.

This is the completed `Thing` class:

Objects/Thing.java

```
package myclasses;

public class Thing {

    private String name;
    private String description;

    public Thing(String aName, String aDescription) {
        // constructor
        this.name = aName;
        this.description = aDescription;
    }
    public String getName() {
        return name;
    }

    public void setName(String aName) {
        this.name = aName;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String aDescription) {
        this.description = aDescription;
    }
}
```

At this point, the class doesn't actually do anything. It is simply a definition of objects that can be created from the `Thing` 'blueprint'. Before we can use a `Thing` object, we have to create it from the `Thing` class. In the *Objects* sample program (the *NewJFrame.java* file), I declare two object variables of the `Thing` type:

```
Thing mySword;
Thing myRing;
```

Before I can use these objects, I need to create them. I do that by using the `new` keyword. This calls the `Thing` constructor method. This returns a new object which I can assign to a variable of the same type as the object's class.

```
Thing mySword;
Thing myRing;

mySword = new Thing("Dwarf Sword", "a sword of great power");
myRing = new Thing("Elvish Ring", "a golden ring of magic power");
```

CONSTRUCTORS

A *constructor* is a special method which, in Java, has the same name as the class itself. When you create a class with `new`, a new object is created, some memory is set aside to store the new object and any code in the constructor is executed. If you need to initialize some values (such as the `name` and `description` variables in the `Thing` class), you must pass these values as comma-delimited arguments between parentheses. If there are no arguments, you must call the constructor using empty parentheses.

This is the `Thing` constructor:

```
    public Thing(String aName, String aDescription) {
        this.name = aName;
        this.description = aDescription;
    }
```

Here `this` is a keyword that refers to the current object – the object that is being created. It could be omitted (I could have written the assignment `name = aName`) but it is used here for clarity.

The code in my constructor assigns the string values of the two arguments, `aName` and `aDescription` to the private variables `name` and `description`.

CLASSES, OBJECTS AND METHODS - SUMMARY

A 'class' is the blueprint for an object. It defines the data an object contains and the way it behaves. Many different objects can be created from a single class. So you might have one Thing ***class*** but many Thing ***objects***. A ***method*** is like a function or subroutine that is defined inside the class.

CLASS HIERARCHIES

One of the key features of object orientation is inheritance. Put simply, this means that you can create a 'family tree' of classes in such a way that one or more classes inherit the features of another class. The class from which features are inherited might be considered to be the ancestor of the classes that inherit features from it. The classes that inherit features from another class might be thought of as descendants of that class. A descendant class is also called a *subclass*. An ancestor class is often called a *superclass*.

To create a subclass, you need to place the keyword `extends` between the new class name and the name of its superclass. This is how I have declared a class named `NoisyThing` to be a subclass of the Thing class:

Objects/NoisyThing.java

```
public class NoisyThing extends Thing
```

A subclass inherits the features (the methods and variables) of its ancestor so you don't need to recode them. In this example, the `Thing` class has two variables, name and description along with four accessor methods, getName(), setName(), getDescription() and setDescription() to access those variables. The NoisyThing class is a descendent of the Thing class so it automatically inherits the name and description variables and their accessor methods.

In addition, the NoisyThing class adds on these two new variables:

```
  private String noise;
  private int volume;
```

Once again, I've added accessor methods to get and set the values of these variables:

```
  public String getNoise() {
        return noise;
  }

  public void setNoise(String noise) {
      this.noise = noise;
  }

  public int getVolume() {
      return volume;
  }

  public void setVolume(int volume) {
      this.volume = volume;
  }
```

To sum up, then, the NoisyThing class defines two variables: noise and volume; and it inherits another two variables, name and description, from its ancestor class Thing. It defines four accessor methods to get and set the values of the noise and volume variables and it *inherits* another four accessors to get and set the values of the name and description variables. When a new NoisyThing object is created it needs to assign values to all four variables. Its constructor method lists four named parameters so that data items to initialize the four variables can be passed to the constructor when a new NoisyThing object is created:

```
  public NoisyThing(String aName, String aDescription,
                    String aNoise, int aVolume)
```

The NoisyThing constructor starts by passing the relevant data items to its superclass's constructor so that the superclass can initialize its data – that is, the name and description variables defined by the Thing class. The NoisyThing constructor calls the superclass's constructor using the super keyword followed by any required arguments (data items) between parentheses. This is how I pass two data items to the superclass `Thing` constructor:

```
super(aName, aDescription);
```

Having done that, I go on to initialize the variables defined in the `NoisyThing` class:

```
this.noise = aNoise;
this.volume = aVolume;
```

When I create a new `NoisyThing` object I must be sure to pass four bits of data to its constructor so that the variables of its superclass `Thing` can be initialized in addition to the newly defined variables of `NoisyThing`.

```
NoisyThing myCat;
NoisyThing myDog;

myCat = new NoisyThing("Flossie", "a small tabby cat", "Miaaoow", 3);
myDog = new NoisyThing("Fido", "a big smelly dog", "Woof!", 10);
```

FUNCTIONS OR METHODS

Functions provide ways of dividing your code into named 'chunks'. In object oriented languages such as Java, functions are bound into objects and they are referred to as 'methods'. A method is declared using a keyword such as `private` or `public` (which controls the degree of visibility of the function to the rest of the code in the program) followed by the data type of any value that's returned by the function or `void` if nothing is returned. Then comes the name of the function, which may be chosen by the programmer. Then comes a pair of parentheses.

The parentheses may contain one or more 'parameters' separated by commas. The parameter names are chosen by the programmer and each parameter must be preceded by its data type. When a method returns some value to the code that called it, that return value is indicated by preceding it with the `return` keyword.

Here are some example methods:

Methods/NewJFrame.java

1) A function that takes no arguments and returns nothing:

```
private void doSomething() {
   outputTA.append("Hello".toUpperCase() + "\n");
}
```

2) A function that takes a single string argument and returns nothing:

```
private void doSomething(String aString) {
    outputTA.append(aString.toUpperCase() + "\n");
}
```

3) A function that takes two String arguments and returns a String:

```
private String doSomething(String aString, String anotherString) {
    return (aString + anotherString).toUpperCase() + "\n";
}
```

NOTE: In other languages, functions that return nothing may be called 'subroutines' or 'procedures'. In Object Oriented terminology, a function is often called a 'method' and, in this course, the terms 'function' and 'method' may be regarded as synonymous.

To execute the code in a method, your code must 'call' it by name. In Java, to call a method with no arguments, you must enter the method name followed by an empty pair of parentheses like this:

```
doSomething();
```

PARAMETERS AND ARGUMENTS

Parameters are the named variables declared between parentheses in the method itself, like this:

```
private String doSomething(String aString, int aNumber)
```

Arguments are the values passed to the method when that method is called, like this:

```
doSomething("Hello", 100 );
```

While computer scientists may make a distinction between the terms 'parameter' and 'argument', programmers often use the words interchangeably. So, informally, a method's 'parameter list' may be referred to as its 'argument list'.

When you call a method, you must pass the same number of arguments as the parameters declared by the method. Each argument in the list must be of the same data type as the matching parameter.

If a method returns a value, that value may be assigned (in the calling code) to a variable of the matching data type. Here, the `doSomething()` method returns a string and this is assigned to the `aStr` string variable.

```
String aStr;
aStr = doSomething("Hello, ", "dear programmer.");
```

It is also possible to use the return value of a method without first assigning it to a variable. For example, consider this code which assigns the return value from `doSomething()` to the string variable `aStr` and then passes that variable to the `append()` method of the text area, `outputTA`:

```
String aStr;
aStr = doSomething("Hello, ", "dear programmer.");
outputTA.append(aStr);
```

Since I never need to use the returned string value again, I don't really need to store that value in the `aStr` variable. I might as well pass the value returned from the method-call directly to the `append()` method. I can do that by rewriting my original three lines of code as single line of code (omitting the `aStr` variable) as shown below:

```
outputTA.append(doSomething("Hello, ", "dear programmer."));
```

METHOD OVERLOADING

Note, by the way, that you may use the same name for several different methods in Java as long as the list of parameters is different in each. The parameter list defines the '*method signature*' and when you create methods with the same names but different signatures, this is called '*method overloading*'. Not all languages permit method overloading.

Method overloading will be explained more fully in Chapter 9.

An example of a commonly-used overloaded method is the `indexOf()` method of the `String` class. When you call this method on a `String` object and past to it a single `String` argument, `indexOf(String str)` returns the index within the `String` of the first occurrence of the specified substring `str`. When passed a `String` argument and an `int` argument, `indexOf(String str, int fromIndex)` returns the index within this string of the first occurrence of the specified substring `str`, starting at the specified index, `fromIndex`. There are, in fact, four overloaded `indexOf()` methods supplied by the `String` class. While overloaded methods may be useful in certain cases, as a general rule you should not make a habit of using the same method names when there is an alternative.

STATIC

Usually, when you want to use a method provided by some class, you have to create a new object based on that class. Then you can call the method from that object. So, for example, if you want to get the Northern exit of a Room object you would first need to create a new Room object and then call the `getN()` method from that Room object like this:

The code fragment shown below is a slightly simplified version of some code from the *Adventure2* project.

```
Room goldRoom;
int exit;
goldRoom = new Room("room2", "Gold room", 0, 4, Direction.NOEXIT, 3)
exit = goldRoom.getN();
```

Sometimes, however, it may be useful call a method without having to create an object in order to do so. For example, let's suppose you want to convert a string such as "200" to its integer representation 200. You could do that by first creating a new `Integer` object and then calling the `valueOf()` method with the string as an argument, like this:

```
Integer intOb;
int anInt;
intOb = new Integer(0);
anInt = intOb.valueOf(s);
```

This is pretty ugly and inefficient code, however. I have created an `Integer` object which I never really need simply in order to call the conversion method, `valueOf()`which is defined inside the `Integer` class. It would be much neater if I could just call `valueOf()` without having to create an `Integer` object first. Well, I can do that. Instead of creating an *object* from the `Integer` class, I just call `valueOf()` from the class itself, like this:

```
Integer.valueOf(s);
```

You'll find an example of this in the *ClassMethods* project. I have a string variable `s` which has the value "200" (in a real-world program this value might have been entered into a text field by the user or it might have been read from a file on disk). When I want use this value in a calculation I simply call `Integer.valueOf(s)` to return an integer value:

ClassMethods

```
int x;
int total;
String s;
s = "200";
x = 5;
total = x * Integer.valueOf(s);
```

Many other standard Java classes provide methods that can be called in this way. For example, the `String` class provides the `format()` method which lets you create a string by embedding values at points marked by 'format specifiers' (just like those we used with the `printf()` function described in *Chapter 2*):

```
String.format("%d * %s = %d", x, s, total)
```

But you can't just call any method in this way. In order to be called directly from the *class* rather than from an *object*, a method has to be declared as `static`. These are the declarations of the `String.format()` and `Integer.valueOf()` methods:

```
public static String format(String string, Object[] os)
public static Integer valueOf(String string)
```

CLASS METHODS & INSTANCE METHODS

You can think of `static` methods as being 'class methods' – they 'belong' to the class itself. This contrasts with other methods which you can think of an 'instance methods' – they 'belong' to instances of the class. An 'instance' of a class is an object created from the class 'blueprint'.

If you want methods to be callable from your own classes, you need to add the keyword `static` to their declarations. Let's take a look at a class that includes a `static` method. Open the *Methods* project and view the source of *MyObject.java* in the `myclasses` package. This includes this `static` method:

Methods/MyObject.java

```
public static String numberOfObjects() {
        return "There are " + obcount + " objects.\n";
}
```

This displays a string that includes the value of the `int` variable `obcount`. You will see that this variable has been declared to be `static` and it has been initialized with a value of 0:

```
private static int obcount = 0;
```

A `static` variable, like a `static` method, 'belongs' to the class rather than to individual objects created from that class. In my code, the `MyObject` constructor, which is called whenever a new object (a new 'instance' of the `MyObject` class) is created, adds 1 to the value of `obcount`.

```
obcount = obcount + 1;
```

Since a `static` variable belongs to the class, there is only ever one copy of that variable so it can only ever have one value, no matter how many objects based on that class have been created.

The `MyClass` class also has an *instance* (non-static) variable, `obnumber`:

```
private int obnumber;
```

Each time a new object is created the constructor assigns the current value of `obcount` to this variable:

```
obnumber = obcount;
```

This means that each time a new object is created the `static` variable `obcount` is incremented and so it will be set to the total number of objects that have been created. There is only one copy of this `static` variable. So even if there are many objects, at any given moment this *static* variable will always have a *single* value. The *instance* variable `obnumber` is also incremented each time a new object is created. However, each object has its own copy of this *instance* variable so each object will have a *different* value for `obnumber`.

Let's suppose you were to create three objects, `ob1`, `ob2` and `ob3`. The `obnumber` of `ob1` would be the original value of the variable (0) + 1; so for `ob1`, its value would be 1; for `ob2` its value would be 2 and for `ob3` its value would be 3. But the `obcount` variable, *which belongs to the class rather than to any individual object*, would be the same for all three objects – namely 3.

In brief then, *instance* variables store different values for each object. *Static* or 'class' variables store a single value which is accessible by all objects created from that class. Instance methods are called from an object but class methods are called from a class.

> You can also call static methods from an object but this can be confusing and it is generally discouraged.

CONSTANTS

Recall from *Chapter 2* that constant values are generally declared using the `static` and `final` keywords. It is the `final` keyword that makes a variable into a constant – that is, it prevents its value from being changed after it has been assigned. The `static` keyword ensures that we only create one copy of the constant, stored by the class itself, rather than creating new copies for each object. For example, this is how the constant value of PI is declared:

```
public static final double PI = 3.141592653589793;
```

Chapter 5 – Conditional Tests and Operators

You will frequently need to test values in your programs to verify if one variable has a higher or lower value than another, for example. Java provides a number of 'operators' to help you do these tests. We've seen some operators already – for example we used the + (addition) operator and the = (assignment) operator in the Tax Calculator from *Chapter 3*. For example, we used added the values of two variables and assigned the result to a third variable using this expression:

```
grandtotal = subtotal + tax;
```

In this chapter we will look at a number of other operators too.

OPERATORS

So what exactly is an operator? Put simply, operators are special symbols that are used to do specific operations such as the addition and multiplication of numbers or the concatenation (adding together) of strings. One of the most important operators is the assignment operator, =, which assigns the value on its right to a variable on its left. Note that the type of data assigned must be compatible with the type of the variable.

This is an assignment of an integer (10) to an integer (int) variable:

```
int myIntVariable = 10;
```

Note that while *one* equals sign = is used to *assign* a value, *two* equals signs == are used to *test* a condition.

This is an assignment (let myIntVariable equal 1):

```
myIntVariable = 1;
```

This is a test for equality (if myIntVariable equals 1 then put the string "yes" into textField2):

```
if (myIntVariable == 1) {
      textField2.setText("Yes!");
}
```

You will find, in the *EqualsOps* sample project a slightly more complete example which tests a value entered by the user and displays "Yes!" if the value is 1 and "No!" if it is any other value.

EqualsOps

```
int myIntVariable;
myIntVariable = Integer.parseInt(textField1.getText());

if (myIntVariable == 1) {
    textField2.setText("Yes!");
} else {
    textField2.setText("No!");
}
```

There is a problem with this code. Before the user enters a value into `textField1`, that field is empty. If I try to convert its contents to an integer using `Integer.parseInt()` I will cause an error. I will explain some useful ways of recovering from errors when I discuss exception-handling in *Chapter 9*. Here, however, it will suffice to avoid the error by testing if the text field contains an empty string "" and, if so, displaying an error message instead of trying to convert that string to an integer. Here is my first attempt:

```
if (textField1.getText() == "") {
    textField2.setText("You must enter a number into textField1..");
} else  {
    myIntVariable = Integer.parseInt(textField1.getText());
    if (myIntVariable == 1) {
          textField2.setText("Yes!");
    } else {
          textField2.setText("No!");
    }
}
```

What I am trying to do here is to get the text from `textField1` and compare it with an empty string:

```
if (textField1.getText() == "")
```

If it is empty string, then I want to run this code:

```
    textField2.setText("You must enter a number into textField1..");
```

If it is not an empty string then I want to run all the code following the keyword `else`:

```
{
    myIntVariable = Integer.parseInt(textField1.getText());
    if (myIntVariable == 1) {
          textField2.setText("Yes!");
    } else {
          textField2.setText("No!");
    }
}
```

But that is not what happens. When `textField1` is empty, the code following `else` is run – which is not what I want at all. To understand why that happens, you have to understand how Java treats a string. Whereas an integer 1 can be compared with another integer 1 and they will be found to be the same (so the test evaluates to *true*), a string "1" compared with another string "1" may be found to be different (so the test evaluates to *false*). In order to determine if two strings containing identical characters (or no characters at all) I need to use the `equals()` method of the string object. This is how I have rewritten my test (which now works as I intended):

```
if ("".equals(textField1.getText())) {
    textField2.setText("You must enter a number into textField1..");
}
```

STRING EQUALITY

Let's consider why two apparently identical strings are considered to be different. It turns out that each string occupies its own chunk of memory and it is, therefore, considered to be different from every other string. This may be true even when two strings contain exactly the same characters.

Let's suppose you initialize two strings as follows:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
```

Now you compare `s1` and `s2` using a test such as if `(s1 == s2)`. While it might appear that this test should evaluate to *true*, in fact, the test evaluates to *false* because, even though the two strings contain the same characters, `s1` is a different string object (I've created each explicitly using `new`) to `s2`. The same will be true of any string obejcts that are created at runtime (for example, by assigning the text from a text field to a string variable).

However, when two identical strings are assigned to variables (without explicitly creating new string objects) they will reference or 'point to' a single string. That is because string literals created in your code prior to compiling an application are saved by Java into a table of strings and any variable that is assigned that string will reference this stored string value. As a consequence, two variables such as those below, which are assigned the same string will return true when tested for equality using the `==` operator:

StringCompare

```
String s1 = "Hello";
String s2 = "Hello";

(s1 == s2) // This evaluates to true
```

That is why Java strings have the `equals()` method. This method compares the characters in two strings and returns *true* if those characters are the same. When comparing two strings, always use the `.equals()` method. Unlike `==`, the results from comparisons using `equals()` are clear, consistent and unambiguous. The `equals()` method is, in fact, supplied for use with all Java objects, not just strings, to test if two variables refer to precisely the same object.

STRING ASSIGNMENT

When one String variable is *assigned* to another String variable using an expression such as `s2 = s1`, both variables refer *to the same String object* so the test `if (s2 == s1)` now evaluates to *true*. Look carefully at the code in the *StringCompare* project to help you understand this.

TESTS AND COMPARISONS

Java can perform tests using the `if` statement. The test itself must be contained within round brackets and should be capable of evaluating to *true* or *false*. If true, the statement following the test executes. A single-expression statement is terminated by a semicolon like this:

```
if (inputStr.equals(""))
    output.setText("You didn't enter anything");
```

A multi-line statement must be enclosed within curly brackets. Some programmers (myself included!) prefer to enclose single-lioje as well as multi-line statements between curly brackets for clarity and consistency. So the above example could be rewritten like this:

```
if (inputStr.equals("")) {
    output.setText("You didn't enter anything");
}
```

IF..ELSE

Optionally, an `else` block may follow the `if` and this will execute if the test evaluates to false. You may also 'chain together' multiple `if..else if` sections so that when one test fails the next condition following `else if` will be tested. Here is an example:

Tests

```
if (inputStr.equals("")) {
    output.setText("You didn't enter anything");
} else if ((inputStr.equals("hello")) || (inputStr.equals("hi"))) {
    output.setText("Hello to you too!");
} else {
    output.setText("I don't understand that!");
}
```

Remember that you will normally use the `.equals()` method with strings but you can use the equality operator `==` with numbers. You may use other operators to perform other tests on numbers. For example, the code shown below tests if the value of `i` is not equal to the value of `j` (here `!` is the 'not' operator and `!=` is the 'not equals' operator). If the values of the variables are not equal then the conditional evaluates to *true* and "Test is true" is displayed. Otherwise the condition evaluates to *false* and "Test is false is displayed:

```
int i = 100;
int j = 200;
if (i != j) {
    output.setText("Test is true");
} else {
    output.setText("Test is false");
}
```

These are the most common comparison operators that you will use in tests:

```
==        // equals
!=        // not equals
>         // greater than
<         // less than
<=        // less than or equal to
>=        // greater than or equal to
```

SWITCH..CASE

If you need to perform many tests, it is often quicker to write them as 'switch statements' instead of multiple `if..else if` tests. In Java a switch statement begins with the keyword `switch` followed by a test value. Then you must put values after one or more `case` tests. Each `case` value is compared with the test value and, if a match is made, the code following the `case` value and a colon (e.g. `case "hi":`) executes. When you want to exit the `case` block you need to use the keyword `break`. If `case` tests are not followed by `break`, sequential `case` tests will be done one after the other until `break` is encountered. You may specify a `default` which will execute if no match is made by any of the `case` tests. Here is an example:

```
String inputStr;
inputStr = userinput.getText();
switch (inputStr) {
    case "":
          output.setText("You didn't enter anything");
          break;
    case "hello":
    case "hi":
          output.setText("Hello to you too!");
          break;
    default:
          output.setText("I don't understand that!");
          break;
}
```

In this example, if `inputStr` is an empty string (`""`), it matches the first `case` test and "You didn't enter anything" will be displayed. Then the keyword `break` is encountered so no more tests are done. If `inputStr` is either "hello" or "hi", then "Hello to you too!" is displayed. This matches "hello" because there is no `break` after that test. If `inputStr` is anything else, the code in the `default` section is run, so "I don't understand that!" is displayed.

As you may have noticed, the Java `switch` statement tests the equality of the string specified in the test expression (at the start of the `switch` block) with the string specified after each `case` label as if it were performing a test using the `String.equals()` method rather than the `==` operator, so a match is made when the characters in the strings are the same.

Logical Operators

In some of the code examples in this chapter, I have used the `&&` operator to mean 'and' and the `||` operator to mean 'or'. The `&&` and `||` operators are called 'logical operators'. Logical operators can help you chain together conditions when you want to take some action only when *all* of a set of conditions are true or when *any one* of a set of conditions is true. For example, you might want to offer a discount to customers only when they have bought goods worth more than 100 dollars *and* they have also bought the deal of the day. In code these conditions could be evaluated using the logical *and* (`&&`) operator, like this:

```
if ((valueOfPurchases > 100) && (boughtDealOfTheDay))
```

But if you are feeling more generous, you might want to offer a discount *either* if a customer has bought goods worth more than 100 dollars *or* has bought the deal of the day. In code these conditions can be evaluated using the logical *or* (`||`) operator, like this:

```
if ((valueOfPurchases > 100) || (boughtDealOfTheDay))
```

You will find some examples of using these operators in the *LogicalOperators* sample program:

LogicalOperators

```
int age;
int number_of_children;
double salary;
age = 25;
number_of_children = 1;
salary = 20000.00;

if ((age <= 30) && (salary >= 30000.00)) {
    System.out.print("You are a rich young person\n");
} else {
    System.out.print("You are not a rich young person\n");
}

if ((age <= 30) || (salary >= 30000.00)) {
    System.out.print("You are either rich or young or both\n");
} else {
    System.out.print("You are not neither rich nor young\n");
}

if ((age <= 30) && (salary >= 30000.00) && (number_of_children != 0)) {
    System.out.print("You are a rich young parent\n");
} else {
    System.out.print("You are not a rich young parent\n");
}
```

BOOLEAN VALUES

Logical operators test Boolean values. A Boolean value can either be *true* or it can be *false*. Java provides a special `boolean` data type which can hold either a true value or a false value.

Booleans

```
boolean happy;
boolean rich;

happy = true;
rich = false;
```

It is possible to create quite complex conditions by chaining together tests with multiple `&&` and `||` operators. You may need to enclose multiple-part tests between parenthese to ensure that the the component tests are all evaluated together to return a true or false value. For example, by placing parentheses ariound the first two conditions here I can be sure that `((income > 100000 ) || (wonTheLottery))` is evaluated as a single test.

```
(((income > 100000 ) || (wonTheLottery)) && happy)
```

Be careful, however. Complex tests are often hard to understand and if you make a mistake they may produce unwanted side effects. For example, there are so many 'or' and 'and' conditions here that it becomes difficult to see the exact meaning of this test (and if I happened to put the parentheses in the wrong places, I might end up accidentally changing the meaning of the test):

```
if ((((income > 100000 ) || (wonTheLottery)) && happy) || (rich && happy))
```

COMPOUND ASSIGNMENT OPERATORS

Some assignment operators in Java perform a calculation prior to assigning the result to a variable. This table shows some examples of common 'compound assignment operators' along with the non-compound equivalent.

CompoundOperators

operator	**example**	**equivalent to**
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`

It is up to you which syntax you prefer to use in your own code. If you are familiar with other C-like languages, you will probably already have a preference. Many C/C++ programmers prefer the terser form as in `a += b`. Basic and Pascal programmers may feel more comfortable with the slightly longer form as in `a = a + b`. At any rate, you need to know and understand the effect of these operators in your own, or other people's, Java code.

INCREMENT ++ AND DECREMENT -- OPERATORS

When you want to increment or decrement by 1 (add 1 to, or subtract 1 from) the value of a variable, you may also use the `++` and `--` operators. Here is an example of the increment (`++`) and decrement (`--`) operators:

UnaryOperators

```
int a;
a = 10;
a++;      // a is now 11
a--;      // a is now 10
```

UNARY OPERATORS

`++` and `--` are called 'unary operators' because they only require one value, one 'operand' to work upon (e.g. `a++`) whereas binary operators such as `+` require two (e.g. `a + b`).

Prefix and Postfix Operators

You may place the `++` and `--` operators either before or after a variable like this: `a++` or `++a`. When placed *before* a variable, the value is incremented before any assignment is made:

```
int a;
a = 10;
b = ++a;              // a = 11, b = 11
```

When placed *after* a variable, the assignment of the existing value is done before the variable's value is incremented:

```
int a;
a = 10;
b = a++;              // a = 11, b = 10
```

Mixing prefix and postfix operators in your code can be confusing and may lead to hard-to-find bugs. So, whenever possible, keep it simple and keep it clear. In fact, there is often nothing wrong with using the longer form, which may seem more verbose but is at least completely unambiguous – e.g. `a = a + 1` or `a = a - 1`.

Chapter 6 – Arrays and collections

Whether you are developing a payroll application or an adventure game, you will often need to deal with collections of data items: a list of a company's employees, perhaps. Or a collection of all the treasures to be found in the Dragon's Lair in your game. Java provides various ways of handling collections of items. As in most other programming languages, the simplest way of creating a collection is to add items to a sequential list or array.

ARRAYS

You can think of an array as a set of slots in which a fixed number of items of the same type can be stored. As with just about everything else in Java, an array is an object. An array is declared by specifying the type of elements which it will store, such as `int`, `double`, `char` or `String`, followed by a pair of square brackets, the variable name and a semicolon. Here are four array declarations:

ArrayDeclarations

```
int[] intarray;
double[] doublearray;
char[] chararray;
String[] stringarray;
```

It is also permissible to place the square brackets after the variable name rather than the type name, like this:

```
int intarray[];
double doublearray[];
char chararray[];
String stringarray[];
```

In Java, it is usually more common to place the brackets after the type name, however, and that is the convention I adopt in this course.

Before you can use an array, you have to create it. This can be done, as with other types of object, using the keyword `new`. There is one extra complication with array objects however: each array needs to be given a size to indicate the maximum number of objects that it can contain. Here I create four array objects. The first array can contain one integer; the second can contain two doubles, the third can contain three characters and the fourth can contain five strings:

```
intarray = new  int[1];
doublearray = new double[2];
chararray = new char[3];
stringarray = new String[5];
```

Notice the syntax for creating an array. Following the keyword `new` I need to place the type name of each element, such as `int` or `String`, and then, between square brackets I specify the maximum number of elements of the specified type that can be stored in the array. When arrays are created, the designated number of 'slots' are automatically filled with a default value. For integers and floating points this is 0 or 0.0. For other object types it may be a non-printable `null` value.

ZERO-BASED ARRAYS

The first element of an array is at index 0. So let's suppose we have an array called `chararray` that contains the five characters: 'H', 'e', 'l', 'l', 'o'. The first character, 'H' is would be at the 'array index' 0, which we would write in code as `chararray[0]`, the second at index 1 and so on. You can think of an array as being like a container with a fixed number of slots numbered from 0 upwards like this:

Contents:	**'H'**	**'e'**	**'l'**	**'l'**	**'o'**
Index:	0	1	2	3	4

Notice that, since the first element in an array is at index 0, the last element is at the index given by the length of the array - 1. In the array shown above, there are five characters so the array length is 5. The first element 'H' is at index 0 and the last element 'o' is at index 4, that is the position given by the array length 5 - 1.

Be aware that while it is permissible to place fewer objects in an array than the maximum declared size, it is not permissible to add more objects to an array than the declared maximum size. So, for example, consider this `stringarray` variable, declared with a maximum size of 5:

```
stringarray = new String[5];
```

Here the length of the array is 5 so the index of the last element is 4. What happens, then if I try to add a string at index 5, like this?

```
stringarray[5] = "cloud";
```

If you try this you will find that when you try to compile and run your program, Java shows an message that says your code has caused an error of the type `java.lang.ArrayIndexOutOfBoundsException`. That is because you have tried to add an element beyond the end (the 'upper bound') of the array.

INITIALIZING ARRAYS

In the examples up to now, I declared array variables first then created array objects using the `new` keyword with the maximum size of the array between square brackets like this:

```
String[] stringarray;
stringarray = new String[5];
```

Having done this, I then added elements to the array, one by one, specifying the position by providing an array index between square brackets like this:

```
stringarray[0] = "I";
stringarray[1] = "wandered";
stringarray[2] = "lonely";
stringarray[3] = "as";
stringarray[4] = "a";
```

There is, however, a much simpler way of creating arrays and filling them with data items all in a single operation. This is how to create a 6-element array of strings in a single line of code:

```
String[] stringarray = {"I", "wandered", "lonely", "as", "a", "cloud"};
```

If you want to create and initialize arrays in this way you must start with the desired data-type such as `String` followed by an empty pair of square brackets `[]`, then a name for the array variable, here that's `stringarray`, the assignment operator `=` and finally a comma-delimited list of elements of the appropriate type enclosed between curly brackets. Notice that I have not use the `new` keyword, nor have I specified the array length in the square brackets. When you use this short-form syntax, a new array is automatically created with a length sufficient to hold the items between curly brackets and those items are placed sequentially into each available slot of the array. Here are some more examples:

```
int intarray[] = {0, 1, 2, 3, 4};
double doublearray[] = {1.2, 2.3, 3.4, 4.5};
char chararray[] = {'H', 'e', 'l', 'l', 'o'};
```

You can verify that the arrays have been created and filled with the list of items by iterating over the array elements using a `for` loop like this:

```
for (int i = 0; i < stringarray.length; i++) {
   System.out.printf("%s\n", stringarray[i]);
}
```

Here `length` is a 'property' or 'attribute' of the array. It stores the array size. Here `stringarray` has been initialized with six elements so `stringarray.length` evaluates to 6.

The code above produces this output:

```
I
wandered
lonely
as
a
cloud
```

In this code I am once again using the `printf()` function which lets me embed formatting specifiers such as `%d` (a decimal number), `%f` (a floating point number), `%c` (a character) and `%s` (a string) as explained in Chapter 2.

FOR LOOPS

When you want to execute some code repeatedly, you could a `for` loop to run one or more lines of code for a predetermined number of times. A `for` loop takes the form of three parts separated by semicolons and placed between parentheses. The basic syntax is:

PARTS OF A FOR LOOP

```
for (initialization; test; increment) {statement(s);};
```

`initialization`	This executes once when the loop starts. Normally the loop counter variable is assigned its initial value here.
`test`	This is evaluated every time the loop runs. If it evaluates to *true*, the loop continues. If it evaluates to *false*, the loop ends.
`increment`	This is evaluated every time the loop runs. Normally the loop counter variable is incremented here

Here is a simple example that executes 10 times and displays ten integers from 0 to 9:

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

The first part initializes the value of a counter variable, the integer `i`, which is assigned the value 0: `int i = 0`; the second part tests the value of the counter variable and stops running the loop when that test condition evaluates to **false** (here the loop continues running as long as `i` is less than 10: `i < 10`; as soon as it has the value or 10 or more, the loop ends); the third part increments the value of the counter variable: `i++` adds 1 to the value of `i`. The code that executes at each turn through the loop is placed between a pair of curly brackets. In this case, it simply prints out the value of `i`.

While it is often convenient to initialize a counter variable with 0, especially when working with arrays or strings, whose first items have the index 0, this is not obligatory. For example, suppose you wanted to count from 1 to 10 instead of from 0 to 9. One way of doing this would be to initialize `i` to 1 and run the loop while `i` is *less than* ***or*** *equal to* 10. In order to do that you could rewrite the for loop above as follows:

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

Recall that the operator `<` means 'less than' whereas the operator `<=` means 'less than or equal to'.

Finally, here is another example of a `for` loop:

```
for (int i = 0; i < s.length(); i++) {
    System.out.printf("'%c' ", s.charAt(i));
}
```

In this case I use it to iterate over the characters in a string. The first character is at index 0 and the last character is at an index of one less than the length of the string. The string length can be returned by the `length()` method as I explained in *Chapter 3*. So to display all the characters, one at a time (the character at each index in the string is given by the `charAt()` method), I count through the characters at each index starting at 0 (the first character) and ending when the test condition (`i < s.length()`) is no longer true. Since the length of the string is one greater than the index of its final character, the end condition will be true when `i` is greater that the length of the string. At that point the code inside the loop will no longer be run. So the test condition `i < s.length()` ensures that the code inside the loop will only run while `i` has a value that less than the length of the string.

GARBAGE COLLECTION

C and Pascal programmers may be alarmed to see that I keep creating objects using `new` but I never bother to dispose of them (to free up their memory) afterwards. I don't do this because I don't need to. In Java, when objects are no longer required (i.e. they are no longer referenced by anything in your program), Java gets rid of them for you. Once upon a time, garbage-collection languages were rather rare. These days, many languages ranging from C# to Ruby provide automatic garbage-collection.

ArrayList

Arrays have one big limitation: they are fixed in size. If you create an array to hold 10 elements you can't add an eleventh or a twelfth item. When you need to work with variable-length lists of objects, Java has some more flexible alternatives to arrays. In fact, Java has an entire 'Collection Framework' which provides a number of special-purpose classes and interfaces for managing both lists and networks of linked objects.

Interfaces

An interface is like a sort of abstract class. It defines the methods that a class must implement but it does not contain the actual code of those methods. A Java class is able to implement an interface by providing the actual code for each of the interface's predefined methods. This will be explained in more detail in Chapter 8. For now it is sufficient to know that the Collection framework contains various classes such as `ArrayList` which implements predefined Collection interfaces such as the `List` interface. Interfaces are explained more fully in Chapter 8.

Java's `ArrayList` is a useful general-purpose class that lets you create resizable sequential lists of elements. An `ArrayList` may either contain elements of mixed data types or it may be typed to ensure that only elements of a specific type can be added.

Here is an example of an untyped `ArrayList`:

ArrayLists

```
ArrayList aList;
```

Before it can be used, an `ArrayList` has to be created just like any other Java object, using the keyword `new` followed by the name of the class's constructor method:

```
aList = new ArrayList();
```

Now you can use the various methods available to `ArrayList` objects such as `add()` to add an element and `get()` to retrieve an element at a specific index:

```
aList.add("hello");
aList.add(20);
System.out.println(aList.get(0));
System.out.println(aList.get(1));
```

Note that it is permissible to store elements of mixed types – such as integers and strings – in an untyped `ArrayList` object. But if you want to make sure that only elements of a fixed type are allowed, you need to place the type name of the allowable elements between a pair of angle-brackets after the `ArrayList` class name when you declare the variable, like this:

```
ArrayList<String> stringList;
```

Then, when you create a *typed* `ArrayList` object you need to place an empty pair of angle brackets before the parentheses of the `ArrayList` constructor, like this:

```
stringList = new ArrayList<>();
```

NOTE: Some programmers like to repeat the element-type name between the angle brackets like this: `stringList = new ArrayList<String>();` but in modern versions of Java (from Java 7 onwards) this is not necessary. The empty pair of angle brackets is sometimes called the 'diamond operator'.

Now you can use the methods of `ArrayList` to add, get or remove string elements like this:

```
stringList.add("hello");
stringList.add("20");
System.out.println(stringList.get(0));
System.out.println(stringList.get(1));
stringList.remove(0);
```

You cannot add other element types, however. For example, this is not allowed:

```
stringList.add(20);   // Not allowed. stringList is typed for strings
```

GENERICS

A typed `ArrayList` such as my `stringList` object is an example of Java's 'generics'. A generic list is one that provides the behaviour needed to operate on lists of different types of element. In other words, the list-manipulation code is 'general purpose' and I am able to use it with specific types of object by specifying that object name between angle-brackets. Just as I created an `ArrayList` object to store String elements, I could create another `ArrayList` object to store Integers like this:

```
ArrayList<Integer> intList;
intList = new ArrayList<>();
intList.add(100);
intList.add(200);
System.out.println(intList.get(0));
System.out.println(intList.get(1));
```

Be sure to specify classes such as `Integer` rather than primitive types such as `int`. Generic lists operate on objects, not primitives.

Of course, just as I cannot add an Integer to an `ArrayList` typed to hold strings, I cannot add strings to an `ArrayList` typed to hold Integers, so this is not permitted:

```
intList.add("hello");       // Not allowed. intList is typed for Integers
```

If you want to try out the other methods available to `ArrayList`, refer to Oracle's online documentation. Here I'll just mention two more very important methods: `remove()` which removes an element at an index and `size()` which returns the number of elements:

```
stringList.remove(0);
System.out.println("Now stringList size = " + stringList.size());
```

NOTE: This is a very simple introduction to generics. This topic is covered in much greater depth in Chapter 9.

MAPS

Sometimes it may be useful to index a collection of objects by something other than their numerical positions in a list. For example, if you were to publish an English, French or Welsh dictionary it would be useful to be able to look up the definition of a word by searching for the word itself rather than its page number. In programming terms, the entries in a dictionary would be said to be made up of key-value pairs where the key is the word itself and the value is its associated definition.

In Java, you can create dictionary-like collections with something called a Map. `Map` is the name of an interface which defines a collection of key-value pairs in which the key is used to look up a value.

In some programming languages, this type of collection may be called an 'associative array', a 'hash' or a 'dictionary'. In fact, Java also has a `Dictionary` class which supplies similar functionality to a `Map` but that class is now considered to be obsolete and it is recommended that programmers use classes that implement the `Map` interface instead of `Dictionary`.

Just as `ArrayList` is a useful general-purpose implementer classes of the `List` interface, `HashMap` is a useful general-purpose implementer class of the `Map` interface. For a simple example of using `HashMap` to store a collection of elements in the form of key-value pairs, take a look at the *Maps* sample program.

Maps

First I declare and create a `HashMap` object, `hmap`:

```
HashMap hmap;
hmap = new HashMap();
```

Then I add paired elements to it using the `put()` method:

```
hmap.put("sword", "A glittering Elvish sword");
hmap.put("ring", "A golden ring of great power");
hmap.put("wombat", "A small burrowing creature (asleep)");
```

Here the first item of each pair is the 'key' and the second is the 'value'. I can retrieve an element using its key to give me access to the key's associated value. For example, here I use the `get()` method with the key "ring":

```
System.out.println(hmap.get("ring"));
```

This returns the value that is paired with the key “ring” and this is what is displayed at the system prompt:

```
A golden ring of great power
```

As my `HashMap` object is untyped it is able to store elements in which the key and the value may be any type of object. Here, for example, I add one item with the integer key 10 and the String value "A mysterious number" and another item with the string key "five" and the integer value 5:

```
hmap.put(10, "A mysterious number");
hmap.put("five", 5);
```

Once again, I can find the values using the keys:

```
System.out.println(hmap.get(10));
System.out.println(hmap.get("five"));
```

This displays:

```
A mysterious number
5
```

I can remove an element using the `remove()` function:

```
hmap.remove("ring");
```

I can obtain a collection of all the values in the `HashMap` using the `values()` method and a set of all the keys using the `keySet()` method.

```
System.out.println(hmap.values());
System.out.println(hmap.keySet());
```

This displays (after the “ring” element has been removed):

```
[A glittering Elvish sword, A mysterious number, 5, A small burrowing
creature (asleep)]
[sword, 10, five, wombat]
```

SETS

A set is a collection that contains no duplicates. Just as a sequential array list cannot have two identical indexes, so a `Map` cannot have two identical keys. That is why its keys form a 'set'. It may, however, have two identical values.

As with an `ArrayList`, it may sometimes be useful to enforce strict type-checking on the elements that can be added to a `HashMap`. Just as with `ArrayList`, you can do this by using generics to specify the allowed data types when you declare a `HashMap` object. For example, if I wanted to ensure that every element had an Integer key and a String value I would use a declaration similar to the following:

```
HashMap <Integer, String> map;
```

And I could use the 'diamond operator' `<>` when creating the object:

```
map = new HashMap<>();
```

NOTE: it is also permissible to define an object using one of the interfaces which its class implements, and you will commonly see `HashMap` objects defined using the `Map` interface like this:

```
Map <Integer, String> map = new HashMap<>();
```

Defining objects (such as `HashMap`) in this way makes them compatible with the declared interface (such as `Map`) when, for example, you may want to pass that object to a method that is capable of handling *any* object that implements the `Map` interface (not just `HashMap` objects).

You can find an example of a typed `HashMap` in the *RoomMap* sample program. Here I have added a slightly simplified version of the `Room` class from my Adventure game. Each `Room` object has four exits: `N`, `S`, `W` and `E` which returns a number indicating the room at that exit. When I create a new `Room` I give it a name which acts as its key in the `Map` collection. That means that, to ensure type safety, my `HashMap` needs to be typed to accept `String` keys and `Room` values. This is how I have done that:

RoomMap

```
HashMap <String, Room>map;
map = new HashMap<>();
```

To create the map I put key-value pairs into the `HashMap` like this:

```
map.put("Dark Cave", new Room( -1, 2, -1, 1));
map.put("Troll Room", new Room(-1, -1, 0, -1));
map.put("Dragon's Lair", new Room(0, 4, -1, 3));
map.put("Twisty Maze", new Room(-1, 5, 2, -1));
map.put("White House", new Room(2, -1, -1, 5));
map.put("Narrow Passage", new Room(3, -1, 4, -1));
```

To print the keys and values in the collection, I use a special type of `for` loop that iterates over all the keys returned by the `keySet()` function and displays the key and the value of each element:

```
for( String key: map.keySet()){
      Room r;
      r = map.get(key);
      System.out.println(String.format(
          "%s\thas exits: \tN = %d \tS = %d \tW = %d \tE = %d",
          key, r.n, r.s, r.w, r.e ));
}
```

I'll be explaining this alternative type of `for` loop, and some other types of loop, in the next chapter.

Incidentally, if you were to use a `HashMap` to create a map in a fully-working game, the integers that represent each exit would not be all that useful. These assume that each `Room` object has a fixed position in a sequential list such as an `ArrayList` so the exit numbers can be used to index into the map to find a specific room. But the elements in a `HashMap` are not stored in a fixed order.

In general, it is not safe to assume any type of ordering of the elements in a Map. Java does, however, provide the SortedMap interface which does sort elements according to their keys.

When using a HashMap to implement the map in a game, it would be useful to rewrite the `Room` class so that the exits are strings rather than integers. If, for example, a `Room` has at its `N` exit the string "Dark Cave" I would be able to use that string to key into the map in order to locate the `Room` associated with the key "Dark Cave". If you want to explore the possibilities of Maps in more depth, you could try rewriting my *RoomMap* program to using string keys in this way.

Chapter 7 Loops

Computer programs often have to perform repetitive actions such as printing a series of strings until there are no more left to print or counting the number of treasures which a game-player has collected. In this chapter I am going to look at ways of performing repetitive tasks in Java using loops.

FOR LOOPS

We've already used `for` loops a few times in previous projects. As explained in Chapter 6, the traditional type of `for` loop is declared with three parts: *initialization*, *test* and *increment*:

```
  for (initialization; test; increment) {statement(s);};
```

The *initialization* expression initializes the loop. Typically this takes the form of an integer loop variable which is assigned a starting value (for example `int i = 0`). This is executed once only when the loop begins.

The *statements* in the `for` loop continue to execute only as long as the *test* expression evaluates to **true** (for example, while `i < 10`). When it evaluates to **false**, the loop terminates.

The *increment* expression is executed after each iteration through the loop. Typically this increments the loop variable (for example `i++`), but it may also decrement it.

In following example, the loop variable `i` is initialized with the value 0 when the loop begins and it is incremented by 1 each time the loop executes. The statements in the loop – here `System.out.println(i)` – execute at each turn through the loop. The loop ends when the test value `i < 10` is no longer true. In other words, the loop runs *while* `i` is less than 10:

```
for (int i = 0; i < 10; i++) {
        System.out.println(i);
}
```

This is the output that is displayed when this loop runs:

```
0
1
2
3
4
5
6
7
8
9
```

While the example above is typical of the way that `for` loops are written, you should be aware that the initialization, test and increment sections may contain other sorts of expression. For example, here is a `for` loop in which the loop variable is initialized with 20, its value is decremented (1 is subtracted) at each iteration and the termination test is `i != 10`, so it runs just as long as the value of `i` is *not equal* to 10:

```
for (int i = 20; i != 10; i--) {
    System.out.println(i);
}
```

So this loop counts down from 20 to 11 (it ends when `i` is 10) and this is displayed:

```
20
19
18
17
16
15
14
13
12
11
```

Often `for` loops are used to iterate through the items in linear lists such as arrays or `ArrayList`s. This loop prints the items in the `somewords` array of strings from 0 to 2 – that is because the test executes the loop while `i` is less than the length of the somewords array. The length of the array is 3 so the code in the loop does not run when i has the value 3:

```
String[] somewords = {"one", "two", "three"};

for (int i = 0; i < somewords.length; i++) {
    System.out.println(i + " : " + somewords[i]);
}
```

This produces the following output:

```
0 : one
1 : two
2 : three
```

ENHANCED FOR (FOR...EACH) LOOPS

We have also seen an alternative type of for loop which has no loop variable or termination test. This type of loop iterates through all the items in a collection. You may recall that in Chapter 6 I used this loop to iterate through all the Strings returned by the HashMap object's keyset() method:

This type of loop is sometimes called an '*enhanced for*' statement and it is similar to the foreach or for..in loops provided by some other programming languages. The syntax of an enhanced for loop can be summarised as follow:

```
for (type variableName: collection) {statement(s);};
```

Here type is the type-name of the elements returned by the collection. This is an example which prints each string in the somewords array:

```
for (String word:somewords){
    System.out.println(word);
}
```

This is the output:

```
one
two
three
```

WHILE

As we've seen, `for` loops are useful when you have a known number of items to iterate through – from 0 to 4, say. But sometimes you may not know in advance how many items you need to process. For example, you might write some code to format lines of text in a file. But each file may contain a different number of lines so you cannot assume, in advance, that there will be a specific number of lines to process. One way of doing this is to use a `while` loop which continues to execute as long as some test condition remains true. This is the syntax of a `while` loop:

```
while (condition)  { statement(s); }
```

There are some basic examples of `while` loops in the *WhileLoops* sample program. Here I create an array, `intarray`, of five integers and use a `while` loop to count through the elements of that array while the value of `i` is less than 5. Since `i` starts out with the value of 0 this loop would continue for ever unless the value of `i` is incremented so that the test condition will eventually be met. That's what I do here:

WhileLoops

```
int intarray[] = {1, 2, 3, 4, 5};
int i;
i = 0;
while (i < 5) {
    System.out.printf("%d\n", intarray[i]);
    i++;
}
```

You need to be very careful to ensure that every `while` loop has a test condition that will eventually evaluate to **false**, otherwise you risk creating an endless loop, which is almost never a good thing to do! You also need to be aware that sometimes a loop condition may never be met – that is, it may evaluate to **false** immediately so the code never executes, as in this example:

```
i = 2;
while (i < 2) {
    System.out.printf("%d\n", intarray[i]);
    i++;
}
```

For a more useful example of a `while` loop in a 'real life' program, take a look at the *ReadFile* sample program. This uses a `RandomAccessFile` object called `file` to read each line from in a text file using the `readLine()` method until no more lines are available (when null is returned). The total number of lines read is assigned to the int variable linecount. The program displays each line (the string variable line), preceded by its number, in a text area, ta. Finally, it displays the number of lines read from the file. This program contains quite a bit of file-handling and error-recovery code which we will learn about in later chapters. For now, concentrate on the code fragment below which shows how the while loop is used:

ReadFile

```
while ((line = file.readLine()) != null) {
    linecount++;
    ta.append(String.format("[%d] %s\n", linecount, line));
}
ta.append("\nTotal lines in file = " + linecount);
```

DO..WHILE

In some circumstances, you may want to be certain that the code runs at least once – say, for example, if your program is downloading a huge file from the internet and after every ten minutes you want the user to confirm whether or not the operation should be continued. You would want to be sure that the user is asked for confirmation *at least once* (and possibly more than once). One way of doing that would be to write a do..while loop.

A do..while loop is essentially the same as a while loop, apart from the fact that the test condition is placed at the *end* of the loop rather than at the beginning. This means that the statements inside the loop are bound to execute *at least once* before the test condition is even evaluated. This is the syntax of a do..while loop:

```
do {statement(s)} while (condition);
```

Here is a very simple example of a do..while loop:

```
i = 2;
do {
    System.out.printf("%d\n", intarray[i]);
    i++;
} while (i < 2);
```

As in the while loop example given earlier, the condition evaluates to **false** the first time it is tested. But since this test is only performed after the code in the loop, that code is run once and this output is shown (the integer at index 2 in `intarray`):

```
3
```

MULTIDIMENSIONAL ARRAYS

I explained arrays in the previous chapter. In addition to simple linear arrays you can also create arrays of arrays – that is, 'multidimensional arrays'. This sort of array poses a special challenge when you need to iterate over its elements. Before looking at this problem, we need to be clear on exactly what a multidimensional array is.

A simple linear array can be thought of a single row of elements indexed from 0 to some upper limit. A multidimensional array can have both rows and columns like a spreadsheet: one dimension stores the rows, the other dimension stores the columns (in fact, arrays can have more than two dimensions but for our purposes two is sufficient). Conceptually a two-dimensional array forms a sort of 'grid' and you will find an example of this in the *MultidimensionArray* sample project.

Just as with single-dimensional arrays, I can initialize the elements of a multidimensional array by putting each element between curly brackets and assigning them to the array name at the time of its declaration. In this case, one array (the 'outer' array) contains some other arrays (the 'inner' arrays) as its elements, so I need to initialize the arrays by placing one set of curly brackets to delimit the outer array; and as with any array, this contains a comma-delimited list of elements. In this case, the elements happen to be some more arrays (the 'inner arrays'). Each inner array is delimited by its own pair of curly brackets. Here the inner arrays contain comma-delimited lists of integers:

MultidimensionArray

```
int grid [][] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10},
    {11, 12, 13, 14, 15},
    {16, 17, 18, 19, 20}
};
```

In my code I have declared an array named `grid` that contains 4 arrays, each of which contains 5 integers. You can think of this as a sort of matrix with 4 rows and 5 columns. If this were a spreadsheet, the intersection of each row and column would form a 'cell' and the contents of each cell would be an integer: 1, 2, 3, 4 and so on.

Note that I have had to declare my `grid` array using two sets of square brackets – one for each dimension: `int grid[] []`

If you want to iterate over the outer array you could use a simple `for` loop. And, using the `Arrays.toString()` method I could display the four arrays of integers which it contains like this:

```
for (int row = 0; row < 4; row++) {
    System.out.printf("row %d : %s\n", row, Arrays.toString(grid[row]));
}
```

This shows the following:

```
row 0 : [1, 2, 3, 4, 5]
row 1 : [6, 7, 8, 9, 10]
row 2 : [11, 12, 13, 14, 15]
row 3 : [16, 17, 18, 19, 20]
```

Here the arrays of integers have been converted to their string representation. How can I process these arrays so that I can get at each individual integer? The answer of course is: *use a* `for` *loop*. I am already using a `for` loop to iterate through the elements of the *outer* array (each element being one of the arrays of integers), so I need another `for` loop to iterate through the elements (the integers) which each of these *inner* arrays contain. Here I'll explain how to do that.

To access a specific piece of data I need to give its 'cell location' using square-bracket notation with the 'row' index in one pair of brackets and the 'column' index in another pair of brackets. This is how I would print out the value stored at row 1 and column 2 (remember arrays begin at index 0 so this gives the item on the second row in the third column) – namely, the integer 8.

```
System.out.printf("%d\n", grid[1][2]);
```

So first I need one `for` loop (as in my previous example) to iterate over the outer array to get at the 'rows' it contains (where each row is one of the 'nested arrays' of integers):

```
for (int row = 0; row < 4; row++)
```

This `for` loop initializes the `row` variable to the index of one of the 'nested' arrays on each turn through the loop. Now, in order to iterate over the items stored in each nested array (which I think of as the 'columns' of the grid), I need another `for` loop inside the first `for` loop, like this:

```
for(int column = 0; column < 5; column++ ) {
```

In my sample code, this is how I iterate over all the items stored in my `grid` two-dimensional array and display the 'row' number followed by the 'column' numbers and data (the `int`) stored at that location:

```
for (int row = 0; row < 3; row++) {
   System.out.printf("--- row %d : %s --- \n",
                     row, Arrays.toString(grid[row]));
   for (int column = 0; column < 5; column++) {
      System.out.printf("column[%d] = %d\n", column, grid[row][column]);
    }
}
```

This is what is displayed when the program runs:

```
--- row 0 : [1, 2, 3, 4, 5] ---
column[0] = 1
column[1] = 2
column[2] = 3
column[3] = 4
column[4] = 5
--- row 1 : [6, 7, 8, 9, 10] ---
column[0] = 6
column[1] = 7
column[2] = 8
column[3] = 9
column[4] = 10
--- row 2 : [11, 12, 13, 14, 15] ---
column[0] = 11
column[1] = 12
column[2] = 13
column[3] = 14
column[4] = 15
--- row 3 : [16, 17, 18, 19, 20] ---
column[0] = 16
column[1] = 17
column[2] = 18
column[3] = 19
column[4] = 20
```

As I said earlier, the inner or 'nested' arrays in this sample project are all of equal length. This creates a matrix or grid-like structure in which each element can be thought of as occupying a 'cell' formed by the intersection of rows and columns.

In fact, it is not a requirement to make nested arrays of equal length. Look at the code in the *MoreMultidimensionalArrays* project for an example of nested arrays of varying length which I iterate over using nested `for` loops:

MoreMultidimensionalArrays

```
int[][] intgrid = {
    {1, 2, 3},
    {4, 5, 6, 7},
    {8, 9 },
    {10, 11, 12, 13, 14, 15}
};

int numrows = intgrid.length;

System.out.printf( "There are %d rows.\n", numrows);

for (int row = 0; row < numrows; row++) {
    System.out.printf("--- row %d --- \n", row);
    for (int index = 0; index < intgrid[row].length; index++) {
        System.out.printf("index[%d], value=%d\n", index,
          intgrid[row][index]);
    }
}
```

The important thing to observe here is that I iterate over the elements in the nested arrays up to a higher value that is calculated from the length of each array, which may be different each time:

```
    for (int index = 0; index < intgrid[row].length; index++)
```

BREAK AND CONTINUE

There may occasionally be times when you want to break out of a loop right away – even if the loop condition does not evaluate to *false*. You can do this using the `break` statement. Just as `break` was used to cause an immediate exit from a `switch..case` block (as explained in Chapter 5), so too it will cause an immediate exit from a block of code that is run in a loop.

Look at the code that follows:

BreakAndContinue

```
int i;
i = 0;
while (i < 10) {
    if (i == 5) {
        break;
    }
    ta.append(String.format("i = %d\n", i));
    i++;
}
```

The `while` loop has been set to run as long as `i` is less than 10. But inside that loop I test if `i` is 5 and, if so, I exit the loop with `break`. This means that in spite of the `while` condition expecting to run the loop ten times (for values of `i` between 0 and 9) in fact, when `i` is 5 the loop stops running. The code following `break` is not run so this is what is actually displayed:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

Be careful when breaking out of a loop in this way. In a more complex loop, containing many lines of code, the `break` condition may not be obvious and it would be easy to assume that the loop is guaranteed to run while `i` is less than 10. As a general principle, it is best (clearer and less bug-prone) to avoid breaking out of loops unless you have a very good reason for doing so.

Let me turn to a more useful example of a loop that uses a `break`. Here, when the user clicks the 'Encrypt' button the code of the `encryptBtnActionPerformed()` method 'encrypts' a string, `testStr`, by adding 1 to the numeric (ASCII) value of each character in that string. It does this by getting each character at index `i` from 0 to the end of the string and adding 1 to that character's value:

```
testStr.charAt(i) + 1
```

It then casts the new integer value to a character `(char)` and appends this character to another string variable `str`:

```
str += (char) (testStr.charAt(i) + 1);
```

Suffice to say, this is about as simple as an encryption algorithm can get. Even so, it serves to illustrate the basics of how to process strings or arrays of elements using loops. In principle my code could encrypt strings entered at the keyboard or read from a file. For simplicity, however, I have created the string shown below, in order to illustrate how the program works:

```
    String testStr = "Hello world! Goodbye";
```

I set up a `for` loop to iterate over the characters from 0 to the end of the string. However, I have made it a requirement that each string should be terminated by the `'!'` character. When that character is found the rest of the string is ignored because at that point I break right out of the `for` loop:

```
if (c == '!') {
    break;
}
```

There is one other character that I want to treat specially. When a space is encountered – for example, the space character between the two words in "Hello world" – I want to retain that space character rather than encrypt it. This is how my code does that:

```
if (c == ' ') {
    str += c;
    continue;
}
```

The `continue` statement is a bit like a less dramatic version of `break`. Whereas `break` exits the loop and *stops* running the code in the loop any more, `continue` exits the loop at this turn through the loop but then *carries on* running the code in the loop.

So, in my example, if I `break` on `'!'`, the code in the loop stops running. Given the string `"Hello world! Goodbye"` the code would process up to the `'!'` character and no further. But the code will `continue` when it processes the space `' '`. That means it exits the loop when `' '` is found after *"Hello"* but then it continues running the loop to process the rest of the characters, *"world"*.

As with `break`, you should use `continue` with caution. There may sometimes be perfectly good reasons for jumping out of code blocks using `break` or `continue` but jumps like this can make your code hard to understand and, in complex programs, they may result in subtle bugs.

There is another example of `break` and `continue` in the code that translates the encrypted string `str` (by subtracting 1 from each character) back to the unencrypted version. This time, it copies the unencrypted characters into the string `str2`. Since the space characters in the string `str` are not encrypted, this code ignores spaces by executing `continue` when one is found. But it `break`s when the string terminator `'!'` is found:

```
for (i = 0; i < str.length(); i++) {
    c = str.charAt(i);
    if (c == ' ') {
        str2 += c;
        continue;
    }
    if (c == '!') {
        break;
    }
    str2 += (char) (str.charAt(i) - 1);
}
```

The *BreakAndContinue* project also contains an example of breaking out of a `while` loop (see the `whileBtnActionPerformed()` method). This runs a `while` loop with no valid end condition (since `i` will always be greater than or equal to 0):

```
i = 0;
while( i >= 0 )
```

It displays the character and numeric code of each item in `testStr` and it builds a new string `str` that ends when the `'!'` character is found. A loop with no valid end condition will run forever unless you explicitly `break` out of it. In this loop, when `'!'` is found it breaks:

```
private void whileBtnActionPerformed(java.awt.event.ActionEvent evt) {
    int i;
    char c;
    String str = "";
    i = 0;
    while (i >= 0) {
        c = testStr.charAt(i);
        ta.append(String.format("[%d]='%c' ", i, c));
        if (c == '!') {
            break;
        }
        str += c;
        i++;
    }
    ta.append(String.format("\nAfter while loop, str='%s'", str));
}
```

The use of `break` and `continue` may destroy the logic of your code, making it hard to understand. For example, a loop that begins with the condition: `while (i >= 0)` looks as though it should continue running just as long as `i` has any value that is 0 or more. So when I insert an additional test that breaks out of the loop `if (c == '!')` that test violates the condition that I specified at the start of the loop since `i` will have a value greater than 0 at this point so, according to the loop's test condition, the loop should continue to run. You may be able to rewrite your code to avoid using breaks. For example, you could place both tests in the loop header, like this:

```
    while ((i >= 0) && (c != '!'))
```

BREAKING OUT OF A FOR LOOP

In a previous example program (*MultidimensionalArrays*), you may recall that I placed one `for` loop inside another in order to process the elements in arrays nested inside another array. Now you might wonder what you would need to do if you wanted to break out of the inner `for` loop? Would the `break` exit *just* the inner loop or would it exit the outer loop too? Try it out. Here I add code to the inner loop that breaks when the value of `column` is 2:

```
for (row = 0; row < 3; row++) {
    printf( "--- row %d --- \n", row);
    for( column = 0; column < 5; column++ ) {
          printf("column[%d], value=%d\n", column, grid[row][column]);
          if (column == 2) {
                break;
          }
    }
}
```

This time, this is what I see when the program runs:

```
--- row 0 ---
column[0], value=1
column[1], value=2
column[2], value=3
--- row 1 ---
column[0], value=6
column[1], value=7
column[2], value=8
--- row 2 ---
column[0], value=11
column[1], value=12
column[2], value=13
--- row 3 : [16, 17, 18, 19, 20] ---
column[0] = 16
column[1] = 17
column[2] = 18
```

From this you can see that the *inner* loop breaks when `row` is 0 and `column` is 2 but the *outer* loop continues running and it breaks again when `row` is 1 and `column` is 2 – and so on. This shows that `break` causes an exit from the innermost loop only.

LABELLED BREAK

If you want a break from an inner loop to exit to the outer loop, you can do this using a labelled `break` statement. You do this by placing a label – which is a name followed by a colon such as `outerloop:` immediately before the start of the outer loop like this:

```
outerloop:
for (int row = 0; row < 4; row++) {
```

When you want to exit both the inner and the outer loop on a `break` you just add the label name (without the colon) after the `break` keyword like this:

```
break outerloop;
```

You can also have labelled `continue` statements like this:

```
continue outerloop;
```

The *LabelBreak* sample program demonstrates the effect of breaking to a label:

LabelBreak

```
int[][] grid = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10},
    {11, 12, 13, 14, 15},
    {16, 17, 18, 19, 20}
};

outerloop:
for (int row = 0; row < 4; row++) {
    System.out.printf("--- row %d : %s --- \n",
            row, Arrays.toString(grid[row]));
    for (int column = 0; column < 5; column++) {
         System.out.printf("column[%d] = %d\n", column,
                     grid[row][column]);
            if (column == 2) {
                break outerloop;
            }
    }
}
```

This displays the following:

```
--- row 0 : [1, 2, 3, 4, 5] ---
column[0] = 1
column[1] = 2
column[2] = 3
```

Previously the unlabelled `break` exited the *inner* `for` loop when the loop variable `column` equalled 2 and then continued executing the *outer* `for` loop – with the result that the first three items in each of the four nested arrays were displayed. But with a `break` that jumps to the labelled *outer* `for` loop, both loops exit when the `break` is encountered. As a result only the three elements from the first nested array are displayed. The labelled `break` then terminates *both* `for` loops so none of the other nested arrays is processed.

If you find all this hard to follow, that is not surprising. Breaking in this way is rarely the most elegant way of terminating a loop. Whenever possible, you should try to terminate a loop only when the loop *condition* is met – for example when the test in a `while` loop or the expression in the second part of a `for` loop (e.g. `column < 5`) evaluates to **false**. There may be some occasions when it is useful to break from a loop if some critical condition is met but, as a general principle, you should avoid breaking out of loops whenever possible.

Chapter 8 – Enumerated types, interfaces and scope

Sometimes your programs may need to work with a fixed set of related values – for example, a calendar program may need to deal with the seven days of the week, whereas a gambling program might deal with four suits of playing cards. In order to represent these values you could use integers from 0 to 6 for days or from 0 to 3 for suits of cards respectively. But often it would be clearer if you could use named identifiers instead of numbers. For example, consider this code:

```
if ((suit == 0) || (suit == 1)) {
      System.out.println("This card is a red suit");
}
```

If you uses named identifiers, the above code could be rewritten more clearly, like this:

```
if ((suit == Suits.HEARTS) || (suit == Suits.DIAMONDS)) {
      System.out.println("This card is a red suit");
}
```

ENUMS

There is a simple way to create series of identifying names such as Hearts, Diamonds, Spades and Clubs. You do so by creating an 'enumerated type' or `enum`. This is how I would define the four suits of a deck of cards:

```
public enum Suits {
    HEARTS, DIAMONDS, CLUBS, SPADES
};
```

Similarly, I could create an `enum` of the months of the year like this:

```
public enum Months {
    JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};
```

ENUMERATION CONSTANTS

The identifiers listed in an `enum` are actually *constants* because their values cannot be changed. In fact, an `enum` may sometimes be described as an 'enumeration constant'.

If you want to pass `enum` values to functions, be sure to declare the function argument appropriately. For example, here is a function that expects an argument, `suit`, of the `enum` type `Suits`:

```
public static void showsuit( Suits suit ) {
```

When calling this function, I must be sure to send to it one of the constants defined by the `Suits` enum. One way of doing that would be to pass the enum name, followed by a dot and the constant name, like this:

```
showsuit(Suits.HEARTS);
```

Alternatively I could declare a variable of the same type as the enum and assign to that variable one of the enum's constants. That variable could then be passed to the function, like this:

```
Suits card;
card = Suits.SPADES;
showsuit(card);
```

This is my example code in the *Enums.java* project:

Enums

```
public enum Suits {

    HEARTS, DIAMONDS, CLUBS, SPADES
};

public enum Months {

    JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};

public static void showsuit( Suits suit ) {
    if ((suit == Suits.HEARTS) || (suit == Suits.DIAMONDS)) {
        System.out.printf("This card is a %s which is a red suit.\n",
                                                   suit.toString() );
    }else{
            System.out.printf("This card is a %s which is a black suit.\n",
                                                    suit.toString() );
        }
    }


    public static void showmonth(Months month) {
      String msg = "";
      System.out.print(month.toString() + " -- ");
      switch (month) {
         case DEC:
             msg = "Christmas is coming! ";
         case JAN:
         case FEB:
             msg += "Baby, it's cold outside!";
             break;
         case MAR:
         case APR:
         case MAY:
             msg = "Spring is bursting out!";
             break;
         case JUN:
             msg = "June is my favourite month!";
             break;
         case JUL:
         case AUG:
             msg = "Time to go to the beach";
             break;
         case SEP:
         case OCT:
         case NOV:
             msg = "The leaves are falling";
             break;
         default:
              msg = "I don't know what month this is";
              break;
         }
      System.out.println(msg);
    }

    public static void main(String[] args) {
        Suits card;
        card = Suits.SPADES;
        showmonth(Months.DEC);
        showmonth(Months.JAN);
        showmonth(Months.FEB);
        showmonth(Months.APR);
        showmonth(Months.JUN);
        showmonth(Months.SEP);
        showmonth(Months.OCT);
        showsuit(Suits.CLUBS);
        showsuit(Suits.DIAMONDS);
        showsuit(Suits.HEARTS);
        showsuit(card);
    }
```

And this is what is displayed when the program runs:

```
DEC -- Christmas is coming! Baby, it's cold outside!
JAN -- Baby, it's cold outside!
FEB -- Baby, it's cold outside!
APR -- Spring is bursting out!
JUN -- June is my favourite month!
SEP -- The leaves are falling
OCT -- The leaves are falling
This card is a CLUBS which is a black suit.
This card is a DIAMONDS which is a red suit.
This card is a HEARTS which is a red suit.
This card is a SPADES which is a black suit.
```

Notice that I have used the `toString()` method to display the actual name of an `enum` constant!

For another example of `enum`s, take a look at the *Adventure2* project. This is a game with a simple user interface that provides buttons that can be clicked to move the player North, South, East and West. The code that handles these button-clicks calls the `movePlayerTo()` method of the `game` object. The `game` object is created from the `Game` class and it contains all the main elements of a game including the map of rooms and the player.

The important thing to notice is that when I call the `movePlayerTo()` method I provide an argument such as `Direction.NORTH` or `Direction.SOUTH`. This makes it easy to understand my code – giving a descriptive direction is much clearer than simply using an integer value, for example. Here `Direction` is an `enum` which is defined in the *Direction.java* code file:

Adventure2

```
public enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST;

    public static final int NOEXIT = -1;
};
```

The `movePlayerTo()` method in the *Game.java* code file calls the `MoveTo()` method, passing to it the `player` object, representing the game-player, and the `Direction enum` constant. I have deliberately made `moveTo()` able to move any interactive character (in my program, an interactive character is an instance of the `Actor` class) since later on I may want to add other characters which, just like the player of the game, can move around the map and collect treasures. The `moveTo()` method tries to match the value of the `Direction` parameter using a `switch` statement:

```
int moveTo(Actor anActor, Direction dir) {
    Room r = anActor.getRoom();
    int exit;

    switch (dir) {
        case NORTH:
            exit = r.getN();
            break;
        case SOUTH:
            exit = r.getS();
            break;
        case EAST:
            exit = r.getE();
            break;
        case WEST:
            exit = r.getW();
            break;
        default:
            exit = Direction.NOEXIT;
            break;
    }
    if (exit != Direction.NOEXIT) {
        moveActorTo(anActor, (Room) map.get(exit));
    }
    return exit;
}
```

The advantage of using the `Direction enum` here is clarity. When I use named identifiers such as `EAST` and `WEST` it is obvious that I am trying to move the player in the specified compass direction. The code would have worked just as well if I had used integer values instead. But it would not have been as clear.

You will also notice that I have defined `Direction.NOEXIT`. This is used not only here but also when I create the `map` in the `Game()` constructor. In this project the `map` is a simple `ArrayList` and each room in that list occupies a fixed position. The exits from one room to another are simply integers which indicate the map index of the room at any given exit:

```
map.add(new Room(
    "room0", "You are standing in an open field west of a white house",
    Direction.NOEXIT, 2, Direction.NOEXIT, 1));
map.add(new Room(
    "room1", "White House",
    Direction.NOEXIT, Direction.NOEXIT, 0, Direction.NOEXIT));
map.add(new Room(
    "room2", "Gold room",
    0, 4, Direction.NOEXIT, 3));
map.add(new Room(
    "room3", "Troll room",
    Direction.NOEXIT, 5, 2, Direction.NOEXIT));
map.add(new Room(
    "room4", "Dark cave",
    2, Direction.NOEXIT, Direction.NOEXIT, 5));
map.add(new Room(
    "room5", "Troll room",
    3, Direction.NOEXIT, 4, Direction.NOEXIT));
```

Let's take this `Room` object as an example:

```
new Room("room2", "Gold room", 0, 4, Direction.NOEXIT, 3)
```

This room has a name, "room2", a description, "Gold room" and then four exits. The North exit leads, in theory, to the room at index 0 in the map (that is the room with the name "room0"), the South exit leads to the room at index 4, the West exit is marked by the constant `Direction.NOEXIT` to indicate that there is no room in that direction, and the East exit leads to the room at index 3 in the map.

To understand `Direction.NOEXIT`, look back at the definition of the `Direction enum`. The `NOEXIT` constant is declared rather differently from all the others: `NORTH`, `SOUTH`, `EAST` and `WEST` are just entered as items in a comma-separated list with no associated data-types or values. But `NOEXIT` is declared as a `static final int` (that is, an integer constant that 'belongs' to the `Direction` class):

```
public static final int NOEXIT = -1;
```

REMINDER: Constants were explained in Chapter 4.

I can't use the identifiers `NORTH`, `SOUTH`, `WEST` and `EAST` to initialize exits when creating `Room` objects because the exit fields in those objects are all integers. But I can use the `NOEXIT` constant to initialize one of these fields as I have specifically assigned to that constant the `int` value -1. I've chosen -1 because I can be sure that no room in the map will have that index.

If you have used `enum`s in other programming languages, you may be surprised to find that static or 'class' constants can be created in the body of an `enum`. In most languages `enum`s are not classes. In Java they are.

The Enum class

In many programming languages, `enum`s are simply collections of names – constants that may, or may not, have some associated value such as an integer. For example, this is how I might declare an `enum` in the C language:

```
enum directions {
    NORTH, SOUTH, EAST, WEST
};
```

By default, C assigns integer values to `enum` constants. Here `NORTH` would be assigned 0, `SOUTH` would be assigned 1 and so on. At first sight, a Java `enum` may look pretty much the same as a C `enum`. But in fact, when you create an `enum` in Java you are actually defining a class that is the descendent of the `Enum` class. Your `enum` therefore (just like a regular class) is able to contain specifically typed data items such as the `NOEXIT` constant which I added to my `Direction enum`. In fact, a Java `enum` can even contain methods. The Oracle Java tutorial site provides an example of quite a complex `enum` called `Planet` which has its own constructor, methods and variables. While this kind of `enum` may have a use in some special cases, many programmers (myself included!) prefer to stick with the traditional, simple type of `enum`, broadly similar to those in C, which just contains a list of constants to assist in coding clarity.

For more on the `Enum` class, see the Oracle Java tutorial:
https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
Also refer to the JDK entry on `Enum`:
https://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html

Interfaces

We first came across interfaces in Chapter 6. You may recall that I described an interface as being a sort of abstract class: it defines the methods that a class must implement but it does not contain the actual code of those methods. (But be careful – Java also lets you define `abstract` classes which, unlike interfaces, are *extended* rather than *implemented*. See the section later in this chapter describing abstract classes). Another way of describing an interface would be as a 'contract' or an agreement on the behaviour required by any class that implements that interface. This is how Oracle describes an interface in its Java documentation:

"There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a 'contract' that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts."

https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

A Java `ArrayList` is declared like this:

```
public class ArrayList<E extends Object> extends AbstractList<E>
      implements List<E>, RandomAccess, Cloneable, Serializable
```

Each Java class can only descend from one immediate parent class – here `ArrayList` descends from or 'extends' the `AbstractList` class. When a class descends from another class it automatically inherits the behaviour that was defined by its ancestor class's methods.

The `ArrayList` class also 'implements' `List`, `RandomAccess`, `Cloneable` and `Serializable`. But these are not classes. They are interfaces. A class can extend only one other class but it can implement many interfaces. An interface is like a definition of a class without executable code. Each interface may define a set of constants and methods but it does not provide the behaviour – the code – of those methods.

This means that whenever a class implements an interface it is obliged to provide methods matching all the methods that the interface defines. For example, since the `ArrayList` class implements the `List` interface, and the `List` interface declares the `size()`, `add()` and `remove()` methods, the `ArrayList` class is obliged to provide its own `size()`, `add()` and `remove()` methods. If it fails to do so, it will be an error. What's more, every other class that implements the `List` interface must also provide their own `size()`, `add()` and `remove()` methods. Each class may implement those methods in different ways. For example a `TreasureList` class and a `BeerList` class might both implement the `List` interface. The `TreasureList` class might implement `size()` to return the number of treasures in a game whereas the `BeerList` class implementation of `size()` might return the number of gallons stocked by a brewery. As long as both those classes implement `size()` to return an `int` value as required by the `List` interface, they may implement the behavior of the method in any way they like.

A CUSTOM INTERFACE

Let's see how you might go about writing your own interface. Let's suppose that you are leading a team of programmers writing a game. The game may define a variety of classes that have some sort of value. These may be anything from a `Treasure` to a `Score`.

The trouble is the `Treasure` and `Score` classes are not related through inheritance. `Treasure` is a descendent of the `Thing` class whereas `Score` is a descendent of the `GameStatistics` class. In other words, `Treasure` and `Score` have nothing in common apart from the fact that they store a value and they need to have methods that can add to or subtract from that value. One way of dealing with this would be to write an interface that defines the methods needed to manipulate values and then make both the `Treasure` and `Score` classes implement that interface.

In my *Interfaces* sample program, I define this interface in the *ValueableItem.java* code file:

Interfaces

```
package myinterfaces;

public interface ValuableItem {

    public int value();

    public void addValue(int aValue);

    public void deductValue(int aValue);

}
```

I create a `Treasure` class that inherits the features of its ancestor `Thing` class and implements the methods declared by the `ValuableItem` interface:

```
public class Treasure extends Thing implements ValuableItem {

private String description;
private int treasureValue;

public Treasure(String aName, String aDescription, int aValue) {
    // constructor
    super(aName);
    this.description = aDescription;
    this.treasureValue = aValue;
}

public String getDescription() {
    return description;
}
```

```
public void setDescription(String aDescription) {
    this.description = aDescription;
}

@Override
public int value() {
    return treasureValue;
}

@Override
public void addValue(int aValue) {
    treasureValue += aValue;
}

@Override
public void deductValue(int aValue) {
    treasureValue -= aValue;
}
```

The `@Override` annotations here are optional but recommended. They provide information to the compiler to tell it that the methods that follow the annotations have been previously declared. The compiler can then compare the method 'signature' – its name and arguments – with the previous declaration, to make sure that they match.

In my sample program I have not written a `Score` class. If you'd like to experiment with implementing my interface, you could try adding that class yourself.

ABSTRACT CLASSES

In addition to interfaces, Java also provides abstract classes and methods. An abstract class is a class that cannot be instantiated but may be subclassed. An abstract class is a class that is declared using the `abstract` keyword like this:

```
abstract class MyClass{}
```

An abstract class may optionally contain abstract methods. An abstract method is a method that is declared using the `abstract` keyword but without an implementation like this:

```
abstract void myMethod(int someValue);
```

Abstract classes may be thought of as 'prototypes'. They define the structure of a class but not its implementation. An abstract class can be subclassed (*extended* rather than *implemented*) and the behaviour of the abstract class's methods can then be programmed in the subclass. Refer to Oracle's documentation for more information:
https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

SCOPE

Variables in your Java programs are all 'visible' within certain limits. For example, a variable declared *inside a class* but *outside a method* is visible to (in other words, it can be referenced by) all the methods in that class.

```
public class MyClass{
    int x = 10;

    private void aMethod() {
          // x is visible here
     }
}
```

But a variable defined *inside a method* is only visible within that method:

```
public class MyClass{
    int x = 10;

    private void aMethod() {
          int y = 100;
          // x and y are visible here
     }

    private void anotherMethod() {
          // x is visible here but y is not visible
     }
}
```

The visibility of a variable is called its *scope*. Variables declared inside methods are said to be 'local' to that method. A named parameter is also local to a method. In other words, it has 'local scope':

```
private void aMethod(int y) {
    // y is visible here
}

private void anotherMethod() {
    // y is not visible here
}
```

When an identifier occurs in two different levels of scope, the one in the narrower (more 'local' scope) will be used. Look at this example:

```
int x = 10;
private void methodA() {
    // here x has the value 10
    outputTA.append("\n x = " + x);
}

private void methodB() {
    int x = 100;
    // here x has the value 100
    outputTA.append("\n x = " + x);
}

private void methodC() {
    methodA();
    methodB();
    outputTA.append("\n x = " + x);
    // here x has the value 10
}
```

Here `methodC()` calls `methodA()`. The only variable called `x` that is in the scope of `methodA()` is the one declared outside that method which has the value 10. So when the value of `x` is appended to the text in the box, `outputTA`, 10 is displayed. Now `methodC()` calls `methodB()`. This method declares a local variable called `x` which is assigned the value 100. The local variable takes precedence over the class-level variable so the value displayed now is 100. It is important to note that even though the local variable `x` has the same name as the class-level variable `x` (the one that is declared outside the methods), these are two different variables. The local variable pops into existence when the method is entered and it pops out of existence again when the method is exited. Its value does not affect the value of the variable with the same name outside that method. So even though the local variable `x` was assigned 100, the value of the class-level variable declared outside the method is unchanged.

GLOBAL AND LOCAL VARIABLES

In the preceding example, the scope of the variable `x` within `methodA()` is 'local' to the method itself. It is a *local* variable with *local* scope. The scope of the variable `x` declared outside that method is the class in which it is declared. You could say its scope is local to that class. However, you will often hear programmers say that it is a 'global' variable – meaning that its scope is widely or 'globally' visible to all the methods in the class. The precise meaning of 'global' scope in programming varies from language to language (a truly global variable would be visible to all the code everywhere in your program). I prefer, therefore, to refer to 'class-level variables' to describe variables whose scope is a single class.

ACCESS MODIFIERS

You can restrict the visibility of methods by declaring them using the keywords `private`, `public` or `protected`. When a method is *private* it can be accessed only from other methods declared in the same class. This is useful if you want to make sure that code that uses the class or objects created from that class cannot directly access those methods. Here is an example of a *private* method:

Adventure2

```
private int moveTo(Actor anActor, Direction dir) {
    int exit;
          // more code here
    return exit;
}
```

If a method is declared as *public* then it can be accessed from code outside its class. This is useful when you want to be able to invoke methods from elsewhere in your program. The methods we call upon objects using a dot after the variable name are public methods. This is a *public* method:

```
public int movePlayerTo(Direction dir) {
    return moveTo(player, dir);
}
```

The two methods above (`moveTo()` and `movePlayerTo()`) are both declared in the `Game` class of my adventure game project. Given a `Game` object called `game` I can call the *public* method like this:

```
game.movePlayerTo(Direction.WEST);
```

I cannot call the *private* method, however:

```
game.moveTo(player, Direction.WEST); // Error! Can’t call private method
```

Finally, there may be some cases in which you want to make variables and methods available both to the current class and to its subclasses but not to any other code. In that case you may use the `protected` keyword when declaring members of a class. A protected member is visible to all classes in the same package as well as to subclasses of the current class in other packages.

You can also limit the visibility of an entire class. If a class is declared as `public` it is visible everywhere. If it has no modifier it is visible only to classes in the same package; that is, it will be ‘package private’. If you omit an access modifier for a class ‘member’ (a variable, constant or method) it will be ‘package private’.

See Oracle’s documentation on *Controlling Access to Members of a Class* for more on access modifiers:
https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Chapter 9 – Generics and Exceptions

You will often need to perform similar operations on different types of objects in your Java programs. Maybe you want to create lists of objects and have the ability to add or delete items from that list. We've done this sort of operation many times in previous projects. For example, you might declare an `ArrayList`, `aList`, and then add items to it like this:

Generics

```
ArrayList aList = new ArrayList();
aList.add("one");
aList.add(2);
aList.add("three");
```

The basics of generic lists were described in Chapter 6. But consider what happens if you now want to write a method that performs some action on the items in that list. Let's suppose, for example, that you write a method to return the last item:

```
private String getLastItem(ArrayList aList){
    return (String) aList.get(aList.size()-1);
}
```

This method assumes that the last item is a string so the method's return type is declared to be `String`. But since the items in an `ArrayList` can be of *any* type, the item must be 'cast' to a String by preceding it with the `String` type name between parentheses (this was explained when we discussed Type Casts in Chapter 3):

```
return (String) aList.get(aList.size()-1);
```

So now all is well and good just so long as the final element of `aList` is indeed a string. But let's suppose I add on an integer:

```
aList.add(4);
```

And now I call my method to return the final element in the list:

```
outputTA.append(getLastItem(aList) + "\n");
```

The `getLastItem()` method expects this element to be a string. So what happens when it actually finds an integer? The program compiles without any errors. However, when you click a button that runs the code that tries to return a string from the end of `aList`, you will see a nasty-looking error message:

```
Exception in thread "AWT-EventQueue-0" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String
```

This should be no surprise. We cannot simply 'cast' an integer to a string. The two objects are incompatible. But, that being so, you may wonder why we were able to compile the program in the first case since it contained this obvious error. It turns out that the compiler allowed this because an `ArrayList` is capable of holding objects of any type and it is entirely legitimate for our method to cast an object to a specific class such as String. So, as far as the compiler is concerned, all the code is valid. In fact, if we never click the button that runs the code that calls the `getLastItem()` method with a list containing a final integer, the code will all work correctly. This means that we have to take extra care when writing code to handle the elements in an `ArrayList` to make sure that we do not accidentally try to manipulate an element in some inappropriate way.

STRONGLY-TYPED LISTS

One way of avoiding this problem would be to insist that the `ArrayList` stores elements only of a single type. You might think it would be sufficient just to make a note to remind yourself that `aList` should only be used to store strings. But what happens if, at some later date in a complicated programming project, you forget that this was what you intended and you write some code that adds an integer? As long as that integer is at any position other than the last in your list, your code continues to work fine so you don't realise that it contains a bug. And then one day, something in your code happens to put an integer at the final position and the error occurs. If you are working with a programming team, the chances that someone might accidentally add the wrong sort of data to your `ArrayList` are even greater. One way to make sure that this cannot happens would be to make the list strongly-typed – that is, to insist that only strings can be added to it. In Java, this is how you would declare an `ArrayList` that can only hold strings:

```
ArrayList <String>gStrList = new ArrayList<>();
```

Now, with `gStrList` typed to hold String elements the compiler can spot immediately if you attempt to add some other type. For example, this code will not compile:

```
gStrList.add(2);
```

The compiler produces this message:

```
error: no suitable method found for add(int)
        gStrList.add(2);
method Collection.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
```

I am now obliged to fix my code, thereby removing any possibility that the error will occur when the code is run. This is good news for me as a developer. It forces me to avoid errors that might otherwise occur in a running program. My strongly-typed `ArrayList` is a simple example of Java’s *generics*.

GENERICS

This is how Oracle defines 'generics':

> **Generics** enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

What this means is that you are able to define classes and methods that can work with some non-specified type of object. That is they are common or 'generic' to all object types. The actual type of the object can be specified when those classes and types are used in your code. The `ArrayList` class is generic. It is declared as being a collection of elements, each of which has the type `E`:

```
public class ArrayList<E>
```

Here `<E>` is not a real type – it is a place-holder that will be replaced with an actual type when you create an instance of `ArrayList`. You simply place the actual type between pointy brackets and also place an empty pair of pointy brackets before the parentheses of the constructor. So this is how I create an `ArrayList` to hold string elements:

```
ArrayList <String>gStrList = new ArrayList<>();
```

I could just as easily create an `ArrayList` to contain integers, like this:

```
ArrayList <Integer>gIntList = new ArrayList<>();
```

GENERIC KEY-VALUE PAIRS

Java defines a number of generic collections. You may recall from Chapter 6 that we created a `HashMap` object to hold strictly-typed *Key-Value* pairs in which the Key was obliged to be a `String` and the Value was an instance of my `Room` class:

```
HashMap <String, Room>map;
map = new HashMap<>();
```

I was able to specify types for the `HashMap` keys and values due to the fact that the `HashMap` class is generic and, just as `ArrayList` has the `<E>` placeholder, a `HashMap` has a pair of placeholders `<K, V>` that can be replaced by actual types when a `HashMap` object is declared and created in your own code.

GENERIC CLASSES

You can write your own generic classes which, just like the generic `ArrayList` class, can be used to handle objects of different types. This is how I would declare a simple generic class:

```
public class MyGenericClass<T> {}
```

Here the `<T>` is the 'type parameter'. It provides a placeholder for a real type so that this type can then be used inside the class. It is possible to have more than one type parameter `<T1, T2>` and so on. But for most purposes one should be enough. This is how I declare a private variable `someData` of the type `T`:

```
private T someData;
```

If I want to provide a constructor to initialize this variable, I do so by declaring the input parameter as being of type T:

```
public  MyGenericClass( T aValue ){
    someData = aValue;
}
```

Just as with a normal class, I can provide getter and setter methods to assign or return the values of private variables, once again being sure to enter the type as `T`:

```
public T getSomeData() {
    return someData;
}

public void setSomeData(T someData) {
    this.someData = someData;
}
```

In my example, I've also written one other method that returns a string description of a `MyGenericClass` object and its private data:

```
public String describe() {
    return this.getClass() + " <" + someData.getClass() + ">  " +
           someData.toString() +"\n";
}
```

When I create an object from my generic class I have to supply an actual type (between a pair of pointy brackets) to replace the `T` parameter, like this:

```
MyGenericClass<String> myStrOb;
```

In fact, since my class has its own constructor, with a parameter given by the same type `T`, I can pass a piece of data of the same type specified for my object (here, that's a String) when I create that object:

```
MyGenericClass<String> myStrOb = new MyGenericClass<>("Hello world");
```

I could equally create objects to work with other types such as `Integer`:

```
MyGenericClass<Integer> myIntegerOb = new MyGenericClass<>(10);
```

And now I can calls the `describe()` method of my generic class as well as its getter and setter methods so that the same methods operate on different data types:

```
outputTA.append(myStrOb.describe());
outputTA.append(myIntegerOb.describe());
myIntegerOb.setSomeData(500);
outputTA.append(myIntegerOb.describe());
```

And this code displays:

```
class MyGenericClass <class java.lang.String>  Hello world
class MyGenericClass <class java.lang.Integer>  10
class MyGenericClass <class java.lang.Integer>  500
```

You may choose your own parameter names (such as `<T>` or `<E>`) when defining generics. However, by convention, these are the names typically used:

T : Type
E : Element (in a collection)
K : Key in a Map/Hash
V : Value in a Map/Hash
N : Number

Generic Lists and Collections

Commonly, classes that manage collections or lists may be made generic. This is because lists of all types of object may need to perform similar functions on the elements of the list – for example, to add and remove elements or to find an element at a given index. Implementing the same methods time after time for each class that defines some strongly-typed list (a `ListOfString` class and a `ListOfInteger` class, for example) would clearly be highly repetitive. It would be better to implement a single generic class with all the required methods for manipulating list elements *in general* and then specify the *actual* element type whenever an instance of that generic class is created. You can either implement your list from the ground up or you can extend an existing generic collection type and add any additional features that you need. In my *Generics* sample program I have extended `ArrayList` and added on one new method to return the last element of a list:

```
public class MyList<E> extends ArrayList<E> {

    public E getLastItem() {
        return this.get(this.size() - 1);
    }
}
```

This time I have followed the Java convention of naming the generic parameter `E` to represent the type of the elements in the list. `MyList<E>` extends `ArrayList<E>` since both handle elements of the type given by `E`. The method `getLastItem()` returns the last item – whose type is once again represented by the generic type parameter `E`. The index of the last item is calculated by subtracting 1 from the list's size.

When I used a standard `ArrayList` to hold elements, I was obliged to pass that `ArrayList` as an argument to a method such as `getLastItem(ArrayList aList)` which returned a specific element type such as String. But with the generic `MyList` class, the `getLastItem()` method acts *upon the list itself* and it is able to operate on Strings, Integers or any other specified object type. You can see examples of this in the code inside the *NewJFrame* class of the *Generics* project. First I create two `MyList` objects, one typed to hold strings and the other typed to hold integers:

```
MyList<String>sList = new MyList<>();
MyList<Integer>iList = new MyList<>();
```

Now I can create a list of strings and get and display the last string element (note that `sList` does not allow me to add an integer as the list has been typed for strings only, so I have had to comment out the line that tried to add the integer 2):

```
sList.add("one");
    // sList.add(2);
sList.add("two");
sList.add("three");
outputTA.append(sList.getLastItem() + "\n");
```

Similarly I can create a list of integers, then get and display the last integer element (this time `iList` does not allow me to add a string as the list has been typed for integers only, so I have commented out the line that tries to add the string "two"):

```
iList.add(1);
    // sList.add("two");
iList.add(2);
iList.add(3);
outputTA.append(iList.getLastItem().toString() + "\n");
```

NOTE: In the above code, the final call to the `.toString()` method is not really needed as the `Integer` object will be automatically converted to its string representation during string concatenation (since I use the `+` operator to add `"\n"`.). But it seems rather haphazard coding practice to rely on this behaviour – which would, in any case, change if I were to remove the concatenation operator – and that is why I have explicitly used the `.toString()` method here for clarity. For more on this, see Automatic String Conversions in Chapter 3.

Overriding Methods

Sometimes you may want a method in a subclass to replace a method with the same name in its ancestor class. That is called 'method overriding'. For example, if you have a `Troll` class that extends the `Monster` class you might want the `Troll`'s `saySomething()` method to produce a different noise (a string) from the `Monster`'s `saySomething()` method. This is a simple example of how these two classes would implement that method:

Overriding

```
public class Monster {
    public String saySomething() {
        return "Grrrrr\n";
    }
}

public class Troll extends Monster{
    public String saySomething() {
        return "Ugh, ugh, ugh!\n";
    }
}
```

Now if let's suppose I create a `Monster` object and a `Troll` object, I call the `saySomething()` method on each and then I display the returned string in a text area, `outputTA`:

```
outputTA.append(myMonster.saySomething());
outputTA.append(myTroll.saySomething());
```

This is what will be displayed:

```
Grrrrr
Ugh, ugh, ugh!
```

When I call `saySomething()` on the `Troll` object, its own *overridden* method is executed rather than the method with the same name in its ancestor class. But what happens if I made a mistake when I wrote that method and accidentally named the method `saysomething()` instead of `saySomething()`? As we know, Java is a case-sensitive language so the fact that I have accidentally used a lowercase 's' in the `Troll`'s method is important. Java treats `saysomething()` as a different name from `saySomething()`. So now when I ruin my code, this is what I see:

```
Grrrrr
Grrrrr
```

The code is still valid because when I call `myTroll.saySomething()` Java looks for a method with that name and finds it in the ancestor class `Monster`. But, of course, that wasn't my intention. My intention was that the method I wrote in the `Troll` class should be used – that it should *override* the method with the same name in the `Monster` class. My silly typing error (using the lowercase 's') has caused a bug which, in a more complicated program, might be very hard for me to spot.

There is a simple trick to help me avoid this sort of problem. When I intend one method to override another, I can place the `@Override` annotation before that method. In Java an annotation is an instruction to the compiler. The `@Override` annotation tells it to check that a method has successfully overridden a method with the same name in its ancestor class. You may recall that I used `@Override` in Chapter 8 when implementing interfaces, to check that the named methods were declared by an implemented interface. In the current example, if I can place `@Override` above the method in my `Troll` class like this:

```
@Override
public String saysomething() {
    return "Ugh, ugh, ugh!\n";
}
```

Now the compiler will compare this method with the methods in the ancestor class. It will discover that there is no method with this name so the method cannot be overridden and an error occurs. In fact, your IDE may also perform this check and it too may flag an error. Now I can fix this by capitalising the 'S' to name the method `saySomething()` and the error goes away.

METHOD OVERLOADING

Don't confuse method overriding with method overloading. We first encountered overloaded methods back in *Chapter 4* when I mentioned that some Java classes have multiple methods with the same name but with different signatures.

METHOD SIGNATURE

A method signature is the complete method declaration including both its name and its parameters.

For example, the `String` class has several methods called `indexOf()`, each version having a different set of parameters. The fact that the method signatures are different means that Java is able to identify them as different methods, so that `indexOf(String str)` is different from `indexOf(String str, int fromIndex)`, for example. It is also permitted for a single class to have more than one constructor. In fact, if you look at the documentation for the `String` class, you will see that it has numerous constructors, each of which defines a different list of arguments so that each constructor has a unique *signature*. When two methods in the same class have the same name but different signatures they are said to be *overloaded*.

This is an example of an overloaded method:

```
public class Troll extends Monster{

    public String saySomething() {
        return "Ugh, ugh, ugh!\n";
    }

    public String saySomething(String thingToSay) {
        return "Ugh, ugh, " + thingToSay +"!\n";
    }
}
```

Now when I call the `saySomething()` method, the first of the two methods with that name will be executed if no argument is passed to it and the second will be used if I pass a string argument:

```
outputTA.append(myTroll.saySomething());
outputTA.append(myTroll.saySomething("I want to be fed"));
```

This displays:

```
Ugh, ugh, ugh!
Ugh, ugh, I want to be fed!
```

EXCEPTIONS

There is one important little problem I have not yet considered in any depth in this course. Namely: *bugs*! Few programs of any ambition are totally bug-free. What is more, even if your programs work when *you* use them, there is always the risk that they will go wrong when *someone else* uses them; because other people may not do all the things your expect them to do. For example, in a calculator program, you may expect the user to enter a number such as 10 but in fact someone enters a string such as "ten".

It is the programmer's responsibility to assume that the end user will, at some time or another, do the most stupid things possible – things that could cause your program to produce incorrect results or even to crash completely. It is in your interests, therefore, to build into your code some means of recovering from potential disaster.

For an example of this, consider the **TaxCalc** project from *Chapter 3*. This prompted the user to enter a subtotal and it then used this value to calculate the tax due and the grand total (the subtotal plus tax). But what would happen if the end user entered some text that could not be converted to a floating-point number by the `parseDouble()` method here?

Step03/TaxCalc

```
subtotal = Double.parseDouble(subtotalTF.getText());
```

This is what would happen: a `NumberFormatException` would occur and you would see an error message like this…

```
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException:
For input string: "ten"
  at
sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043
)
  at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
  at java.lang.Double.parseDouble(Double.java:538)
```

An exception is an object. When an error happens, an instance of one of Java's Exception classes, such as `NumberFormatException`, is created. Your code can make use of this by trapping the exception object and accessing its properties and methods. Load and run the **SafeTaxCalc** project for *Step09*.

Enter 'IO' (letters I and O, not the numbers 1 and 0) into the subtotal edit box at the top of the form. Now click the button. An error message pops up telling you both the class of the exception (`NumberFormatException`) and the cause ('For input string "IO"'). The user can now close the dialog and fix the error by entering a valid number into the text box. Clearly this is more civilised behaviour than when the exception went unhandled as before. Let's look at how I have handled this exception in my code. First I create a constant and three `double` variables, plus one Boolean variable which I'll explain shortly:

SafeTaxCalc

```
final double TAXRATE = 0.2;
double subtotal = 0.0;
double tax = 0.0;
double grandtotal = 0.0;
boolean no_error = true;
```

Now, between a pair of curly brackets preceded by the keyword `try`, I put the bit of code that I know may cause an error if the user enters an incorrect value:

```
try {
    subtotal = Double.parseDouble(subtotalTF.getText());
}
```

This tells Java to try to perform this operation – that is, to convert the text from the `subtotalTF` text field to a `double`. But if it fails – if an error occurs – an exception object will be created. In order to gain access to this exception object I can use the `catch` keyword followed by an exception parameter between parentheses and some code between curly brackets which will execute if an exception object is caught:

```
catch (Exception e) {
    JOptionPane.showMessageDialog(this,
            "--- Oops! Could not convert input to double! ---"
            + "\nError class: " + e.getClass()
            + "\nError message: " + e.getMessage() );
    no_error = false;
}
```

An exception object provides access to a number od methods and properties and I have used two methods here, `getClass()` and `getMessage()`, to display more information. If an exception occurs I set the Boolean variable `no_error` to false so that I can skip the rest of the code by making this simple test:

```
if (no_error)
```

Exception Types

It is also possible to catch specific *types* of exception. Here is an example:

Exceptions

```
int i = 10;
ta.setText("");
try {
    i = 10 / Integer.valueOf(textField1.getText());
    ta.append("Result = " + i);
} catch (ArithmeticException e) {
    ta.append("Oops! Can't divide by " + textField1.getText());
    i = 0;
} catch (NumberFormatException e) {
    ta.append(textField1.getText() + " is not a valid integer!");
    i = 0;
} catch (Exception e) {
    ta.append("ERROR: " + e.getClass() + " '" + e.getMessage() + "'");
    i = 0;
}
```

This code first tries to catch an `ArithmeticException`. This occurs when, for example, the user enters 0. It is not possible to divide a number by zero so the `ArithmeticException` is thrown. At that point all the rest of the `catch` blocks are skipped. But even if an `ArithmeticException` is not thrown, that doesn't mean that some other type of error might not occur. Let's suppose, for example, that the user enters some text such as "xxx". This text cannot be converted to an integer, and when an attempt is made a `NumberFormatException` is thrown. In that case, the second `catch` block executes.

I've also added a very generic block right at the end which will deal with any `Exception` object that has not already been caught. If a specific exception such as `ArithmeticException` or `NumberFormatException` has already been caught, this final block will be skipped. If some other sort of exception occurs, however (one that I haven't even thought of!) then this block will deal with it. Naturally, if no exception is thrown then none of the `catch` blocks will execute.

FINALLY

You may also optionally provide a `finally` section enclosing code that you want to be executed whether or not an exception occurs. A `finally` block is often used to 'clean up' some sort of resources. Oracle provides this example of a `finally` block which closes a `PrintWriter` object (a class that writes 'streams' of data – in this case, by saving text to a file on disk) whether or not an error occurred:

WriteList

```
try {
    System.out.println("Entering" + " try statement");
    out = new PrintWriter(new FileWriter(fileName));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
} catch (IndexOutOfBoundsException e) {
    System.err.println("Caught IndexOutOfBoundsException: "
            + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
} finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

For a more in-depth look at Exceptions, refer to Oracle's online tutorial: http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

We will look more closely at files and streams in the next chapter.

Chapter 10 – Files and Serialization

File-handling is one of the fundamental skills of programming. Whether you are programming a spreadsheet, a word processor or a game, you will, at some stage, need to read data from one place and write it to another. In this chapter we shall look at some ways of reading and writing data to and from files on disk.

There are, in fact, many ways of writing and reading data to and from files using Java. In the *WriteList* project from Chapter 9 we used a `PrintWriter` object to save lines of text to a file on disk. `PrintWriter` is one of many special-purposes classes that write text into 'streams'. A stream can be thought of as a channel that conducts data between a source and a destination – for example from memory to disk or vice versa. Streams can be used to save data either in the form or linear sequences of bytes or as well-defined data-structures representing objects of various sorts. I'll explain how to save and load complex objects shortly.

First though I want to look at random access files. These are structured files that that let us locate data at precise positions. In older programming languages, random access files provided the most common way of storing records of fixed sizes – such as the records in a database.

Random Access files

We first came across the `RandomAccessFile` class in Chapter 7. You may recall that the *ReadFile* sample program in that chapter created a `RandomAccessFile` object, `file`, and used the `readLine()` method to read lines of text from that file.

Step07/ReadFile

```
RandomAccessFile file = null;
file = new RandomAccessFile(pathName, "r");
int linecount = 0;
String line = "";
while ((line = file.readLine()) != null) {
    linecount++;
     ta.append(String.format("[%d] %s\n", linecount, line));
}
```

I've slightly updated this program in the `RAFile` project for Step 10. This new version illustrates one of the key features of random-access files: the ability to find data by 'seeking' (moving the file–pointer) to specific locations. If you seek to 0 that will move the file pointer to the start of the file. If you have saved chunks of data or 'records' each of a fixed size, you can locate the start of a specific chunk of data by seeking to an index multiplied by the size of the record. For example, if you have saved a set of records from an employee database, and the size of each record is given by the constant `RECSIZE` (which, for the sake of argument, we'll suppose has the value 500 to show that each record takes up 500 bytes), you could seek to the record at index 5 like this:

```
file.seek( 5 * RECSIZE );
```

In my example project, I am working with a plain text file – a file of characters, and each character has the size of one byte – so I can seek to a specific character position, like this:

RAFile

```
file.seek(60);
```

Now, from position 60, I can read in a specific number of bytes. I've create a 10-byte array like this:

```
byte[] line = new byte[10];
```

And I read bytes into that array like this:

```
bytesread = file.read(line);
```

Here `bytesread` is an `int` variable that is assigned the number of bytes that were read into the buffer. If I want to continue reading from where I left off. I can just add on the number of bytes that were read in previously to the previous offset (the starting position) and then move the file pointer to this new offset:

```
file.seek(60+bytesread);
```

If there is no data to read then the value returned by the `read()` method will be -1 and I can test this in my code. For example, there are fewer than 1000 bytes in my short file so if I attempt to read data from position 1000 the operation fails and -1 is returned. This code handles that:

```
file.seek(1000);
bytesread = file.read(line);
if (bytesread != -1) {
     s = new String(line);
     ta.append("\n" + s);
} else {
    ta.append("\nEnd Of File!!! -- No more data to read! ");
}
```

When I've finished with the file, I close it:

```
file.close();
```

An alternative way of handling random access files in Java, using the `SeekableByteChannel` interface – which is found in the `nio` ('New IO') package – can be found in Oracle's online tutorial:
http://docs.oracle.com/javase/tutorial/essential/io/rafs.html

STREAMS

Most of Java's input/output tasks are handled by streams. A stream is nothing more than a flow of bytes from one place to another. It doesn't have to be to and from disk. It could be to and from some remote location on the Internet, for example. For our purposes, however, we'll restrict ourselves to streams of data flowing between a computer's memory and a disk.

Java defines two important streams for disk IO operations - `FileInputStream` and `FileOutputStream`, each of which takes a string parameter representing a file name. Once a `FileInputStream` has been created, it can be passed for processing to streams of other types. In my sample project, *FileInputStr*, I make use of two stream classes, `DataInputStream` and `DataOutputStream`, which can be used to read and write mixed primitive data types within a single stream. I have declared two arrays each containing four data items, one of `int` and one of `double`:

FileInputStr

```
int numbers[];
double dblnumbers[];
```

These arrays are initialized in the constructor:

```
this.numbers = new int[]{10, 20, 30, 40};
this.dblnumbers = new double[]{1.1, 2.2, 3.3, 4.4};
```

Now find the `saveBtnActionPerformed` method. This starts off by creating a `FileOutputStream` to write data to a file named "test.dat. It passes this stream for further processing to a DataOutputStream object which I have called dos. This is the code:

```
DataOutputStream dos = new DataOutputStream(
                            new FileOutputStream("test.dat") );
```

A for loop then counts through the numbers array and writes each integer to the DataOutputStream stream using the writeInt() method. Next it writes each double to the same stream using the writeDouble() method:

```
for (int i = 0; i < numbers.length; i++) {
    dos.writeInt(numbers[i]);
    dos.writeDouble(dblnumbers[i]);
}
```

Finally, it closes the dos stream:

```
dos.close();
```

The close() method automatically calls the flush() method which ensures that any data that is still waiting in the stream buffer is written out before the stream itself is closed. It also calls the close() method of the stream being processed – that is, the FileOutputStream that was passed to the DataOutputStream constructor.

Now all that remains is to write a procedure to load the data back from disk again. The loadBtnActionPerformed method does that:

```
DataInputStream dis = new DataInputStream(new
                            FileInputStream("test.dat"));
for (int i = 0; i < numbers.length; i++) {
    aNum = dis.readInt();
    aDouble = dis.readDouble();
    textArea1.append("Read (int)   : " + aNum + "\n");
    textArea1.append("Read (double): " + aDouble + "\n");
}
dis.close();
```

Writing and reading data from your own object types is a bit more complicated. It requires that you write each data item within an object one at a time and subsequently read them back in the same order they were written. Try to imagine how difficult it would be to write out complex networks of objects if I had to code all this behavior from the ground up. Let's take my adventure game as an example. In order to save the game state, I would have to save each Room object from my map one by one, being sure to save each Treasure object in each Room, and also the Treasure objects in the player's inventory. Then, when I wanted to reload a saved game I would need to reconstruct all these different Treasure objects, place them into the correct Room objects, put the player back into the right Room and reassemble the Player's inventory. Each time the game is played, the game state would change and the positions of all the Treasure objects would need to be reconstructed on each save or load operation. Fortunately, Java gives me a simple way of doing all this with very little effort on my part. It is called serialization.

SERIALIZATION

Serialization describes the process of saving an object's data into a stream. If you serialize one object that contains another object – for example, a `Room` that contains a `Treasure` – the contained object (here the `Treasure`) is serialized automatically when its container object (the `Room`) is serialized. This might not sound like a big deal. But imagine now that you have a complex network of objects of mixed types – a game that contains a map that contains multiple rooms each of which contain zero, one or more treasures. In fact, maybe some treasures, such as jewel-boxes and treasure-chests might themselves contain other treasures. Well, you begin to see the problem. Saving and loading that complicated network of objects one by one would be a daunting task. But I can serialize the entire tree of objects simply by serializeing the object at the *root* of the tree – here that's the game object – and all the other objects which that contains will be serialized automatically!

In order that an object can be serialized, its class needs to implement the `Serializable` interface from the `java.io` package. To do that you need to add this to each class declaration:

```
implements java.io.Serializable
```

In my adventure game, I add this `implements` statement to all the classes I want to be available for serialization:

```
public class Game implements java.io.Serializable
public class Treasure extends Thing implements java.io.Serializable
// …and so on
```

Unlike the interfaces we've looked at previously, the `Serializable` interface contains no methods or data fields. By implementing the `Serializable` interface a class simply *informs* Java that it is available for serialization. If you attempt to serialize objects whose classes do not implement the `Serializable` interface, an exception of the type `NotSerializableException` will be thrown.

SAVING AND RESTORING OBJECTS

As in the *FileInputStr* project which we looked at earlier in this chapter, when I want to save data to disk I begin by creating a new `FileOutputStream` object, passing to its constructor the name of the file to be created and opened. You may recall that in the *FileInputStr* project I passed a `FileOutputStream` object as an argument to the constructor of `DataOutputStream` which handled the nitty-gritty details of writing primitive data types to the output stream. In my adventure game I need to write complex object types, so instead of using a `DataOutputStream` I have used an `ObjectOutputStream`. This is how the Java documentation describes this class:

> An `ObjectOutputStream` writes primitive data types and graphs of Java objects to an `OutputStream`. The objects can be read (reconstituted) using an `ObjectInputStream`.

In my program, I create an `ObjectOutputStream` to write complex networks (or 'graphs') of mixed object types into a stream. That stream of data is written to disk by a `FileOutputStream`:

Adventure4

```
  FileOutputStream fos = new FileOutputStream("Adv.sav");
  ObjectOutputStream oos = new ObjectOutputStream(fos);
```

All I have to do in order to write all the objects in my game (and all the objects in other objects – the treasures in the rooms in the map in the game and so on) is call the `writeObject()` method to write the root-level object. When that object is written all the other objects (assuming they are serializable) are written too:

```
  oos.writeObject(game);
```

And finally I flush (to write out any unwritten data) and close the `ObjectOutputStream`. This is the complete code that handles the saving of my game data to a file called "Adv.sav":

```
try {
      FileOutputStream fos = new FileOutputStream("Adv.sav");
```

```
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(game);
        oos.flush(); // write out any buffered bytes
        oos.close();
        outputTA.append("Game saved\n");
    } catch (Exception e) {
        outputTA.append("Serialization Error! Can't save data.\n");
        outputTA.append( e.getClass() + ": " + e.getMessage() +"\n");
    }
```

Loading a game is just as easy. I create a `FileInputStream` object to read from the same file to which the objects were previously written. I pass this to the `ObjectInputStream` constructor and I then read in data using the `readObject()` method whose return value is cast to the `Game` class. This has the effect of deserializing all the saved data of my game and its entire network of objects, thereby recreating and reinitializing the game data in memory:

```
    try {
        FileInputStream fis = new FileInputStream("Adv.sav");
        ObjectInputStream ois = new ObjectInputStream(fis);
        game = (Game) ois.readObject();
        ois.close();
        outputTA.append("\n---Game loaded---\n");
    } catch (Exception e) {
        outputTA.append("Serialization Error! Can't load data.\n");
        outputTA.append( e.getClass() + ": " + e.getMessage() +"\n");
    }
```

Try this out to see just how powerful serialization is in action. Run the *Adventure4* project and move around the game map, taking and dropping objects. Save the game. Then move around the map some more and take and drop other objects. Then restore your previously saved game. You will see that the game is restored to its state when you saved it. The player will be placed back into the room that was the current location when the game was saved and the objects in the rooms (and in the player's inventory) are once again back where they were when the game was saved. Serialization is not only useful for game programmers, of course. It provides a powerful mechanism for saving and restoring data of all sorts.

Appendix

IDEs/Editors

Here are a few (though there are others) IDEs and editors that support Java:

NetBeans
https://netbeans.org

Eclipse for Java
https://eclipse.org/downloads/

IntelliJ IDEA
http://www.jetbrains.com/idea/

Web Sites

There are many web sites that provide useful information and sample Java programs. However, the two sites listed below are invaluable:

The Java Platform API
http://docs.oracle.com/javase/8/docs/api/

Oracle Java tutorials:
http://docs.oracle.com/javase/tutorial/

Bitwise Courses

This Java course (the video lessons, source code archive and eBook) was created by Huw Collingbourne, founder of Bitwise Courses. More courses on other language including, CX, C#, JavaScript, Object Pascal and Ruby, can be found on the Bitwise Courses web site at https://bitwisecourses.com

Bitwise Courses on YouTube

Bitwise Courses has a YouTube channel which includes numerous free tutorials, including some on Java and NetBeans. For news of all the latest tutorials, be sure to subscribe to our channel.