The Little Book of Java

Huw Collingbourne

bitwise books

The Little Book Of Java
Copyright © 2021 by Huw Collingbourne
ISBN: 978-1-913132-14-9

bitwise books is an imprint of **dark neon**

All rights reserved.

written by
Huw Collingbourne

The right of Huw Collingbourne to be identified as the Author of the Work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the prior written permission of the publisher, nor be otherwise circulated in any form of binding or cover other than that in which it is published and without a similar condition being imposed on the subsequent purchaser.

Download the Source Code

To download the source code please join our mailing list and you will receive a download link by email:

http://www.bitwisebooks.com/subscribe

Or scan this QR code:

Contents

CHAPTER 1 – GETTING STARTED 9

WHAT IS JAVA? 9
WHAT WILL YOU LEARN? 10
HOW TO FOLLOW THIS BOOK 10
DOWNLOAD THE SOURCE CODE 10
INSTALL JAVA 10
WHICH IDE? 11
USING NETBEANS 11
LOADING AND RUNNING A PROGRAM IN NETBEANS 12
LOADING AND RUNNING PROGRAMS IN OTHER IDES 13
LOADING GROUPS OF PROJECTS IN NETBEANS 13
COMPILING AND RUNNING PROGRAMS FROM THE COMMAND PROMPT 15
RUNNING A PROGRAM IN A NAMED PACKAGE 16
NAMING CONVENTIONS 18
MAKING SENSE OF THE TEXT 18
ABOUT THE AUTHOR 20
ABOUT THE TECHNICAL EDITOR 21

CHAPTER 2 – FIRST STEPS IN JAVA 21

HELLO WORLD 21
HOW TO LOAD THE SAMPLE PROJECTS IN NETBEANS 22
RUNNING A PROGRAM FROM THE COMMAND PROMPT 23
INTRODUCTION TO JAVA CODE 25
PACKAGES 26
CLASSES 26
METHODS OR FUNCTIONS 27
PASSING COMMAND LINE ARGUMENTS 28
WHAT ARE COMMAND LINE ARGUMENTS? 29
COMPILING JAVA PROGRAMS – FROM SOURCE CODE TO BYTECODE 30
VISUAL JAVA APPLICATIONS 32
HOW TO CREATE A VISUAL INTERFACE WITH NETBEANS 33
DISPLAYING OUTPUT 35

CHAPTER 3 – ELEMENTS OF JAVA PROGRAMMING 39
Variables and Types 39
Constants 40
A Tax Calculator with a User Interface 41
Command Prompt Calculator 42
More on Packages 44
Importing 45
Type Conversion 46
Primitives and Wrappers 47
Type Casts 50
Conversion Methods 51
Numeric Literals 51
Automatic String Conversions 53
Boxing and Unboxing 54
Strings 56
Escape Characters 57
CHAPTER 4 – OBJECT ORIENTATION 59
Classes and Objects 59
A Class is the Blueprint of an Object 61
Constructors 62
Class Hierarchies 63
Functions or Methods 66
Return Values 68
Overloaded Methods 68
Class Methods 70
static Methods 71
static Constants 74
Class Networks 75
CHAPTER 5 – TESTS AND OPERATORS 77
Operators 77
= Assignment 78
== Equality 78
Comparing Strings 79
String.equals() 80
String Equality 80
Tests and Comparisons 82
if...else 83
switch...case 84
Logical Operators 85

boolean Values 87
Compound Assignment Operators 88
Increment ++ and Decrement -- Operators 89
Prefix and Postfix Operators 90

CHAPTER 6 – ARRAYS AND COLLECTIONS 91

Arrays 91
Arrays are Zero-Based 92
Indexing Errors 93
Initializing Arrays 93
for Loops 95
Collections 97
ArrayList 97
Typed ArrayLists 98
Generics 99
Maps and Dictionaries 100
HashMap 101
Declaring an Object Using an Interface 104
Creating and Initializing a Typed HashMap 104

CHAPTER 7 – LOOPS 107

for 107
for...each 109
while 110
do...while 112
Multidimensional Arrays 113
break 119
continue 121
Breaking Out of a ‘for’ Loop 124
Labelled break 125

CHAPTER 8 – ENUMS, INTERFACES AND SCOPE 129

Enum Types 129
Assigning Values to enum Constants 133
The Enum Class 136
The Planet Enum 136
Interfaces 137
Extending and Implementing 138
Interfaces as ‘Contracts’ 139
Custom Interfaces 141
Abstract Classes 143

An Abstract Class Example 144
Scope 147
Access Modifiers 149

CHAPTER 9 – GENERICS AND EXCEPTIONS 153

Strongly-typed Lists 155
Generics 156
Generic Key-Value pairs 156
Generic Classes 157
Generic Lists and Collections 159
Overriding Methods 160
@Override Annotation 162
Overloading Methods 163
Exceptions 164
Exception Types 168
finally 169

CHAPTER 10 – FILES AND SERIALIZATION 171

Random Access Files 171
Streams 176
Serialization 178
The Serializable Interface 180
Saving and Restoring Objects 180
Where Next? 183

APPENDIX 185

The Adventure Game Projects 185
The Little Java Book Of Adventure Game Programming 185
IDEs/Editors 187
Web Sites 187
Bitwise Books and Courses 187

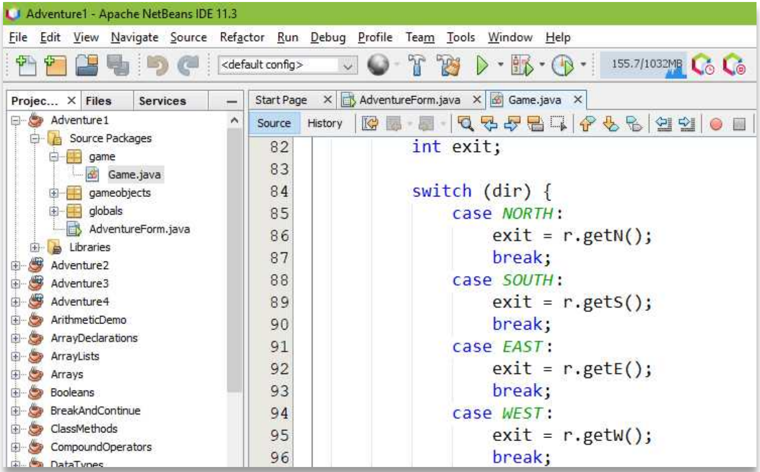
A MESSAGE FROM THE AUTHOR 189

Leave a Review 190
More Little Programming Books… 191

Chapter 1 – Getting Started

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

If you already have a Java editor and compiler installed and you know how to run and build Java programs, you are all ready to go. When I wrote the code for this book, I used the NetBeans IDE (Integrated Development Environment). Even if you generally use some other Java editor, I recommend that you install NetBeans so that you will be able to load and run my sample programs in the simplest possible way.



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.

What will You Learn?

This book assumes no prior knowledge of programming in general or the Java language in particular. It teaches you everything you need to write, compile and run 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, 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 book. Be sure to download the ready-to-run code samples provided in the source-code archive available from the publisher's web site.

How to Follow this Book

It is entirely up to you to decide whether you prefer to read this book, chapter by chapter, from start to finish or just 'jump in' to specific sections that are of interest to you. If you are new to programming, I would suggest that you read everything in strict order. If you are an experienced programmer – but new to Java – you may want to read the introductory information about IDEs and compilation then browse to topics that are unfamiliar to you.

Remember that source code is available for all the projects described in this book. You may prefer to run the code first and then read the explanations in the book. I would also recommend that you try modifying and extending my example code to help you understand how the Java language works.

Download the Source Code

All the source code shown in this book is available for download. Just sign up to our mailing list and we will send you the download link:

www.bitwisebooks.com/subscribe

Install Java

If you don't already have Java installed, you should download it from:

https://www.java.com/

There are two components needed in developing Java – the Java Runtime Environment or JRE and the Java Development Kit (the compiler and libraries) or JDK. You must install the JDK. If you install the JDK, the JRE is usually installed as well.

Installation Help

The Java web site has links to Help on installing and using Java on different operating systems: https://www.java.com/. It is important to read this help information in order to avoid or fix any potential problems.

Which IDE?

You can load the source code into any programming editor that supports Java syntax. The projects were developed using the NetBeans IDE. If you are unsure which IDE to use, I recommend using NetBeans.

To install NetBeans, download one of the ready-to-run installation programs from:

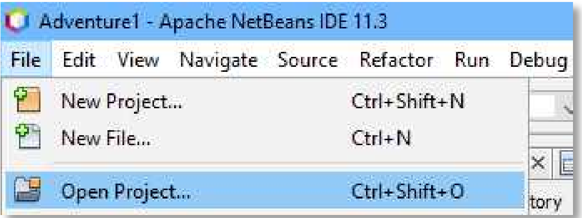
https://netbeans.apache.org/

Using NetBeans

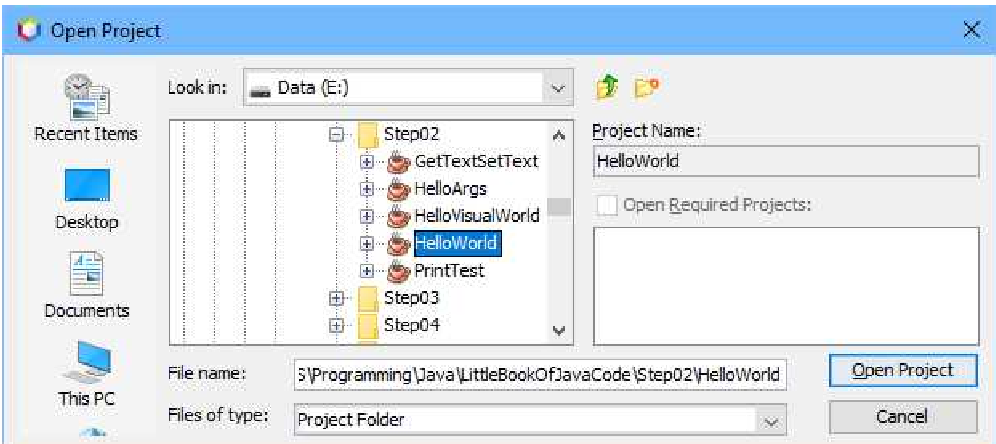
While you don't *have* to use NetBeans in order to write and run my Java programs, I will occasionally show all the steps you need to follow in order to create files and 'projects' using NetBeans. In addition some projects have been created using the NetBeans tool for designing visual interfaces. If you load these projects into NetBeans you will be able to view and edit the visual interface. You probably won't be able to do that if you load those projects into other IDEs. For these reasons, I ***strongly recommend*** that you use NetBeans to load, edit and run the sample code that accompanies this book.

Loading and Running a Program in NetBeans

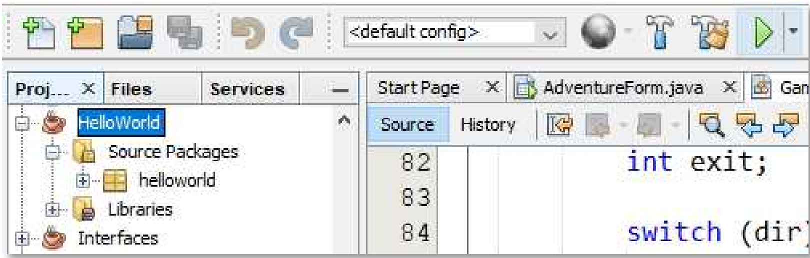
If you are using NetBeans, you can load a project from the source code archive just by selecting *File*, *Open Project*:



Now navigate to the directory containing the NetBeans project file (this will be indicated by a little coffee-cup icon in the project-browse dialog) then select the project and click the *Open Project* button:



If you already have a project open in NetBeans the new project will be loaded beneath it in the Projects window. To compile and run a project, first make sure the project is selected in the Projects window then select *Run* from the *Run* menu or press the green arrow-head button in the NetBeans toolbar:



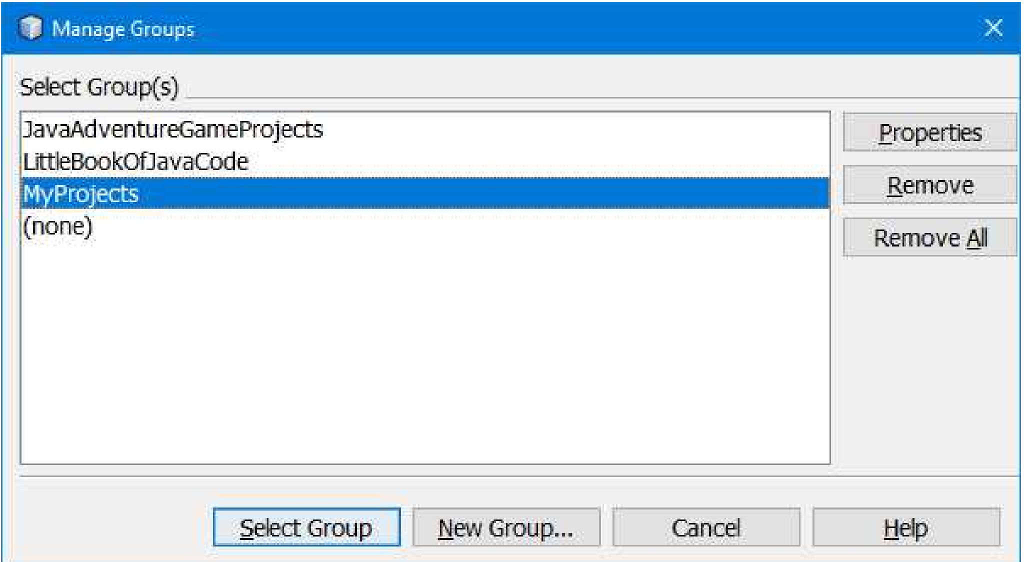
Loading and Running Programs in Other IDEs

If you are using another editor or IDE you need to check the developers' documentation of that IDE in order to learn how to create projects and add Java code files to them. While my code should work equally well with all Java editors and IDEs, I do not provide any help or support for any IDE other than NetBeans. So if you decide to use another IDE, you must be sure that you already know how to create, compile and run projects and how to add code files to that IDE. Alternatively, you may edit the code files in any Java-enabled editor and compile and run the code from the command prompt.

Loading Groups of Projects in NetBeans

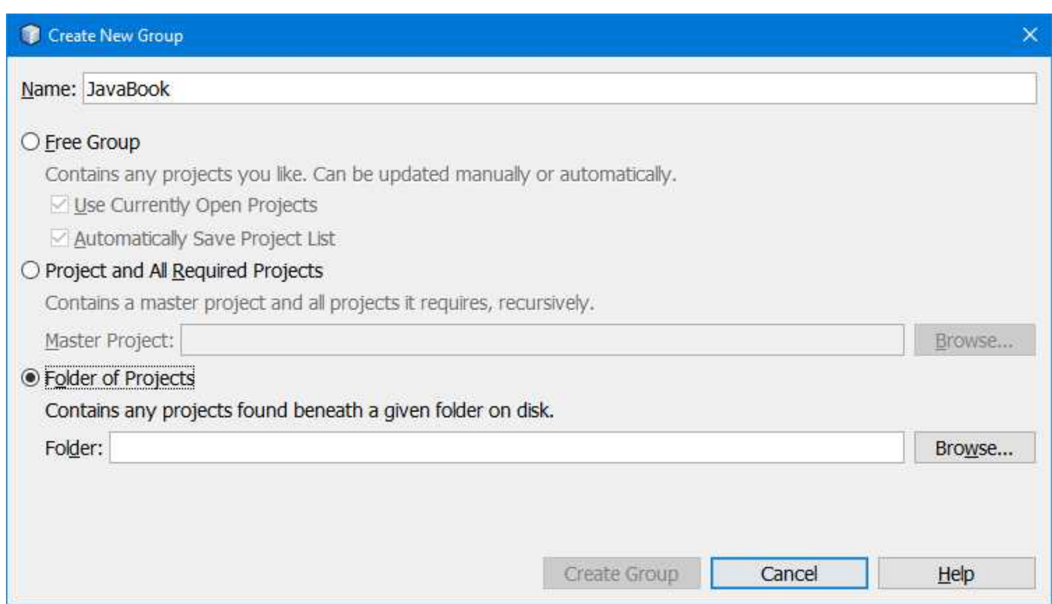
Since the code archive supplied with this book contains a great many projects you may prefer to load all the projects simultaneously and then switch from one to another by selecting a project in the Projects window. To load multiple projects you need to create a Project Group. This is how to do that with the code archive.

- Select *File | Project Groups…*
- The *Manage Groups* dialog pops up.

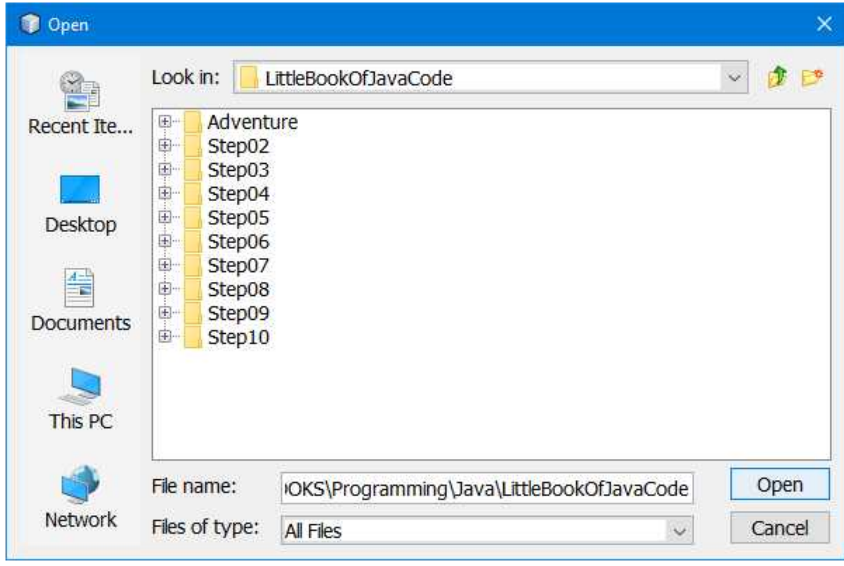


- Click *New Group*
- Give the group a name such as 'JavaBook'

- Select the *Folder of Projects* option
- Click the *Browse* button



Assuming you have unzipped the code archive into the *LittleBookOfJava* directory, you should browse to that directory:



When you click 'Create Group', the NetBeans *Projects* window will now list all the projects beneath the selected directory:

Compiling and Running Programs from the Command Prompt

If you have properly installed Java, you can go to the command prompt or Terminal in order to compile and run programs. You must be sure that both the Java compiler *javac.exe* and the java application launcher *java.exe* are on the system path so that they can be run from any directory. To verify this open a command prompt and type these two commands:

```
javac
java
```

In both cases, some lines of help text should be displayed. If, on the other hand, you see an error message telling you that the command is not recognised or a program with that name cannot be found, you need to add the directory containing those programs to your system path. For example, on Windows 10, you could open Windows Settings, then search for "path" in the search box. Click the link to edit system variables. In the dialog box, click *Edit the system environment variables*, then click *Environment Variables*. In the *System Variables* list, select `Path` then click *Edit*. Click *New*. Now add the full path to the directory containing *java.exe* and *javac.exe*. For example, on my PC this is:

```
C:\Program Files\Java\jdk1.8.0_111\bin
```

Click *OK*. Open a new command prompt and check that the `java` and `javac` commands are understood. To compile a program, open a command prompt and navigate to the directory containing the 'main' Java program. For example, in the *AdventureGame1* project, the main program is *AdventureGame.java* and it is likely to be found in a directory such as:

```
D:\JavaAdventureGame\AdventureGame1\src
```

Once logged into that directory, enter the name of the main code file after the `javac` command in order to compile the project:

```
javac AdventureGame.java
```

This should create a compiled file called *AdventureGame.class*. To run this, now enter `java` followed by the name of the compiled file (without the extension `.class`):

```
java AdventureGame
```

As long as the class that was compiled is not declared to be within a named package, this command should run the program.

Running a Program in a Named Package

If you compile a program in which the main class is declared to be inside a named package, you must include that package name when you run the program. Let me illustrate the problem. The main class of the *HelloWorld* sample program in the *\Step02* folder of the code archive is `HelloWorld.java`. This begins with this statement:

```
package helloworld;
```

What is a Package?

Packages provide ways of grouping together related code units. They are explained more fully in Chapter 2.

If I open a command prompt in the directory containing the file `HelloWorld.java` (if you have unzipped the code archive on the *C:* drive this directory will have a name something like the following: *C:\LittleBookOfJavaCode\Step02\HelloWorld\src\helloworld*) I can compile the code with this command:

```
javac HelloWorld.java
```

This will compile my code and create a file called *HelloWorld.class* which is capable of being run by the Java virtual machine. I now try to run this file using the command:

```
java HelloWorld
```

But it won't run! I will see this error message:

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

That is because the `HelloWorld` class is defined to be in the `helloworld` package. In order to run this program, I need to go up to the directory above the *\helloworld* folder – that is, to *C:\LittleBookOfJavaCode\Step02\HelloWorld\src* and then enter this command which includes the package name `helloworld`:

```
java helloworld.HelloWorld
```

The program should now run correctly and display this output:

```
Hello world
```

If you find this confusing, remember that if you compile and run programs from within an IDE such as NetBeans, all the details of packages and folders will be handled automatically. It is only when you compile and run from the command prompt that you need to enter these details yourself.

Java is Case Sensitive

If your main class is called *AdventureGame* with a capital 'A' and a capital 'G', you must be sure to use the same mix of upper and lowercase letters when you try to run the program.

Let's suppose I forget to capitalize the 'G' when I enter the command like this:

```
java Adventuregame
```

Java will show an error message similar to this:

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

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 functions (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 on a strict set of naming conventions in this book. In real-world programming, the naming (and code formatting) conventions are often defined by the team-leader of a specific project. If you are programming alone, you have more freedom to decide on naming conventions. In this book I will mention important conventions, ones that are very widely adopted or are syntactically required, 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;
}
```

Some code may be preceded by a project name, sometimes with the source file name shown in parentheses:

AdventureGame1 (Thing.java)

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

This shows that the code of *Thing.java* can be found in the *AdventureGame1* project or folder in the source code archive. To follow along with the tutorial, you can load the code files from the named project into your preferred Java editor or IDE. 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!
```

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

This is a Note

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 is the author of the cult text adventure game, *The Golden Wombat Of Destiny*. Huw has been a programmer for more than 30 years. He is an online programming instructor with successful courses on Java, C, C#, Object Pascal, Ruby, JavaScript and other programming languages and topics. For a full list of available courses be sure to visit the Bitwise Courses web site: http://bitwisecourses.com/

Huw is author of *The Little Java Book of Adventure Game Programming*, *The Little Book Of C#*, *The Little Book Of Ruby*, *The Little Book Of C*, *The Little Book Of Pointers* and *The Little Book Of Recursion* from Bitwise Books. He is a well-known technology writer in the UK and has written numerous opinion and programming columns for a number of computer magazines, such as Computer Shopper, PC Pro, and PC Plus. 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.

About the Technical Editor

Dr. Dermot Hogan is a software developer who has led major projects written in Assembly Language, C , C# and a number of other programming languages. A specialist in real time trading technologies, he has managed and developed global risk management systems for several international banks and financial institutions. He is the lead developer of the independent software company, SapphireSteel Software. He holds a Ph.D in physics from the University of Cambridge. His current area of research is devoted to robotic control and imaging systems.

Chapter 2 – First Steps in Java

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

By tradition, the first program many people like to create when learning a new programming language is one that simply displays "Hello world". Let's do that now.

Hello World

If you are using NetBeans, select *File* | *New Project*. In the *New Project* dialog, make sure the *Java* (or *Java with Ant*) 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 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 something like this:

HelloWorld.java

```
package helloworld;

public class HelloWorld {

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

}
```

In fact, there may also be some 'comments' – that is, text placed between `/*` and `*/` or placed after the `//` 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.

Why 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
   the words "Hello world"
   on a new line */
```

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
```

An IDE such as NetBeans may automatically insert comments when it creates code files. You may edit or delete those comments without affecting the behavior of the code.

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, the text `Hello world` – displayed in the NetBeans *Output* window.

How to Load the Sample Projects in NetBeans

If you are using NetBeans, you can load the projects from the source code archive one at a time or you can create a 'project group' so that all the projects can be loaded simultaneously.

How to Load a Single Project

To load a single project, select *File | Open Project*, then browse to the directory containing the project (e.g. *C:\LittleBookOfJavaCode\Step02*), click the '+' icon to expand the directory and select a named project (e.g. *HelloWorld*).

How to Create a Project Group

To create a group of projects stored beneath a directory, select *File | Project Groups* then click *New Group*. In the dialog, select *Folder Of Projects*, then click the *Browse* button. Find the folder containing the project folders that you wish to add to the group. If you have unzipped the sample source code into the default directory name, this will be a folder called *\LittleBookOfJavaCode*. Select this folder and click *Open*.

If necessary, edit the name field in the '*Create New Group*' dialog then click the *Create Group* button. You should now see all the projects beneath your selected folder shown in the Netbeans Projects panel.

If you close the project group (*File | Close All Projects*) you can open it up again by selecting *File | Project Groups*, highlighting the group name in the dialog and clicking *Select Group*.

How to Activate a Project in a Project Group

When you want to compile and run a specific project in a group, highlight the project name in the Projects panel then click the *Run Project* button (the green arrowhead) or select the *Run | Run Project* menu item.

Running a Program from the Command Prompt

If you want to run a Java program from the command 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. If you are using the NetBeans IDE, as I recommend, you will be able to compile and run programs using menus and buttons within the IDE itself rather than from the command prompt.

Using the Command Prompt

If you plan to compile and run programs from the command 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

Load the *HelloWorld* project provided in the code archive. Look at the code in the file *HelloWorld.java* (in NetBeans, this is shown under the *Source Packages\helloworld* branch of the Projects window. Let's now try to identify the essential parts of this code. The program begins with this:

HelloWorld (HelloWorld.java)

```
/*
 * The Little Book Of Java - sample Java code
 * http://www.bitwisebooks.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, it is a predefined word that has a special meaning in a Java program, 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 the *HelloWorld* 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 book.

Packages

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. Packages can help you to organize your code, just as the named folders on your computer's hard disk can help you to organize your documents. Your disk folders group together related documents. Your Java packages group together related code. In fact, Java packages are often placed into disk folders that have the same names as the packages. This has the advantage of making the organization of packages clear.

Following the package declaration in *HelloWorld.java*, we come to this:

```
public class HelloWorld {
```

Classes

If you have programmed an object oriented language before you will probably be familiar with the concept of a class. If this is the first time you have come across this, it is enough for now to know that a class defines a 'block' of code that may contain some data and also some 'subroutines' or 'functions'. In this program, I have a class called `HelloWorld`. The code of the class is contained between a pair of curly brackets. This class is extremely simple: it 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 or Functions

In Object Oriented terminology, a named block of code – that is, a function or subroutine – is called a 'method'. In Java, the names of methods 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 method (or 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 in my code the function contains a comment to document 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 a data 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. You don't need to understand the way this works for now. I'll have more to say about printing and streams in Chapter 10.

Strings

A *string* is a sequence of characters delimited by double-quotes. For example "Hello" is a string containing five characters.

In order to call the `println()` method I have to specify the full 'path' in the form of the name of the class `System`, the name of the field `out` and finally the name of the function `println()` function, all joined together by dots, like this: `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. This is what I see if I run the program in NetBeans:

Output - HelloWorld (run) × Notifications Search Results

```
run:
Hello world
BUILD SUCCESSFUL (total time: 0 seconds)
```

Passing Command Line Arguments

Sometimes it is useful to pass an argument (a piece of data) 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 as soon as the program begins running.

To do this you can pass one or more string arguments to the program. You can see how to do this in the *HelloArgs.java* sample project:

HelloArgs (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 line – that is, 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:

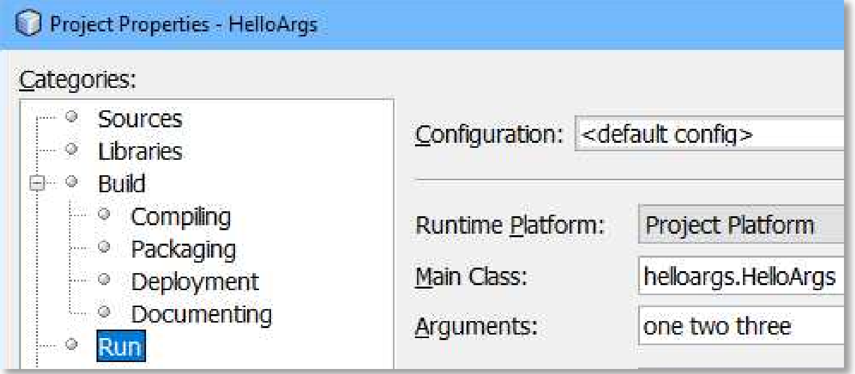
```
Hello.
one
two
three
```

Remember to Run Within the Package

As explained in Chapter 1, when you want to run a program in which the main class is declared to be inside a package, you need to be at the command prompt *within the directory one level above the package directory* (for example, here the package directory is *\helloargs*) and you must include the package name (here that's `helloargs`) as well as the class name (`HelloArgs`), like this:

```
java helloargs.HelloArgs
```

If you compile and run the program from within an IDE such as NetBeans, this will be done automatically. In fact, you can even get NetBeans to pass arguments to the program. To do that, right-click the project name (*HelloArgs*) in the *Projects* window, click the *Properties* menu item then select *Run* in the left-hand of the dialog. Enter any arguments to pass to the program into the *Arguments* field. Here I've entered three arguments: *one two three*



What are Command Line Arguments?

When you enter one or more items of text after the name of the program itself when you run a Java program, those items of text are passed as 'arguments' to the program. In this

example, I call the Java virtual machine (java) and ask it to run a program called *HelloArgs* in the *helloargs* package. I then pass three arguments, `one two three`:

```
java helloargs.HelloArgs one two three
```

The arguments are passed as an 'array' – a sequential list – of strings: "one", "two" and "three". This array is automatically assigned to the `args` argument in `main()`:

HelloArgs (HelloArgs.java)

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

My code now 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 (args.length == 0){
    System.out.println("You didn't enter any commandline arguments.");
}
```

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 (to repeatedly run the code between curly brackets) which 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.

Compiling Java Programs – from Source Code to Bytecode

Some programming languages – such as C and Pascal – are compiled into 'machine code'. This is a non-human-readable type of code that can be run by a specific combination of computer hardware and operating system. This means that if you compile a C program on Windows, for example, it won't run on Linux or MacOS. Java programs, on the other hand, compile to 'bytecode' which can be run by a special bytecode-running program called the Java Virtual Machine. If you compile a bytecode program on one operating system such as Windows, it can also be run on any other operating system – even using different computer hardware – such as Linux or MacOS, just so long as there is a Java Virtual Machine installed on that operating system.

The Compilation Process

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 Java Compiler

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.

Compiler Options

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 command prompt. More detailed documentation can be found here:

http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html

More information on the `java` command can be found here:

http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html

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 command prompt:

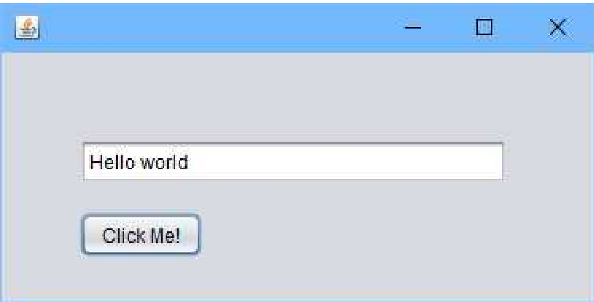
```
javac HelloWorld.java
```

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
```

Visual Java Applications

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 window which contains a number of 'components' such as buttons and text fields. It turns out that Java (especially when used from an IDE such as NetBeans) makes it very easy to create visual applications and some of the programs described in this book, and provided in the source code archive, will be have 'visual' user interfaces rather than having to be run from the command line.



This is "Hello World" with a user interface (the HelloVisualWorld project in the code archive).

Visual Design in Other IDEs

Other IDEs such as Eclipse and IntelliJ also provide visual Graphical User Interface (GUI) designers. Typically a Java visual interface relies upon a set of Java tools and classes called Swing. Note, however, that a design created using one IDE may not be directly editable in another IDE. The visual designs supplied in the code archive of this book have all been created with NetBeans.

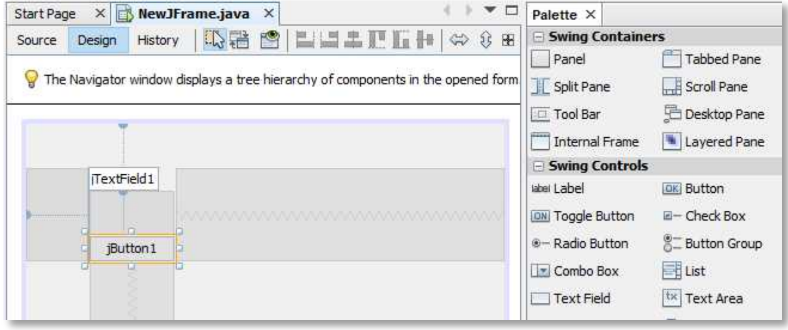
How to Create a Visual Interface with NetBeans

Here I want to guide you step-by-step through the creation of a visual version of the 'Hello world' program in NetBeans. This program will pop up in a window and it will display '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.

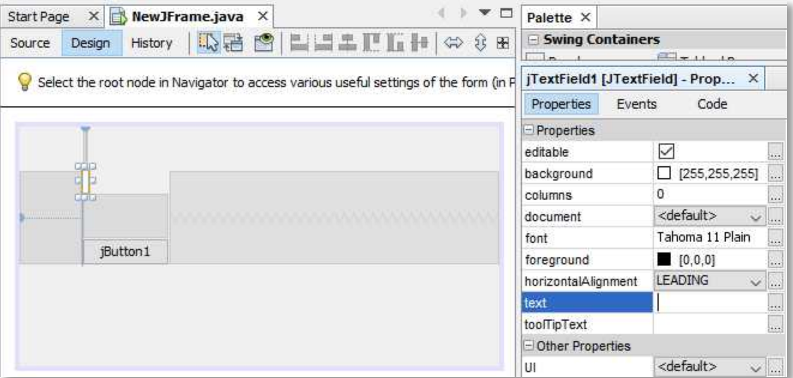
- In NetBeans select *File* |*New Project* |*Java* (or *Java With Ant*) | *Java Application*.

In some versions of NetBeans, you will need to select Java in the left-hand pane. In other versions, you need to select Java With Ant.

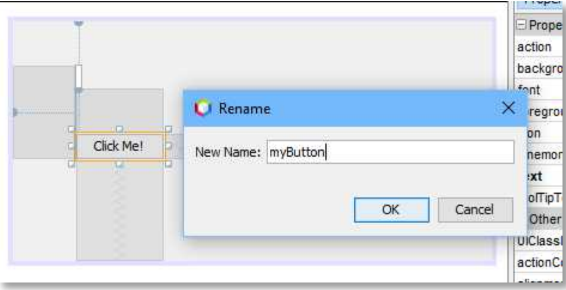
- Click *Next* (optionally browse to a project location) and give the project a name.
- *Un*check *Create Main Class* (make sure the item is *not* ticked). Click *Finish*.
- Right-click the main node of your project in the Project panel.
- From the popup menu, select *New* | *JFrame Form*. Click *Finish*.
- 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*
- Use your mouse to drag out the right edge of the text field to the same width as the button.
- 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!

You should now see a popup user interface similar to the one shown below. Click the button to display "Hello World" in the text field:

Displaying Output

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 (command prompt) applications.

Printing to the Command Prompt

When you want to display or 'print' some text at the command prompt you can use the `print()`, `println()` or `printf()` functions. 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
```

printf

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 an 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.
```

String Format Specifiers

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:

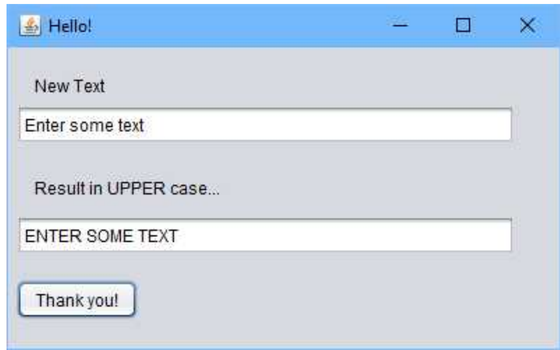
http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

Displaying Text in Visual Controls

When you want to display strings in a graphical user interface, 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 code appends the '!' character to the existing title, "Hello":

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

Chapter 3 – Elements of Java Programming

In this chapter we'll look at how different types of data are categorised and how data items are assigned to variables.

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.

Variables and Types

In Java, 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;
```

Variables without Types?

It is also possible, in some cases, to declare a variable using the keyword `var` instead of specifying its data type. Java may then work out or 'infer' the actual data type. In most cases, however, you should specify an actual data type such as `int`, `double` or `String` whenever you define a variable.

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 (the subtotal plus the tax), you might have three floating point (`double`) variables named `subtotal`, `tax` and `grandtotal`.

These variables could be assigned different values each time a new calculation is done, based on numbers entered by the user. You could 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 some other programmer 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% to the `TAXRATE` variable like this:

```
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, you can create a constant by placing the `final` keyword before the declaration like this:

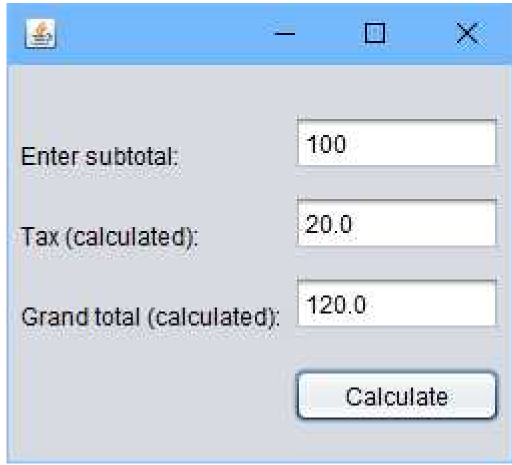
```
final double TAXRATE = 0.2;
```

Now it would be an error (my program would not compile) if I tried to assign a new value to `TAXRATE` in my code.

A Tax Calculator with a User Interface

I've decided to create a simple tax calculator with a simple user interface which I've designed in NetBeans. As explained in Chapter 2, NetBeans provides tools to create visual applications simply by dragging and dropping controls such as buttons, fields and labels onto a 'form'.

When run, a visual application pops up in its own window. In my tax calculator, the user can enter a numeric value into the subtotal text field and then click the Calculate button to compute the tax and grand totals. These computed values are then displayed in two other text fields:



This is the calculation method which runs when the button is clicked:

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 book.

Command Prompt Calculator

Of course, you don't have to design user interfaces for your applications. Sometimes you may want to run the program from the command prompt. To show you how to do this, I've rewritten my tax calculator as a command prompt application (*TaxCalcCmd*):

TaxCalcCmd

```
public class TaxCalcCmd {

    public static void main(String[] args) {
        final double TAXRATE = 0.2;        // 20% tax rate
        String input;
        double subtotal;
        double tax;
        double grandtotal;
        Scanner sc;

        sc = new Scanner(System.in);        // input stream
        System.out.println("Enter a number: ");
        System.out.print("> ");
        input = sc.nextLine();              //read string
        subtotal = Double.parseDouble(input);
        tax = subtotal * TAXRATE;
        grandtotal = subtotal + tax;
        System.out.printf("Subtotal = %.2f\nTax = %.2f\nGrand Total = %.2f\n",
                subtotal, tax, grandtotal);
    }
}
```

The calculation is done in the same way in this command line program as in the previous visual program. The only real difference is that this time the program reads input that was entered at the command prompt and displays the results in the form of a formatted string.

The `System` class provides the `in` and `out` fields for reading and writing to and from input. The `in` and `out` fields are 'streams' of data. Think of them as storing the characters we need to read and write. We'll look more closely at streams in Chapter 10.

In order to get at the input data, we use the `Scanner` class. This provides various ways of manipulating text such as the `nextLine()` method, which I use here, to return a line of text. In my code, then, I create a `Scanner` object, `sc`, and assign a line of text to the string variable, `input`:

```
Scanner sc;

sc = new Scanner(System.in);
input = sc.nextLine();
```

I 'parse' (read and convert) that string to a floating-point `double` called `subtotal`:

```
subtotal = Double.parseDouble(input);
```

Having then calculated the tax and grand totals and in the previous program, I display formatted output using the `printf()` method of `System.out` (which was explained in Chapter 2):

```
System.out.printf("Subtotal = %.2f\nTax = %.2f\nGrand Total = %.2f\n",
                subtotal, tax, grandtotal);
```

When this program runs, you must enter a number at the command prompt. This is what you will see:

```
> java taxcalccmd.TaxCalcCmd
Enter a number:
> 100
Subtotal = 100.00
Tax = 20.00
Grand Total = 120.00
```

Running from the Command Prompt

Remember, that when you want to run a command line program you must first compile it from the command prompt as explained in Chapter 1. With the *TaxCalcCmd* project, you need to open a command prompt in the directory above the folder containing *TaxCalcCmd.java*. For example, on Windows this may be:

```
C:\LittleBookOfJavaCode\Step03\TaxCalcCmd\src
```

Be careful. The way path names are entered may be different on other operating systems and you will need to be familiar with system commands for whichever operating system you are using! Having moved to the appropriate directory, you must now enter a command to compile the `TaxCalcCmd.java` in the `taxcalccmd` package. Again on Windows, this is:

```
javac taxcalccmd\TaxCalcCmd.java
```

Finally, you can run the compiled code:

```
java taxcalccmd.TaxCalcCmd
```

As mentioned before, the complications involved in compiling and running programs from the command prompt may be avoided by compiling and running them using the tools integrated into an IDE such as NetBeans.

More on Packages

We've already seen packages a few times in previous projects. For example, in the *TaxCalcCmd* program I wrote a code file called *TaxCalcCmd.java* and saved it into a sub directory named *taxcalccmd*. At the top of the code is this:

```
package taxcalccmd;
```

This states that the code in *TaxCalcCmd.java* exists in the package named `taxcalccmd`. So what exactly is a 'package'? Put simply, a package is a named group of code files. In the *TaxCalcCmd* program there is just one code file in the `taxcalccmd` 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 methods that do some sort of file handling operations, and so on. In other words, packages define named libraries of reusable code.

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 book are very short, this project will grow in size as we go through the book. It provides an example of a more complex Java 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 of the *Adventure1* project you'll see that I have imported the `Actor` and `Room` classes like this:

Adventure1 (Game.java)

```
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 as I have done here. 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 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 command line tax calculator sample program, *TaxCalcCmd,* the string that the user entered, input, was 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:

TaxCalcCmd

```
subtotal = Double.parseDouble(input);
```

In the version of the tax calculator that had a user interface, *TaxCalc*, I needed to display the value of the `double` number `grandtotal` in the `grandtotalTF` text field. I can only display a `String` in a text field so I passed that floating-point value, `grandtotal`, as an argument to `Double.toString()` which returned its string representation and I then called `grandtotalTF.setText()` to display that string value like this:

TaxCalc

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

Double or double?

In the examples given here, `Double` is a class. A class can wrap up data and functions (methods). Here the `Double` class provides the `parseDouble()` and `toString()` methods. Note that the `Double` class has a capital `D`. A `double` with a lowercase initial `d` is a 'primitive' data type. That is, it is an *actual* floating-point number. A primitive data type is just a piece of data so it does not have any methods. In Java, we may often use classes such as `Double` and `Integer` to perform actions on primitive data types such as `double`, and `int`.

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 such as 10.5 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 `doubles` and `ints`. You can find more examples of converting `double` and `int` numbers to and from strings in the *DataTypes* project.

DataTypes

```
// 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!

Primitives and Wrappers

Earlier on, I said that `double` with a lowercase initial `d` is a primitive data type whereas `Double` with an uppercase initial `D` is a class. Let me explain why that distinction matters. Java is a high-level object-oriented language. That means it uses data-types and programming techniques which are useful 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 that I added to the visual design of the *DataTypes* project:

```
    // Variables declaration - do not modify
    private javax.swing.JButton doubleBtn;
    private javax.swing.JButton intBtn;
    private
    private
    private
    // End o
}
```

Total = 115.6

double int parseStrings

This is what you will see when you run the *DataTypes* project. Each 'visual component' is created from a high-level data type – a 'class' such as `JButton`. When you use the NetBeans visual design tool to create a user interface, the classes and the code needed to deal with button-clicks is automatically generated by NetBeans. In fact, an experienced Java programmer could create a user interface by entering the code straight into the editor rather than by using a visual design tool (though that would be hard work, so I don't suggest you do it!).

`JButton` defines the appearance and the behavior of a button object. Other high-level data-types include `String` and `Integer`. A `String` object, just like a `JButton` object, wraps up some data and some behavior. The behavior of objects takes the form of methods such as the `setText()` method of a `JButton`, to change the button label, or the `toLowerCase()` method of a `String` to return the string in all lowercase characters:

ObjectsAndMethods

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

Complex data-types – objects with built-in methods – such as `JButton` and `String` are created in Java code and they are not meaningful to the computer hardware which only recognizes very 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*	$3.40282347 \times 10^{38}$ to $1.40239846 \times 10^{-45}$
`double`	*64 bits*	$1.7976931348623157 \times 10^{308}$ to $4.9406564584124654 \times 10^{-324}$
Character		
`char`	*16 bits*	
Boolean		
`boolean`	*1 bit*	`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. 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:

Primitive	**Wrapper Class**
`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, that it actually makes sense to treat one type as the other type – for example, it wouldn't make sense to treat "Hello world" as an integer). You can accomplish this by 'type casting'.

Let's assume that you 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).

double or float?

With floating point numbers, the `double` data type is 'double precision' which means that a `double` it can store a bigger possible range of floating point values than a 'single precision' `float`.

The compiler will object if you attempt to assign the larger (`double`) value to the smaller (`float`) variable, like this - that is like trying to pour a gallon of water into a pint glass:

```
floatVar = doubleVar;
```

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

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

But be careful! Remember that a `float` has a smaller range than a `double` so if the `double` value that you assign is greater than the range of a `float`, that value will be truncated (you may *try* to pour a gallon into a pint glass but the glass is never actually going to contain more than a pint). Look at this code (in the `castBtnActionPerformed()` method):

Primitives

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

If you run the program and click the 'Cast' button, this is message will be displayed:

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

Conversion Methods

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()` method 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 (in the `calcBtnActionPerformed()` method) showing a conversion from a `double d1`, to an `integer i`, using the `intValue()` method of the `Double` variable `d1`, and a conversion of the `int` value `i` back to a `double` using the `doubleValue()` method of the `Integer` variable `i`:

Primitives

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

If you click the 'Calc' button, this is what will be displayed:

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

Notice that the final values of `d1` and `d2` are different because `intValue()` created an integer from the original double value and the floating-point data (the digits 123 after the point) was lost in the process. When a new double value is created from this integer value, a 0 is placed after the point (105.0).

Numeric Literals

When you enter a number such as 10 or 100.5 it is called a *literal*. It is not a variable or a calculated value – it is literally the number that you enter. 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 data types the Java compiler may complain. If, for example, you try to assign to a `long` variable a numeric integer (`int`) 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:

Primitives – typesBtnActionPerformed()

```
long l;
l = 9223372036854775807;
```

Even though a `long` (such as the variable `l`) is capable of holding the value 9223372036854775807, this assignment is an error because an integer literal is treated as an `int` – and an `int` has a maximum value of 2147483647. In NetBeans I see this error message:

```
long l;

l = 9223372036854775807;
```

integer number too large: 9223372036854775807

The assigned value is never used

(Alt-Enter shows hints)

In order to fix this I 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 = 9223372036854775807L;
```

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 the appropriate values:

```
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
```

The *Primitives* sample program contains an example of this which can be found in the `typesBtnActionPerformed()` method.

Automatic String Conversions

As I mentioned earlier, when you want to convert numeric values to strings – for example, if you want to display an integer in a text area using the `setText()` method – you 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:

```
incompatible types: double cannot be converted to String
----
(Alt-Enter shows hints)
        myTextArea.setText(total);
    }
```

So how is it that my original version worked without problems but this new version does not? It turns out that when you concatenate strings (here by appending the value of the `double` variable `total` to the string `"Total = "`), 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 and Unboxing

When you use primitives in your programs, you may also need to use the methods provided by the corresponding 'wrapper' classes. 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 the built-in methods of that class.

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 in 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;

boxedInt = 500;          // auto-boxing
boxedDouble = 200.25;

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

primitive

int unboxedInt = 500;

object

Integer boxedInt = 500;

methods...

floatValue();

toString();

doubleValue();

This diagram shows a simplified view of the primitive `int` variable, `unboxedInt`, compared with the `Integer` object, `boxedInt`. The primitive `int` is just a number; it has no built-in behavior. It can be used in calculations such as `10 * unboxedInt` but not much else. In short, a primitive is just a piece of data.

The `Integer` object, is altogether more complicated. It includes some data (here the `int` value 500) but it also has functions or methods such as `floatValue()`, `toString()` and `toDouble()` that can perform actions on that data or return values of different data types such as floats, strings and doubles.

You can think of an `Integer` object as a box that contains an `int` value and some software machinery (the methods) to do things with that data. It can also do simple calculations such as `10 * boxedInt` in which case it will 'auto-unbox' the `int` value that it contains so that it is as easy to use a complex `Integer` object in calculations as a simple `int` primitive.

Strings

We use strings frequently 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 its 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 minus 1.

Assuming the string `s` contains the text, "Hello world". The expression, `s.length()`, would return a value of 11.

How Can a String Literal Have Methods?

Earlier I explained that an `int` literal such as 500 is just a piece of data with no methods so I can't write `500.toString()`. I would need an `Integer` object in order to do that. So why is it that string literals have access to methods? For example, I can write `"hello".toUpperCase()` which will return the string "HELLO". This is because Java automatically creates a `String` object when you assign a string literal to a variable. In fact, although string values such as "Hello" are *literals* they are not *primitives*. A primitive is a very simple single piece of data such as a number or a character. A string is a whole series of characters and can therefore be regarded as a relatively complex data type.

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();
```

Below 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. A string is indexed from 0 (the first character) to the length of the string minus 1 (the last character):

```
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();
```

Java Strings are Immutable

Java strings are immutable which means that they cannot be *modified*. For example, when I call the `toUpperCase()` method on the string object `s`, that string object is not set into uppercase. The string "HELLO WORLD", which is returned by the `toUpperCase()` method is a *new string object*. It is this string which I assign to the variable `upcaseS`. The *original* string, `s`, remains unchanged: it still contains "Hello world".

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 form feed 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. In this chapter I'll explain what objects are, how they are related to classes and why object orientation is useful.

Almost everything you work with in Java – from a string such as "Hello world" to a treasure in an adventure game – is wrapped up inside an object that contains the data itself (for example, the characters in a string or the name and description of a treasure) and the functions or 'methods' that can be used to manipulate that data – such as, for example, a string object's `toUpperCase()` method.

Primitives are Different

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 as explained in *Chapter 3*.

Classes and Objects

Each object that you use is created from a 'class'. You can think of a class as a blueprint that defines the data and the behavior (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 'get' and 'set' accessor methods:

Objects (Thing.java)

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

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

public String getDescription() {
    return description;
}

public void setDescription(String aDescription) {
    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*.

Private Variables, Public Methods

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. If you create public variables, other programmers will have direct access to them. If you want to prevent direct access at a later date, it will be difficult to do so. But if you create public methods and private variables, you can change the code of the methods to limit access, display error messages or make other changes as required.

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
        name = aName;
        description = aDescription;
    }

    public String getName() {
        return name;
    }

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

    public String getDescription() {
        return description;
    }

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

A Class is the Blueprint of an Object

My `Thing` class doesn't do anything. It is simply the 'blueprint' of a `Thing` object. Before you can use an *actual* `Thing` object, you will have to create it from the `Thing` class just as you would have to create an *actual* car from a Car blueprint before you can drive it!

The Car blueprint and the `Thing` class are the *plans* or *designs* which define what a Car or a `Thing` are. From a single Car blueprint, a manufacturer may construct numerous car 'objects'. From a single `Thing` class, a programmer may construct numerous `Thing` objects. Given the class shown above, I can see that each `Thing` object will have a string `name` and a `description` along with get/set methods to allow me to assign and retrieve the `name` and `description` of each separate object.

Constructors

The `Thing` class also has a constructor method. A constructor is a special method which, in Java, has the same name as the class itself. Often a constructor method can be used to initialize the fields (the variables) of an object. Here, for example, my `Thing` constructor receives two string arguments, `aName` and `aDescription`, which it assigns to its private variables, `name` and `description`:

```
public Thing(String aName, String aDescription) {
    // constructor
    name = aName;
    description = aDescription;
}
```

In the *Objects* sample program (the *NewJFrame.java* file), I declare two object variables of the `Thing` type:

Objects (NewJFrame.java)

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

Before I can use these objects, I need to create them. I do that by using the `new` keyword followed by the name of constructor and a pair of empty parentheses if the constructor doesn't require any arguments – for example, if the `Thing` constructor didn't define any named arguments, I would create a new object like this: `new Thing()`. In fact, my `Thing` constructor requires two string arguments, `aName` and `aDescription`, so I place two strings, separated by a commas, between the parentheses:

```
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");
```

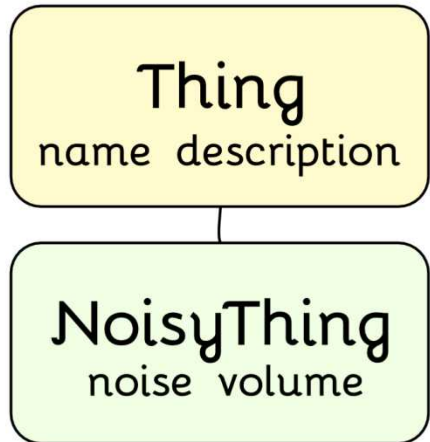
When you call a constructor in this way, a new object will be created and its fields (if there are any) will be initialized with the data – the arguments – that were passed to the constructor. The newly created object can then be assigned to a variable of the same type as the object's class as I have done in the preceding code which creates the two `Thing` objects, `mySword` and `myRing`.

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 an ancestor; the classes that inherit features from the ancestor class might be thought of as its descendants. A descendant class is also called a *subclass*. An ancestor class is often called a *superclass*.

Let's suppose I want to create a `NoisyThing` class that is a 'type of' `Thing` – so it inherits the features of the `Thing` class, that is, a `name` and a `description`, but it adds on two new features, a `noise` and a `volume`. You can picture it like this:



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) {
    noise = noise;
}

public int getVolume() {
    return volume;
}

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

When a new `NoisyThing` object is created it needs to assign values to all four variables (the ones it has defined – `noise` and `volume`, plus the ones defined by its ancestor class, `name` and `description`).

Its constructor method has four named parameters between parentheses so that the data items or 'arguments' needed 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:

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

The `super` method is used to call `the` superclass's constructor and it passes the arguments required by the superclass's constructor between parentheses. Having done that, I go on to initialize the variables defined in the `NoisyThing` class:

```
noise = aNoise;
volume = aVolume;
```

Parameters and Arguments

Parameters are the named variables declared between parentheses at the very top of a method, like this:

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

Arguments are the values (the data items) passed to the method when that method is called, like this:

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

While computer scientists may like to make a clear 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.

When I create a new `NoisyThing` object I must be sure to pass four arguments of the correct data types to its constructor so that the variables of the 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

As I mentioned in Chapter 2, 'method' is the object oriented term for what, in non-object-oriented (or 'procedural') programming languages is usually called a function, though some languages also use the terms 'procedure' and 'subroutine'.

Functions provide ways of dividing your code into named chunks. In procedural languages, functions are 'free-floating', by which I mean that they can be written almost anywhere in your code. In object oriented languages, functions are *bound into* objects. They have to be written inside a class definition 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.

private and public Keywords

The keywords `private` and `public` are 'access modifiers'. A `public` variable or method can be accessed from almost anywhere in your code. A `private` variable or method is 'visible' only within a certain region or 'scope'. For example, a `private` variable inside an object cannot be directly accessed from code that has been written outside the class of that object. A `public` method, however, can be accessed from code outride that object. Access modifiers will be explained in more detail in Chapter 8.

The parentheses following the method name may either be empty or they 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. The three functions that follow are examples of simple methods.

This is a function that takes no arguments and returns nothing:

Methods (NewJFrame.java)

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

This is a function that takes a single `String` argument and returns nothing:

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

This is a function that takes two `String` arguments and returns a `String`:

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

Messages and Methods

Object oriented purists may describe a 'method-call' as being a 'message' which is sent to an object. Consider this:

```
"Hello world".toUpperCase();
```

`Here` the message would be `toUpperCase()`. In principle, the object could refuse to comply with that message. For example, if you sent the message `"HELLO WORLD".toUpperCase()` the object might return an error message "This string is already in uppercase".

Usually, however, the message is 'answered' by a function with the same name (and, optionally, the same argument list) and it will perform the action requested by the message. Such a function provides a 'method' of dealing with the message – hence the name.

In traditional object oriented languages such as Smalltalk, the idea of message-passing is fundamental. In more modern object oriented languages such as Java, programmers tend to talk about function-calls or method-calls rather than 'message-passing' where the terms 'function' and 'method' are used interchangeably.

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();
```

Return Values

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.

```
Methods (NewJFrame.java)
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 a variable such. 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."));
```

Overloaded Methods

You may use the same name for several different methods in Java as long as the list of parameters is different in each method. The parameter list defines the '*method signature*' and when you create methods with the same names but different signatures, this is called '*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 pass to it a single `String` argument, `indexOf(String str)` returns the index within the `String` of the first occurrence of the specified substring `str`. There are, in fact, four overloaded `indexOf()` methods supplied by the `String` class. The overloaded methods can be distinguished from one another by the number of arguments or their data types. These are the four

overloaded `indexOf()` methods, each of which returns an `int` value giving an index into a string (remembering that the first character is at index 0):

`indexOf(int ch)`

Returns the index within this string of the first occurrence of the specified character.

`indexOf(int ch, int fromIndex)`

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

`indexOf(String str)`

Returns the index within this string of the first occurrence of the specified substring.

`indexOf(String str, int fromIndex)`

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

Here is an example using all four of these methods

IndexOf

```
public static void main(String[] args) {
    String s = "The quick brown fox jumps over the lazy dog";
    System.out.printf("s.indexOf(\"quick\") = %d\n", s.indexOf("quick"));
    System.out.printf("s.indexOf('e') = %d\n", s.indexOf('e'));
    System.out.printf("s.indexOf(\"the\", 3) = %d\n", s.indexOf("the", 3));
    System.out.printf("s.indexOf('u', 3) = %d\n", s.indexOf('u', 3));
}
```

And this is the output:

```
s.indexOf("quick") = 4
s.indexOf('e') = 2
s.indexOf("the", 3) = 31
s.indexOf('u', 3) = 5
```

Class Methods

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 object like this:

```
Room goldRoom;
int exit;

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

The code fragment shown above assumes that you already have a `Room` class defined (as in the sample project, *AdventureGame1*, which I'll explain later on).

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;
String s ="200";

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

This is a very complicated way of accomplishing a very simple task, however. I create an `Integer` object 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. In fact, I can do that. Instead of creating an *object* from the `Integer` class, I can 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 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)
```

static Methods

In order to be called directly from the *class* rather than from an *object*, a method has to be declared as `static`. If you consult the Java documentation you will find that 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)
```

The important thing to notice is that both methods have been declared using the keyword `static`. This is the secret to creating methods that '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 directly callable from your own classes (using the class name), 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:

Methods (MyObject.java)

```
public class MyObject {

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

Methods (MyObject.java) (continued)

```
    public MyObject() {
        obcount = obcount + 1;
        obnumber = obcount;
    }

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

    public String objectInfo() {
        return "This is object number " + obnumber + " from a total of "+
                + obcount + " objects.\n";
    }

    public int getObnumber() {
        return obnumber;
    }
}
```

Pay close attention to this `static` method:

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

The `numberOfObjects()` method displays a string that includes the value of the `int` variable `obcount`. You will see that this variable has also 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 store an integer representing 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 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 – that's 0 – plus 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. We can verify this. Here is the code which runs when the button labelled 'Object Methods' is clicked:

Methods (NewJFrame.java)

```
MyObject ob1;
MyObject ob2;
MyObject ob3;

outputTA.setText("");
outputTA.append(MyObject.numberOfObjects());
ob1 = new MyObject();
outputTA.append(MyObject.numberOfObjects());
ob2 = new MyObject();
outputTA.append(MyObject.numberOfObjects());
ob3 = new MyObject();
outputTA.append(MyObject.numberOfObjects());
outputTA.append("When all objects have been created: "
        + MyObject.numberOfObjects());
outputTA.append(ob1.objectInfo());
outputTA.append(ob2.objectInfo());
outputTA.append(ob3.objectInfo());
```

This is what is displayed:

```
There are 0 objects.
There are 1 objects.
There are 2 objects.
There are 3 objects.
When all objects have been created: There are 3 objects.
This is object number 1 from a total of 3 objects.
This is object number 2 from a total of 3 objects.
This is object number 3 from a total of 3 objects.
```

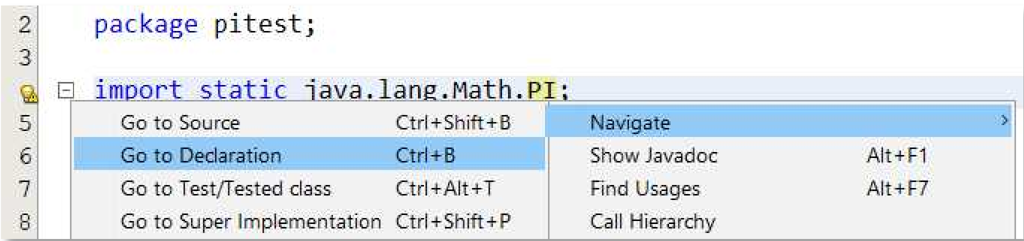
In brief then, *instance* variables store different values for each object. *Class* or `static` variables store a single value which is accessible by all objects created from that class. Instance methods are called from an object but class methods are called from a class. Incidentally, it is also possible to call static methods from an object but this can be confusing so it would generally be better to avoid doing this.

static Constants

Recall from Chapter 3 that constant values are generally declared using the `final` keyword. This prevents the value assigned to an identifier value being changed. In fact, you will often see that Java constants are also defined using the `static` keyword. 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 `PI` is declared:

```
public static final double PI = 3.14159265358979323846;
```

Java typically truncates this value to `3.141592653589793` as you can see if you run the *PITest* project. If you want to view constants in Java, you may be able to use the navigation tools of your IDE. In NetBeans, for example, you can right-click an imported constant name and select *Navigate|Go to Declaration* from the popup menu:



Class Networks

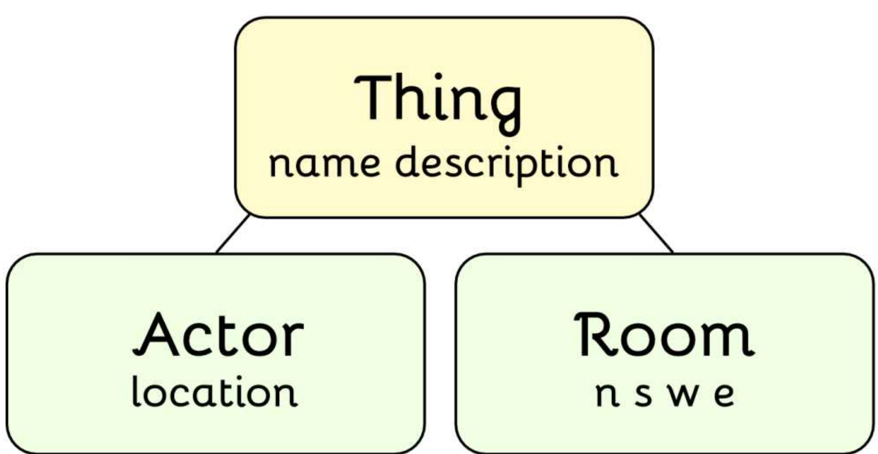
In this chapter, we've learnt how classes are used as the blueprints from which objects are created and how a descendant class can inherit the features of an ancestor class. In some types of application, it may be useful to have more than one 'level' of descent so that class `D` inherits from class `C`; class `C` inherits from class `B`; and class `B` inherits from class `A`.

It may also be useful to have more than one 'line' of descent so that both class `B` and class `X` descend from class `A`, while class `Y` descends from class `B`. In this way, we can create quite complex networks or 'family trees' of classes.

I'll show a real example of a complex class network when I explain my Java adventure game project which I will be developing in later chapters and which is summarized in the Appendix.

The diagram below shows the network of 'game objects' in the *Adventure1* project. The `Thing` class has a `name` and a `description`. The `Actor` class and the `Room` class both descend from the `Thing` class, so `Actor` and `Room` objects will also have a `name` and a `description`.

The `Actor` class defines an interactive character (or the game player) and it adds on a `location`. The `Room` class defines the rooms in the game map and it adds on four exits: `n`, `s`, `w` and `e`.



this and super

Sometimes you will see the keyword `this` which refers to the current object – the object that is being created. It is often omitted: `this.name = aName` has exactly the same effect as the assignment `name = aName`. However, if I had just named the parameter `name` instead of `aName`, then the keyword this is required to remove the ambiguity: `this.name = name`.

The keyword `super` is used in a similar manner to reference variables and methods in the superclass – for example, the superclass's constructor as we saw earlier, and again is often used to remove the ambiguity between a method with the same name in both the ancestor class and the subclass. I will cover method overriding and overloading in detail in chapter 9.

Chapter 5 – Tests and Operators

You will often need to change and test values in your programs using operators so that the code can take different actions in different circumstances. That's the subject of this chapter.

Frequently, in your programs, you will need to test if two variables have the same or different values. Java provides 'operators' to help you do these tests. In this chapter we'll discover how to use Java operators to write conditional tests, modify the values of variables and do arithmetical calculations.

We've seen some operators already – for example, in the *HelloArgs* program in Chapter 1, we used the equality operator (`==`) to test if the number of arguments entered on the command line (`args.length`) equalled 0:

```
if (args.length == 0)
```

We used the multiplication (`*`) and addition (`+`) operators, as well as the assignment (`=`) operator in the *TaxCalc* program from Chapter 3. You may recall that, in that program, we multiplied the `subtotal` variable by the `TAXRATE` constant and assigned the calculated value to the `tax` variable. The values of the `subtotal` and `tax` variables were then added together and the result was assigned to the `grandtotal` variable:

```
tax = subtotal * TAXRATE;
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.

= Assignment

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, `myIntVariable`:

```
int myIntVariable = 10;
```

== Equality

While *one* equals sign `=` is used to *assign* a value, *two* equals signs `==` are used to *test* a condition. 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!");
}
```

But 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!");
    }
}
```

Here I want to get the text from `textField1` and compare it with an empty string:

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

If it *is* an 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.

Comparing Strings

An integer can be compared with another integer so if `myIntVaruable` has the value 1, this test evaluates to `true`:

```
if (myIntVariable == 1)
```

However, when I compare a string "1" with another string "1" the test evaluates to `false`. In other words, Java considers two integers with identical values to be the same but two strings with identical values to be different.

String.equals()

In order to determine if two strings contain identical characters (or no characters at all) you need to use the `equals()` method of the string object. This is how I have rewritten my test (which now works as I intended):

EqualsOps

```
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. Remember, in Java, an integer is a simple piece of data – a primitive. But a string is an object – and every object is different from every other object. This is true even when two strings contain exactly the same characters.

Strings are Objects?

Not all programming languages deal with strings this way. Even the C# language which is, in many respects, very similar to Java, allows you two test two strings using the regular `==` equality operator. In C#, when two different strings contain the same characters, a test using the `==` operator, returns `true`. In Java, it returns `false`. You have to bear in mind that Java treats strings like any other objects. Two `Cat` objects, `catA` and `catB`, might both have the *same* name, "Tiddles", but they are nevertheless two *different* objects. Java treats string comparisons in the same way!

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 this test:

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

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 objects 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') the same 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 to 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

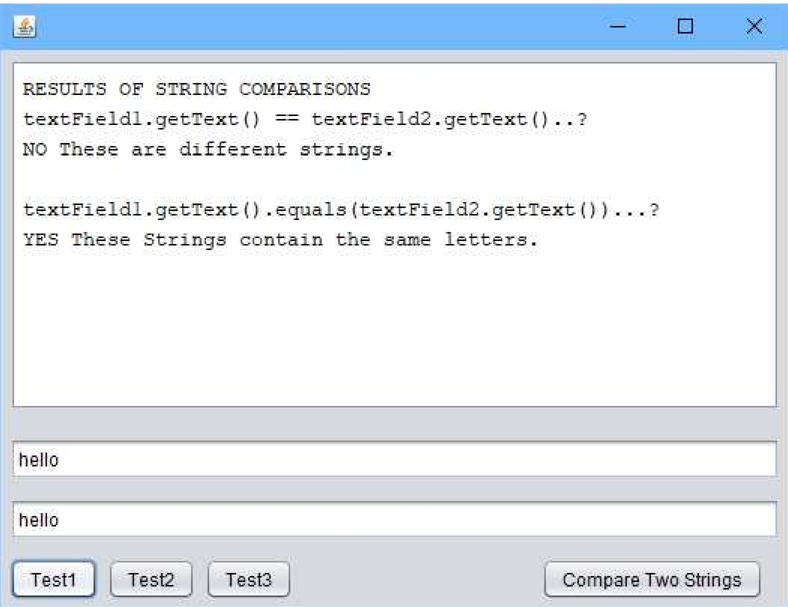
```
String s1 = "Hello";
String s2 = "Hello";

(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.

Which is the 'Best' Way To Test Strings?

Look at the code in the *StringCompare* project to help you to understand how strings are compared in Java. If you find this confusing, just remember that in most cases, it is *not* a good idea to compare strings using the `==` operator. You should, instead use the `equals()` method. Whereas the `==` operator returns different results depending on how and when a string was assigned to a variable, the `equals()` method is consistent in all cases.



Here I have run the *StringCompare* project, entered two identical strings into the text fields and clicked the 'Test1' button to compare the strings using the `==` operator and the `equals()` method.

The `==` operator returns `false`, indicating that these are two different string objects. The `equals()` method returns `true` indicating that the two strings contain exactly the same characters.

Tests and Comparisons

Java can perform tests using the `if` statement. The test itself must be contained within parentheses (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");
```

If you want multiple lines of code to be executed after the test, all those lines of code must be enclosed within curly brackets to form a 'block' of code.

Some programmers (myself included!) prefer to enclose single-line 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` block. The `else` block 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!");
}
```

Translated into plain English, the code shown above says: "If the input string (`inputStr`) is empty then show the string "You didn't enter anything" in the `output` text area. Otherwise, if the input string is either "hello" or "hi" then show the string "Hello to you too!" in the `output` text area. In all other cases, show the string "I don't understand that!" in the `output` text area.

The two vertical lines `||` in the code above is an operator which means 'or'. The `||` operator lets you chain together conditions so that a test evaluates to `true` when *either* the first *or* the second condition is `true`. See the section on Logical Operators later in this chapter for more information.

There are more operators to perform other tests on numbers. For example, the code following 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 test evaluates to `true` and the string "Test is true" is displayed. Otherwise the test evaluates to `false` and "Test is false" is displayed:

Tests

```
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, one after the other, it is often quicker to write them as 'switch statements' instead of multiple '`if...else if`' tests. Here is an example:

Tests

```
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 Java a switch statement begins with the keyword `switch` followed by a test value (here that's `inputStr`). Then you must put the values that you want to compare with the test value after one or more `case` keywords followed by a colon (for example, `case "hello":`). If the test value matches the `case` value any code after the colon is executed until the keyword `break` is encountered. 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.

In the preceding 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. Note that when `inputStr` is "hello" a match is made with `case "hello":` but, as there is no `break` directly after that test, the flow of execution trickles down to the next `case` statement, executes the code after `case "hi":` and only then exits from the whole `switch` block when it encounters a `break`.

```
                      // if the test condition inputStr == "hello"...
case "hello":         // ...a match is made here but there is no ‘break’
case "hi":            // ...so the code after *this* case test is run
    output.setText("Hello to you too!"); //...which shows this string in output
    break;            // ...and this ‘break’ now exits the ‘switch’ block
```

If the test value, `inputStr`, does not match any of the case values then the code in the `default` section is run, so "I don't understand that!" is displayed.

switch…case Test Values

Most commonly, the values tested by `switch` and `case` statements will be integer, character or string data types: `byte`, `short`, `char`, `int` and `String`. When a string is used, any test for equality is done by comparing the characters in the strings as though you were using the `String.equals()` method.

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* if that customer 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");
}
```

This program produces the following output:

```
You are not a rich young person
You are either rich or young or both
You are not a rich young parent
```

Boolean Values

Logical operators test Boolean values. A Boolean value can either be `true` or it can be `false`. Java provides a `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 parentheses to ensure that the component tests are all evaluated together to return a `true` or `false` value. For example, by placing parentheses around 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:

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

You can find some examples of these operators in the *CompoundOperators* project:

CompoundOperators

```
public static void main(String[] args) {
    int a;
    int b;

    a = 10;
    b = 2;
    a = a + b;
    System.out.println(a);
    a += b;
    System.out.println(a);
    a = a - b;
    System.out.println(a);
    a -= b;
    System.out.println(a);
    a = a * b;
    System.out.println(a);
    a *= b;
    System.out.println(a);
    a = a / b;
    System.out.println(a);
    a /= b;
    System.out.println(a);
}
```

The code here uses standard arithmetical and assignment operators such as: `a = a + b;` and also the equivalent compound operators such as: `a += b;`

It displays this output:

```
12
14
12
10
20
40
20
10
```

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 and C++ programmers prefer the terser form as in `a += b`.

Increment ++ and Decrement -- Operators

When you want to increment or decrement by 1 (that is, add 1 *to* or subtract 1 *from*) the value of a variable, you may 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 or '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 of that variable is incremented *before* any assignment is made:

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

When placed *after* a variable, the value is incremented *after* any assignment is made:

```
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

Programs often need to deal with collections of items. In this chapter we'll look at some ways of dealing with everything from arrays of integers to HashMaps of user-defined objects.

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. As in most other programming languages, the simplest way of creating a collection in Java 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 directly after the type name, however, and that is the convention I adopt in this book.

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.

Arrays are Zero-Based

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', would be at the 'array index' 0, which we can write in code as `chararray[0]`, the second character, 'e', would be 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 *minus* 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), minus 1.

Indexing Errors

Be aware that while it is permissible to place fewer objects into an array than the maximum declared size of the array, 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 last element is at index 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 first declared the array variables and then created the 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 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 a 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:

Arrays

```
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]);
}
```

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.

The length Property

In the *Arrays* sample project, `length` is a 'property' or 'attribute' of the array. It stores the array size. In my code, `stringarray` has been initialized with six elements so `stringarray.length` evaluates to the number 6.

for Loops

When you want to run some code repeatedly, you could use a `for` loop to run one or more lines of code for a predetermined number of times. A `for` loop is defined by three parts, between parentheses – the *initialization*, *test* and *increment* parts – separated by semicolon. This is the basic syntax:

```
for (initialization; test; increment) {statements}
```

`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
`{statements}`	The code that executes at each turn through the loop is placed between a pair of curly brackets.

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

ForLoops

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

`int i = 0;`	The first part initializes the value of a counter variable, the integer `i`, which is assigned the value 0.
`i < 10;`	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. As soon as it has the value or 10 or more, the loop ends).

`i++`	The third part increments the value of the counter variable: `i++` adds 1 to the value of `i`.

The *statements* part is any code enclosed between the curly brackets. Here that is just this one line which simply prints out the value of `i`:

```
System.out.println(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 previous for loop 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'. This is the output produced by the loop above:

```
1
2
3
4
5
6
7
8
9
10
```

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

```
s = "Hello world";

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

In this case I use the loop to iterate over the characters in the string s which has been initialized with "Hello world". Remember that the first character in a string (or array) 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. 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`.

The value of `i` increases by 1 every time the loop runs, `i++`. 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 than 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 less than the length of the string.

Collections

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 an abstract class. It defines the methods that a class must implement but it does not contain the code of those methods. A Java class is able to implement an interface by providing the code for each of the methods defined by the interface. The `Collection` framework contains classes such as `ArrayList` which implements `Collection` interfaces such as the `List` interface. Interfaces are explained more fully in Chapter 8.

ArrayList

Java's `ArrayList` is a useful general-purpose class that lets you create resizable sequential lists. 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 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 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));
```

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). 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. This is called 'garbage collection'. Once upon a time, garbage-collecting languages were rare. These days, many languages ranging from C# to Ruby provide garbage collection.

Typed ArrayLists

While it is permissible to store elements of mixed types – such as both integers and strings – in an untyped `ArrayList` object, you can also ensure that only elements of a specific type are allowed. In order to do this, you need to place the type name, such as `Integer` or `String`, between a pair of angle-brackets after the `ArrayList` class name when you declare the variable, like this:

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

When you create a *typed* `ArrayList` object you need to place an empty pair of angle brackets before the parentheses of the `ArrayList` constructor:

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

The empty pair of angle brackets `<>` is sometimes called the 'diamond operator'. Instead of using empty angle-brackets when creating an `ArrayList` object, some programmers like to repeat the element-type name between the angle brackets:

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

In early versions of Java, this syntax was obligatory. In modern versions of Java (from Java 7 onwards) it is optional but not required.

Having created the `stringList` object as an `ArrayList` typed to hold strings, you can now use the methods of `ArrayList` to add, get or remove string elements:

```
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 behavior needed to operate on lists of different types of element.

In other words, the list-manipulation code is 'general purpose' – capable of operating on many types of object – and you are able to use it with specific types of object by specifying a type name between angle-brackets. Just as I created an `ArrayList` object to store `String` elements, I could create another `ArrayList` object to store `Integer` elements 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));
```

Generics Work with Objects only!

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 `String` objects, I cannot add strings to an `ArrayList` typed to hold `Integer` objects, 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 (see the Appendix). Here I'll just mention two more 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());
```

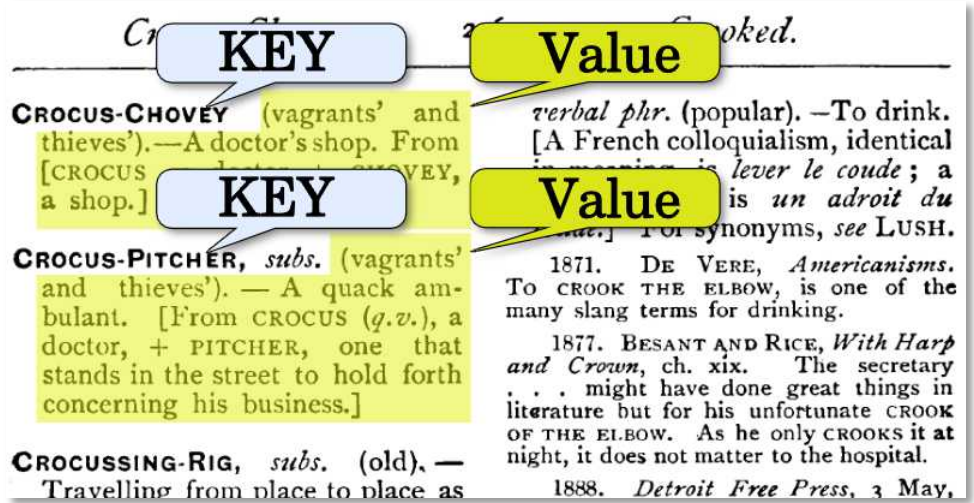
As always, I recommend that you run my sample code (here the *ArrayLists* project) in order to understand how this works. I'll be exploring generic lists in more detail in Chapter 9.

Maps and Dictionaries

Sometimes it may be useful to index a collection of objects by something other than their numerical positions in a list. For example, in a dictionary it is 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`.



CROCUS-CHOVEY (vagrants' and thieves').—A doctor's shop. From [CROCUS
a shop.]

CROCUS-PITCHER, *subs.* (vagrants' and thieves'). — A quack ambulant. [From CROCUS (*q.v.*), a doctor, + PITCHER, one that stands in the street to hold forth concerning his business.]

CROCUSSING-RIG, *subs.* (old). — Travelling from place to place as

verbal phr. (popular). —To drink. [A French colloquialism, identical
lever le coude ; a
is *un adroit du*
synonyms, *see* LUSH.

1871. DE VERE, *Americanisms.* To CROOK THE ELBOW, is one of the many slang terms for drinking.

1877. BESANT AND RICE, *With Harp and Crown*, ch. xix. The secretary . . . might have done great things in literature but for his unfortunate CROOK OF THE ELBOW. As he only CROOKS it at night, it does not matter to the hospital.

1888. *Detroit Free Press*, 3 May,

Even though we'll be using 'maps', you may find it easier to think of them as 'dictionaries'. In a conventional dictionary, like the one shown above, you can find a definition by looking up a word. Each word is unique but the definition for each word may be repeated. For example, the words 'woof' and 'bark' are unique but the definition, 'Sound made by dog', might be repeated. In programming terms, the word you look up would be the 'key' and the definition would be the 'value'.

HashMap

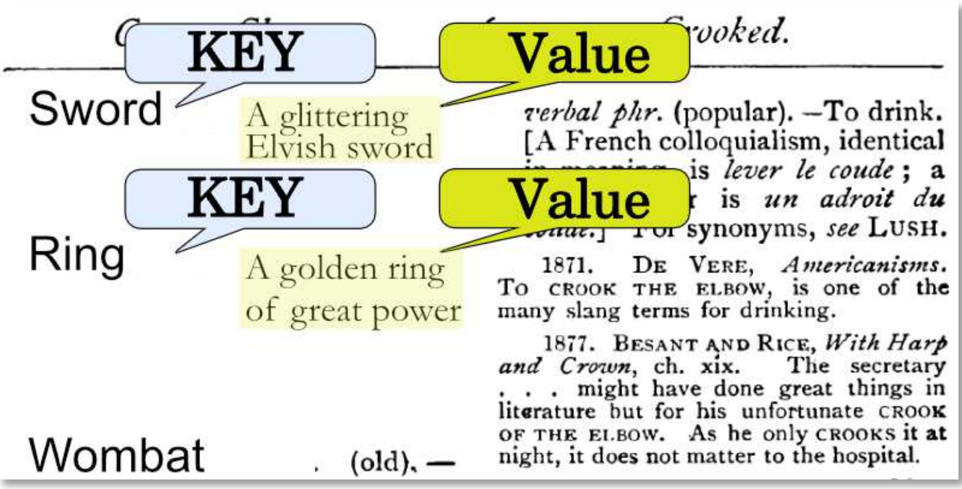
Just as `ArrayList` is a useful general-purpose implementer class of the `List` interface, `HashMap` is a useful general-purpose implementer class of the `Map` interface. For a simple example of using `HashMap` to store a collection of elements in the form of key-value pairs, take a look at the *Maps*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

Maps

```
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 of strings, such as "sword" is the 'key' and the second item such as "A glittering Elvish sword", is the 'value'.

Unlike a dictionary of language, the keys in our programming dictionary don't have to be in alphabetical order. However, they do have to be unique. There should be only one key "sword". I can retrieve an element using its key to give me access to the key's associated value. 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:

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

As my `HashMap` object `hmap` is *untyped* it is able to store elements in which the key and the value may be any type of object. Here 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]
```

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 has an `Integer` key and a `String` value I would use a declaration similar to the following:

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

Now I can use the 'diamond operator' `<>` when creating the object:

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

Declaring an Object Using an Interface

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<>();
```

When an object (such as `HashMap`) is declared in this way it will be compatible with the named interface (such as `Map`). This might be useful when, for example, you want to pass that object to a method that is capable of handling *any* object that implements the `Map` interface (not just `HashMap` objects).

Creating and Initializing a Typed HashMap

You can find an example of a typed `HashMap` in the *RoomMap* sample program. At the bottom of the *RoomMap.java* code file, I have defined 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. In this example, the `Room` objects do not have a `name` field. That's because the map will be used to store the room names.

When I create a new `Room` I add its name as a key in the `Map` collection. In order to ensure type safety (so that only strings can be used as keys and only `Room` objects can be used as values), 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));
```

In order to obtain the value (the Room object) associated with a specific key (a room name), I pass the key to the get() method like this:

```
aKey = "Troll Room";
aValue = map.get(aKey);
```

But sometimes I may want to do some operation on all the keys or on all the values. In this case, I want to display all the room names and all the exits of each Room object. In the code below I use a special type of for loop (which I'll have more to say about in Chapter 7) which iterates over all the keys returned by the keySet() function and uses each key in turn to get the associated Room object, r. It then displays the key (the room name) and the exit values, n, s, w, e, of each Room object:

```
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 ));
}
```

This is the output:

```
Troll Room     has exits:  N = -1 S = -1 W = 0  E = -1
Narrow Passage has exits:  N = 3  S = -1 W = 4  E = -1
Twisty Maze    has exits:  N = -1 S = 5  W = 2  E = -1
Dark Cave      has exits:  N = -1 S = 2  W = -1 E = 1
White House    has exits:  N = 2  S = -1 W = -1 E = 5
Dragon's Lair  has exits:  N = 0  S = 4  W = -1 E = 3
```

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 the keys of a map form a 'set'. The values do *not* form a set, however, because the same values may be associated with more than one key – as in the example I gave earlier: the keys, "woof" and "bark" share the same value: "Sound made by dog".

When using a `HashMap` to implement the map in a game, it might 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 index 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 need to perform the same task many times. Java lets you do this using several different types of loop. In this chapter we'll see how these loops work.

Computer programs often have to perform repetitive actions such as printing a series of strings until there are no more strings left to print or counting the number of treasures which a game-player has collected. Probably the most common type of loop – and one that we've already used in previous chapter – is the `for` loop.

for

As explained in Chapter 6, the traditional type of `for` loop is declared with three parts: *initialization*, *test* and *increment*:

```
for (initialization; test; increment) {statements}
```

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, `i < 10`). When it evaluates to `false` (here when the value of `i` is *not* less than 10), 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 the 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:

ForLoops2

```
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 in which `for` loops are written – with a loop variable incremented from 0 until some end value is met – 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 a `for` loop is used to iterate through the items in a linear list such as an array or an `ArrayList`. This next loop prints the items in the `somewords` array of strings from 0 to 2.

That is because the test executes the code inside 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
```

for...each

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:

RoomMap

```
for(String key: map.keySet())
```

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 follows:

```
for (type variableName: collection) {statements}
```

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

ForLoops2

```
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 elements to iterate through – from 0 to 4, say. But sometimes you may not know in advance how many elements you need to process.

For example, you might write some code to format the 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 dealing with this problem 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) {statements}
```

There are some basic examples of `while` loops in the *WhileLoops* sample program. Here I create an array, `intarray`, of five integers and I 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:

WhileLoops

```
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`.

This program tries to read text from a file named '*test.txt*' in the directory *C:/temp*. If the file is not found an error message is displayed. But, if you create a text file with that name, it will read its contents. It 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. Let's assume that that *C:/temp/test.txt* contains these lines:

```
This is a file
containing three
lines of text!
```

This is what you will see when you click the 'Read File' button:

```
[1] This is a file
[2] containing three
[3] lines of text!

Total lines in file = 3
```

Read 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 block of code 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 continue. 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 {statements} while (condition);
```

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

WhileLoops

```
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 executes, that code is bound to run once (before the test is done). In this case, this code displays the integer at index 2 in `intarray`:

```
3
```

Compare that with the `while` loop that I showed earlier, which *began* with: `while (i < 2)`. This test is done *before* the code inside the loop runs. So when `i` is 2 the test evaluates to `false` and any code inside the loop, *never* executes.

Multidimensional Arrays

In the last chapter, we looked at simple linear arrays that contain simple lists of elements. In addition to linear arrays you can also create 'multidimensional arrays' – that is, arrays of 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, like a row of cells in a spreadsheet:

	A	B	C	D	E
1	1	2	3	4	5

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 or matrix like the rows and columns of cells in a spreadsheet:

	A	B	C	D	E
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15
4	16	17	18	19	20

You will find an example of a two-dimensional array in the *MultidimensionArray* sample project. I declare an array variable called `grid` and show that its elements are arrays of integers by putting the element type `int` before the variable name and two pairs of square brackets after the variable name:

```
int grid [][]
```

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. I initialize the arrays by placing one set of curly brackets to delimit the outer array: `int grid[][] { }` and then I add 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, similar to the spreadsheet I showed on the previous page.

If you want to iterate over the *outer* array you could use a simple `for` loop. 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 using the `toString()` method of the `Arrays` class. But how can I access the individual integers in each of these four arrays?

One way would be to use another `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 a second `for` loop to iterate through the elements (the integers) which each of these *inner* arrays contain.

To access a specific piece of data I need to give its location in the matrix. In a spreadsheet this would the 'cell location' defined by the intersection of the row (for example, Row 3) and the column (for example, column B). Here that intersection (B3) identifies the cell containing the number 12:

	A	B	C	D	E
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15
4	16	17	18	19	20

In a Java program, we can find the value of an element in a two-dimensional array by using a numeric index into the *outer* array, in order to find one of the inner arrays (like a row in a spreadsheet) and then use another index to count along the elements in that *inner* array (like a column in a spreadsheet).

Remembering that arrays are indexed from 0, you can see that in order to get at the integer 12 I would need to index into row 2 and column 1. Here I've numbered the columns to show this:

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15
3	16	17	18	19	20

Using square-bracket notation with the 'row' index in the first pair of brackets and the 'column' index in the second pair of brackets, this is how I would print out the value stored at row 2 and column 1:

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

This displays the following:

```
Value at grid[2][1] = 12
```

If I want to display the value at row 1 (the second array in my `grid`) and column 2 (the third element in that array), this is what I would write:

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

This displays the following:

```
Value at grid[1][2] = 8
```

Now let's return to the problem of displaying every integer in the `grid`. First I need one `for` loop 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. In order to iterate over the elements (the integers) stored in each nested array, I need another `for` loop *inside* the first `for` loop, like this:

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

Remember that this is the 'array of arrays' we are dealing with – the variable `grid` which contains four arrays, each of which contains five integers, just like the spreadsheet I showed earlier:

MultidimensionalArray

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

This can be visualised as a matrix with rows indexed from 0 to 3 and columns indexed from 0 to 4:

	0	***1***	***2***	***3***	***4***
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15
3	16	17	18	19	20

In my sample code, I iterate over all the items stored in the `grid` two-dimensional array using one for loop to iterate over the 'rows' (the four array elements stored in `grid`) and I display the 'row' number (the index of the array) followed by the 'column' numbers (the index of the integers in that array):

MultidimensionalArray

```
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
```

Here the inner or 'nested' arrays are all of equal length which makes it easy for me to iterate over the same number of integers in each of the inner arrays. In fact, it is not a requirement to make nested arrays of equal length. The nested arrays, like the rows in a spreadsheet, might not contain the same numbers of elements, like this (notice that there are different numbers of cells containing data in each row):

	0	1	2	3	4	5
0	1	2	3			
1	4	5	6	7		
2	8	9				
3	10	11	12	13	14	15

In code, this is how I declare an array `intgrid` containing four arrays of varying length:

MoreMultidimensionalArrays

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

I make use of the `length` property of the arrays. I first find the number of array elements in `intgrid` using `intgrid.length` and I use that value in the outer loop. For each array processed in the inner group, I find the number of elements using `intgrid[row].length`:

```
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]);
    }
}
```

This is the output:

```
There are 4 rows.
--- row 0 ---
index[0], value=1
index[1], value=2
index[2], value=3
--- row 1 ---
index[0], value=4
index[1], value=5
index[2], value=6
index[3], value=7
```

```
--- row 2 ---
index[0], value=8
index[1], value=9
--- row 3 ---
index[0], value=10
index[1], value=11
index[2], value=12
index[3], value=13
index[4], value=14
index[5], value=15
```

break

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 this code:

BreakAndContinue

```
private void breakBtnActionPerformed(java.awt.event.ActionEvent evt) {
    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. Test this by running the program and clicking the 'break' button. The code following `break` is not run so this is what is displayed:

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

Breaking Code Logic!

In a 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 (this is the 'logic' of the loop condition). When you add a `break` you are breaking the 'code logic' of the loop. As a general principle, it is better, 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)` as explained in Chapter 3 and it appends this character to another string variable `str`:

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

Suffice to say, this is a very simple (and not very secure!) encryption algorithm. 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 loaded from a file. For simplicity, I have created the string shown below, in order to illustrate how the program works:

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

I have 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;
}
```

This is the complete encryption routine:

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

So what does the `continue` statement do after the first `if` test? Let's find out.

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:

```
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);
}
```

This translates the *encrypted* string `str` (by subtracting 1 from each character) back to an 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 `breaks` when the string terminator `'!'` is found. If you run the program and click the 'Encrypt' button, this is what you will see:

```
Encrypted string is 'Ifmmp xpsme'

Decrypted string is 'Hello world'
```

Notice that the space between "Hello" and "world" has been retained by the '!' character and all the characters following it in the string "Hello world! Goodbye" have been removed.

Look at the code carefully and try to work out why that has happened. If this isn't clear to you, try editing the string `testStr` by removing the '!' character or adding more of them and see what results you get when you click the 'Encrypt' button:

```
// try this
String testStr = "Hello! world! Goodbye";

// and this
String testStr = "Hello world Goodbye";
```

The *BreakAndContinue* project also contains an example of breaking out of a `while` loop (see the `whileBtnActionPerformed()` method):

BreakAndContinue

```
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));
}
```

This code 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:

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

In common with the `break` statement, the use of `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. When I insert an additional test that breaks out of the loop when some other condition is met such as `if (c == '!')` this violates the condition controlling the `while` loop. In many cases, you can avoid breaking from loops by rewriting the test condition of the loop itself, like this:

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

Breaking Out of a 'for' Loop

In a previous example program *MultidimensionalArray*, you may recall that I placed one `for` loop inside another in order to process the elements in arrays that were nested inside another array:

MultidimensionalArray

```
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]);
    }
}
```

This was the output:

```
--- 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
```

Now you might wonder what you would need to do to break out of the inner `for` loop?

```
for(some condition){
    for(some other condition) {
        break;
    }
}
```

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 (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;
        }
    }
}
```

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

```
--- row 0 : [1, 2, 3, 4, 5] ---
column[0] = 1
column[1] = 2
column[2] = 3
--- row 1 : [6, 7, 8, 9, 10] ---
column[0] = 6
column[1] = 7
column[2] = 8
--- row 2 : [11, 12, 13, 14, 15] ---
column[0] = 11
column[1] = 12
column[2] = 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 both the inner *and* the outer loop you can do this using a labelled `break` statement. You should add a label (that 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++) {
```

In order to break to this label (which would exit *both* the *inner* and the *outer* loop) you add the label name (without a colon) after `break`:

```
break outerloop;
```

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

LabelBreak

```
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
```

Labelled continue

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

```
continue outerloop;
```

Previously the unlabelled `break` exited the *inner* `for` loop when the loop variable `column` had the value 2 and then it 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 (if you look at the sample code, you'll see there are four nested arrays) is processed.

Label Positions

There are precise rules on where you can put the label for a labelled break or continue. Labels are used for breaking out of `switch`, `for`, `while`, or `do-while` statements by exiting the outermost statement. The label must appear *directly* above the outermost statement, with no other code between the label and that statement.

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. It is a sort of `goto` statement that suddenly jumps out of a block of code and starts running code somewhere else. That sort of jump can sometimes be useful – when a critical condition means that you need to stop running a piece of code as fast as possible – but it can also be confusing and it risks introducing accidental bugs.

For that reason, 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 are some occasions when it is useful to break from a loop and you certainly need to understand how `break` and `continue` are used, but don't be too enthusiastic about using them. As a general principle, you should avoid breaking out of loops if there is a simpler, and less confusing, alternative.

Chapter 8 – Enums, Interfaces and Scope

In this chapter I'll explain how enumerated constants can help code clarity, how interfaces can predefine the behavior of classes and why the position of a piece of code is important.

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");
}
```

Compare with this code:

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

As far as the computer is concerned the two code fragments above mean the same thing; when your program runs, they will produce exactly the same results. But from the perspective of the programmer, the second example, which uses named identifiers such as `HEARTS` and `DIAMONDS` instead of numbers such as 0 and 1 is certainly much clearer. In order to use identifiers like this, you need to create an `enum`.

Enum Types

To create series of identifying names such as `HEARTS`, `DIAMONDS`, `SPADES` and `CLUBS` and group them together as `Suits`, you need to define 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'. By convention, constant identifiers are normally written all in uppercase. That isn't obligatory – you could call them Hearts, Diamonds, Clubs and Spades if you wished, but all-capital identifiers are usual.

In these example, Suits and Months are the enum type names. 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 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* project:

Enums

```
public class 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;
```

Enums (continued)

```
        }
        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.
```

Enum.toString()

Notice that I have used the `toString()` method to display the actual name of an `enum` constant! The `toString()` method is defined for many standard Java classes including the `Enum` class.

Assigning Values to enum Constants

For another example of `enum`, take a look at the *Adventure2* project (in the *Adventure* folder of the source code archive). 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`. 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 (Direction.java)

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

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

Compiling Adventure2

When you run the Adventure2 project for the first time (or if you select (*Run|Clean* and *Build Project* in NetBeans) you may see some warnings of an "unchecked call to add(E)". That is because the map `ArrayList` in this project can hold any type of object and Java is reminding me that it would be safer if I created an `ArrayList` to hold only a specific object type (here my map will only need to hold `Room` objects). If you look at the code of Adventure3 you'll see that I have declared the `ArrayList` like this:

```
private ArrayList<Room> map;
```

When declared like this, the map `ArrayList` will only be able to store `Room` objects. Java can perform type-checking when I try to add an element to ensure that I don't accidentally add some other type of object. Adding type-checking to an `ArrayList` is an example of 'generics' which is explained in Chapter 9.

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:

Adventure2 (Game.java)

```
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;
}
```

I have deliberately made `moveTo()` capable of moving 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.

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.

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:

Adventure2 (Game.java)

```
map.add(new Room(
"room0", "You are west of a 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;
```

I can't use the identifiers `NORTH`, `SOUTH`, `WEST` and `EAST` to initialize exits when creating `Room` objects because each of the exit fields is an `int`, not a `Direction`. But I can use the `NOEXIT` constant to initialize one of the exit fields because I have declared `NOEXIT` to be an `int` and I have assigned to it the `int` value -1. I've chosen -1 because I can be sure that no room in the map will have that index.

The Enum Class

In some programming languages, an enum is 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 a 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.

The Planet Enum

If you want to explore some of the more esoteric features of Java enums, look at Oracle's sample code which I've copied into the *Planet* project. To run this code, compile the program, either from the command prompt, as explained in Chapter 2 or (more simply) build and run it from within an IDE such as NetBeans. You should now have the compiled class file, *Planet.class*, in the *\planet* folder beneath the *\classes* folder. If your code archive is stored in the *\LittleBookOfJava* directory on the *C:* drive, the full path (on Windows) will be: *C:\LittleBookOfJavaCode\Step08\Planet\build\classes*

At the command prompt enter this:

```
java planet.Planet 175
```

This passes the integer 175 to be run by the Planet program. This is what is displayed:

```
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
```

```
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

Alternatively, you can build and run the project direct from NetBeans. Just make sure that the argument 175 is defined. To do that, right-click the *Planet* project in the *Projects* window and select *Properties*. In the dialog, select *Run*. Make sure 175 is entered into the *Arguments* field.

The code of this project shows how an `enum` can implement many of the features of other Java classes. Unlike regular classes, however, you cannot create objects (new instances) from an `enum`. For a detailed explanation of the `Planet` class, see the Oracle Java tutorial:

https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

You may also want to refer to the JDK entry on `Enum`:

https://docs.oracle.com/javase/10/docs/api/java/lang/Enum.html

Complex Enums?

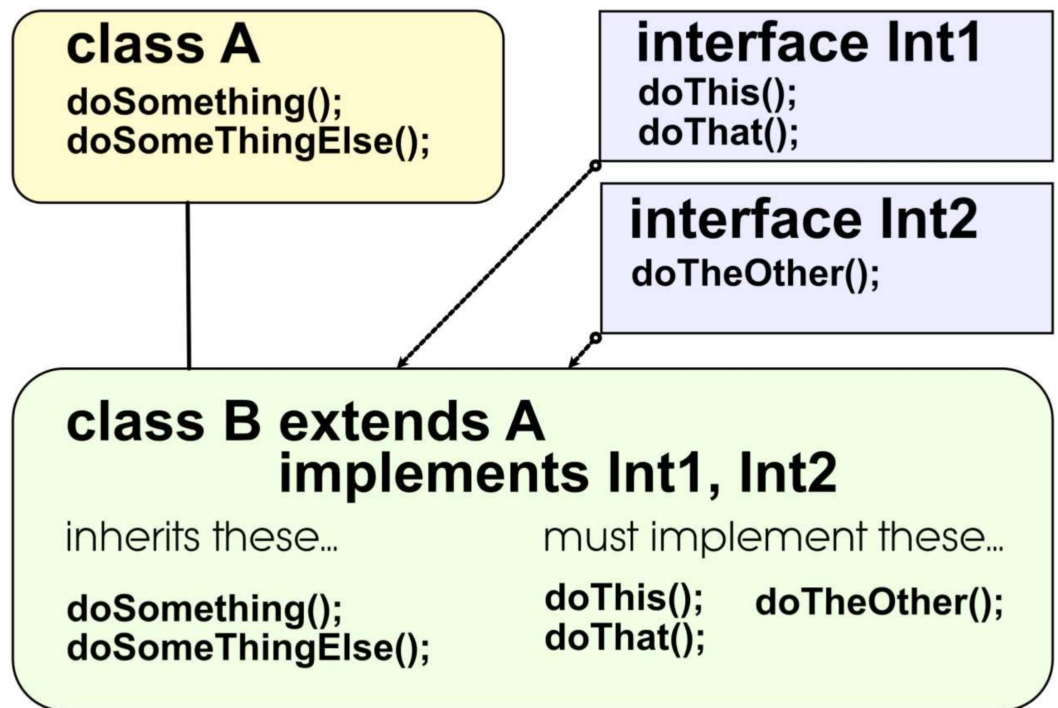
While the kind of complex `enum` implemented in the *Planet* project may have a use in some special cases, many programmers (myself included!) prefer to stick with the traditional type of `enum`, broadly similar to those in C, which simply groups together a list of constants to assist in coding clarity.

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.

Extending and Implementing

A class may extend one other class and implement zero, one or more interfaces.



The diagram above shows a simplified view of a class, `class B`, which *extends* (so it is the descendent of) of `class A` and *implements* the two interfaces `Int1` and `Int2`.

The `class A` contains the code of the methods `doSomething()` and `doSomethingElse()`, and `class B` automatically *inherits* those methods.

The two interfaces, however, only *define* the methods `doThis()`, `doThat()` and `doTheOther()` – *they do not contain the code of those methods*. So `class B` must now implement (that is, the programmer must write the actual code of) those three methods.

In other words, when `class B` *extends* `class A` it inherits the fully-working methods of `class` A. But when `class B` *implements* the interfaces `Int1` and `Int2` it does not gain access to existing methods. Instead, it makes an agreement or 'contract' to write code for the methods that are defined (but not implemented) by those interfaces.

Interfaces as 'Contracts'

An 'interface' is a definition of the code that a class must implement: it is, if you like, a 'contract' that the class using the interface must respect. If you have a contract to build a house, say, the contract isn't the actual house, but it does specify (in more or less detail) what will be built and to what standards the house will be built.

To see an example of a class that implements some interfaces, look at the `ArrayList` class which 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 one class descends from some other class (its superclass) it automatically inherits the behavior – the methods – from the superclass.

In addition to 'extending' `AbstractList`, the `ArrayList` class also 'implements' `List`, `RandomAccess`, `Cloneable` and `Serializable`. 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 the definition of a class *without* executable code. Each interface may define a set of constants and methods but it does not provide the behavior – the actual 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 `List` interface defines the `size()` method like this:

```
int size();
```

Above this declaration there is this comment block, documenting `size()`:

```
/**
* Returns the number of elements in this list.  If this list contains
* more than <tt>Integer.MAX_VALUE</tt> elements, returns
* <tt>Integer.MAX_VALUE</tt>.
*
* @return the number of elements in this list
*/
```

Finding Class and Interface Code

You may be able to use the 'search' tools of your IDE to find interface declarations and documentation. In NetBeans, in one of my programs that uses an `ArrayList`, I double-click `ArrayList` to select it then right-click and select *Navigate|GoToSource*. This takes me to *ArrayList.Java*. The declaration of that class shows me that it implements the `List` interface. I right-click `List` and again select *Navigate|GoToSource* to view *List.java*.

This is how the `ArrayList` class implements `size()`:

```
public int size() {
    return size;
}
```

If you search the code of *ArrayList.java* you'll see that `size` is a private `int` variable and its value is changed by various methods. For example, the `add()` method, unsurprisingly, increments the value of size:

```
size++;
```

If you were to write a `TreasureList` class, this 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.

Custom Interfaces

Let's see how you might go about writing your own interface. Let's suppose that you are leading a team of programmers developing a game. The game defines 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 both store a value and they need to have methods that can add to or subtract numbers 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 in the `myinterfaces` package:

Interfaces (myinterfaces.ValueableItem.java)

```
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 {
```

You can see the complete code of my `Treasure` class on the next page. 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.

I said earlier, that one interface can be implemented by many classes and that the `ValuableItem` interface which is implemented by `Treasure` might also be implemented by another class such as `Score`.

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

Interfaces (advobs.Treasure.java)

```
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;
}
```

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 (no executable code) 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 behavior of the abstract class's methods must then be programmed in the subclass.

An abstract class is a bit like a regular class (it can contain fully implemented methods and it can be subclassed) and it is a bit like an interface (you cannot create objects from an abstract class but you can define 'empty methods' which must be implemented in subclasses of the abstract class). Refer to Oracle's documentation for more information:

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

Abstract Classes Or Interfaces?

There may be occasions when interfaces may be useful as a way of defining 'templates' to ensure that a number of classes (those which implement the interface) all have certain specific methods which are coded differently for each class. An abstract class may be useful as a 'base' class that provides both a set of common methods and also 'method templates' (similar to those defined by an interface) to a number of classes. If, however, you are unsure of the need for interfaces or abstract classes in your own code, don't use them. They are not fundamental to good object oriented programming though they may sometimes be of use.

An Abstract Class Example

In Chapter 4 I explained how to create a simple class hierarchy (the *Objects* project) in which a `Thing` class defined a `name` and a `description`. A descendent class, `NoisyThing`, added on a `noise` and a `volume`.

I was then able to create treasure objects from the `Thing` class and cat and dog objects from the `NoisyThing` class. But let's suppose I now want a dedicated `Treasure` class which descends from `Thing` and adds on a `value`. Both `Treasure` and `NoisyThing` inherit `name` and `description` from their common ancestor `Thing`. I also want both these classes to implement their own `describe()` methods. Since I will never create objects directly from `Thing`, I decided to make `Thing` an abstract class that provides a simple implementation of `name` and `description` but makes `describe()` *abstract* because that method will be implemented differently in the descendent classes. This is my `Thing` class:

AbstractClasses (Thing.java)

```
public abstract 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;
    }

    abstract public String describe();
}
```

The important things to notice here is that the class itself is defined as `abstract`, so objects cannot be created from it (though they can be created from its descendants) and the `describe()` method is also abstract. This method contains no code – the descendent classes will each need to code their own `describe()` methods. This is the `Treasure` class:

AbstractClasses (Treasure.java)

```
public class Treasure extends Thing {

    private int value;

    public Treasure(String aName, String aDescription, int aValue) {
        super(aName, aDescription);
        this.value = aValue;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    @Override
    public String describe() {
        return "The " + getName() + " is "
                + getDescription() + " and is worth "
                + getValue() + " groats\n";
    }
}
```

Note that this implements its own version of `describe()`. So does the `NoisyThing` class:

AbstractClasses (NoisyThing.java)

```
public class NoisyThing extends Thing {
    private String noise;
    private int volume;

    public NoisyThing(String aName, String aDescription, String aNoise,
                      int aVolume) {
        super(aName, aDescription);
        this.noise = aNoise;
        this.volume = aVolume;
    }

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

AbstractClasses (Treasure.java) (continued)

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

    public int getVolume() {
        return volume;
    }

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

    private String noiseLevel() {
        String vol;
        if (volume > 9) {
            vol = "really noisy";
        }else if (volume > 4){
            vol = "fairly noisy";
        }else if (volume > 0){
            vol = "quiet";
        }else{
            vol = "silent";
        }
        return vol;
    }

    @Override
    public String describe() {
         return getName() + " is " +
                getDescription() + ". It is " +
                noiseLevel() + "\n";
    }

}
```

I can no longer create `Thing` objects. But I can create objects from classes which descend from `Thing`, such as `Treasure` and `NoisyThing` objects:

AbstractClasses (NewJFrame.java)

```
mySword = new Treasure("Dwarf Sword", "a sword of great power", 100);
myRing = new Treasure("Elvish Ring", "a golden ring of magic power", 150);
cat1 = new NoisyThing("Tiddles", "a small ginger cat", "", 0);
cat2 = new NoisyThing("Flossie", "a small tabby cat", "Miaaoow", 3);
dog1 = new NoisyThing("Bert", "a small terrier", "Wuff!", 5);
dog2 = new NoisyThing("Fido", "a big smelly dog", "Woof!", 10);
```

And when I run the program and click the button, this is what I see:

```
The Dwarf Sword is a sword of great power and is worth 100 groats
The Elvish Ring is a golden ring of magic power and is worth 150 groats
Tiddles is a small ginger cat. It is silent
Flossie is a small tabby cat. It is quiet
Bert is a small terrier. It is fairly noisy
Fido is a big smelly dog. It is really noisy
```

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:

Scope

```
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
}
```

If I click the second button on the form to call `methodC()`, this is what I see:

x = 10
x = 100
x = 10

Scope methodC()

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 (Game.java)

```
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 using a dot after the variable name (such as `x.toString()`) 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);
```

Here, for example, I call this method from the code of the `nBtnActionPerformed()` method in *AdventureForm.java*, passing the `int` value that is returned to another method called `updateOutput()` which displays some text:

Adventure2 (AdventureForm.java)

```
updateOutput(game.movePlayerTo(Direction.NORTH));
```

I am able to call `movePlayerTo()` here because it is a *public* method. I cannot call a *private* method (such as `moveTo()`) of the `Game` class from within *AdventureForm.java*, however:

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

This is what NetBeans shows:

```
Generated Code
moveTo(Actor,Direction) has private access in Game
----
(Alt-Enter shows hints)
rmed(java.awt.event.ActionEvent evt) {
ePlayerTo(Direction.NORTH));
        updateOutput(game.moveTo(game.getPlayer(), Direction.NORTH));
    }
```

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'.

Keep it Simple!

Just because all those access modifiers are available doesn't mean that you have to use them all. The two most important access modifiers are `public` and `private`. In most cases, you will be able to write good, well-structured, object oriented code using only those two modifiers to keep data fields `private` within an object and to restrict access to methods which you don't want to be used by code outside that object. Methods that you *do* want to be available from outside the object must be `public`.

Chapter 9 – Generics and Exceptions

Generics can help when you need to perform similar operations on different types of objects in your Java programs. Exceptions can help when things go wrong!

Maybe you want to create lists of objects and have the ability to add or delete items from that list. It is likely that you will want to do similar operations on lists of integers, floats or strings. Since those data types are represented by different classes, you might think you would need to write the same methods three times to handle lists of the three different types. Thankfully, generics can help you to avoid all that repetition. Imagine that you have an `ArrayList` named `aList` and you want to add elements to it like this:

Generics (NewJFrame.java) – stringListBtnActionPerformed()

```
aList.add("one");
aList.add(2);
aList.add("three");
```

Now you want to write a method that performs some action on the elements in that list. Let's suppose, for example, that you write a method to return the last element:

Generics (NewJFrame.java)

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

This method assumes that the last element 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 placed between parentheses:

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

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);
```

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

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

The program compiles without any errors. But when you click the button labelled 'string list' 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` must store 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 (or other programmers in your team) forget that this was what you intended and you (or they) 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 won't realize that it contains a bug. And then one day, something in your code just happens to put an integer at the final position – and that's when the error occurs.

One way to make sure that this cannot happen 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:

Generics (NewJFrame.java)
`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:

Generics (NewJFrame.java) – genericStringListBtnActionPerformed()
`gStrList.add(2);`

The compiler produces this message (this may be shown as an error 'hint' in an IDE):

```
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

Generics allow you to use placeholders or parameters when defining classes, interfaces and methods. These are similar to the usual formal parameters used in method declarations, but instead of being *values* they are *types*. Generics let you re-use the same code with different inputs – that is with different types of data.

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 `Integer` elements, 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 I 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:

Generics (MyGenericClass.java)

```
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. 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:

Generics (NewJFrame.java) – myGenClassActionPerformed()

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

In fact, since `MyGenericClass` 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");
```

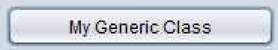
I could equally create `MyGenericClass` objects to work with other types such as `Integer`:

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

Now I can call the `describe()` method of `MyGenericClass` 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());
```

Click the 'My Generic Class' button to run this code:



This is what you will see:

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

Generic Parameter Names

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 actions 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 could either implement your list from the ground up or you could 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:

Generics (MyList.java)

```
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 `String` objects, `Integer` objects 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:

Generics (NewJFrame.java)

```
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):

Generics (NewJFrame.java) – genericStringListBtnActionPerformed()

```
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"):

Generics (NewJFrame.java) – myListIntegerBtnActionPerformed()

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

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 (Monster.java)

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

Overriding (Troll.java)

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

Now 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`:

Overriding (NewJFrame.java)

```
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 run 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' instead of the uppercase 'S' in `saysomething()`) has caused a bug which, in a more complicated program, might be very hard for me to find and fix.

@Override Annotation

There is a simple trick to help me avoid the sort of problem that would result from accidentally misnaming a method. 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 the annotated 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, 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 in the ancestor class 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:

```
method does not override or implement a method from a supertype
----
(Alt-Enter shows hints)
    @Override
    public String saysomething() {
        return "Ugh, ugh, ugh!\n";
    }
```

Now I can fix this by capitalising the 'S' so that the method `saySomething()` matches the method with the same name in the superclass and the error goes away:

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

Overloading Methods

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.

Similarly, a single class may have more than one constructor. The Java JDK documentation for the `String` class lists numerous constructor methods each of which defines a different set of arguments so that each constructor has a unique signature:

```
String()
String(byte[] bytes)
String(byte[] ascii, int hibyte, int offset, int count)
String(byte[] bytes, int offset, int length, String charsetName)
String(byte[] bytes, int offset, int length, Charset charset)
String(byte[] bytes, String charsetName)
String(byte[] bytes, Charset charset)
String(char[] value)
String(char[] value, int offset, int count)
String(int[] codePoints, int offset, int count)
String(String original)
String(StringBuffer buffer)
String(StringBuilder builder)
```

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:

Overriding (Troll.java)

```
public class Troll extends Monster{

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

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

When I call the `saySomething()` method with no arguments, the first of the two methods with that name will be executed. If I call `saySomething("I want to be fed")` passing a single string argument, the second version of the method will be used:

```
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 book. 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 you expect them to do. For example, in a calculator program, you may expect the user to enter a number such as 10 (the characters for one and zero) but in fact someone enters a string such as "IO" (the alphabetic letters, 'I' and 'O') .

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

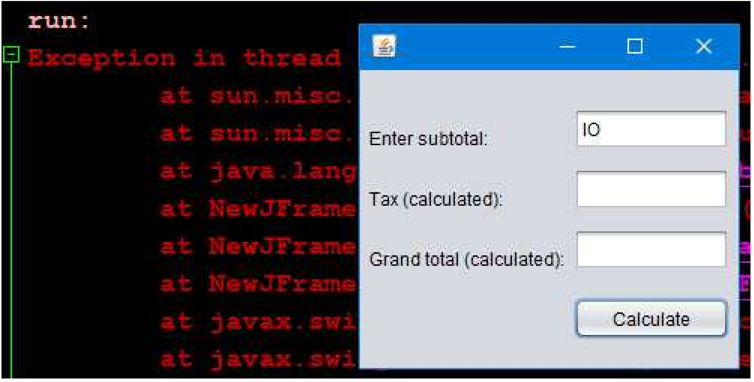
For an example of this, consider the *TaxCalc* project from Chapter 3. This prompted the user to enter a subtotal. It then used this value to calculate the tax due and the grand total (the subtotal plus tax). But what would happen if the user entered some text that could not be converted to a floating-point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

TaxCalc

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

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

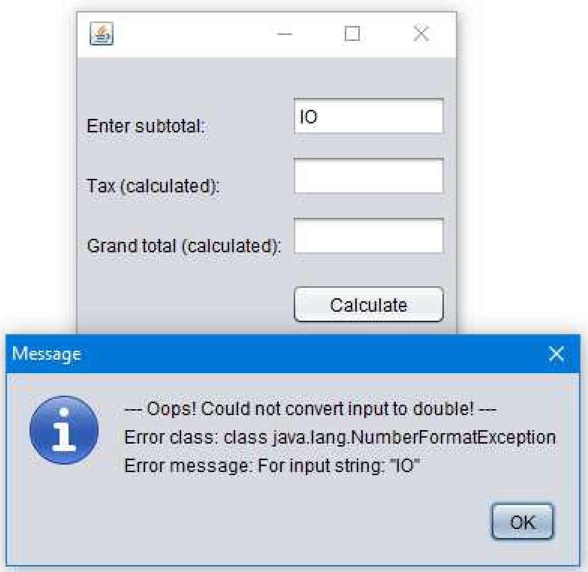
```
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException: For input
string: "ten"
    at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
    at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.lang.Double.parseDouble(Double.java:538)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



An exception is an object. When an error occurs at runtime, 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. 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 'Calculate' 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 civilized behavior 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;
```

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 of methods and properties. 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)
```

This is the complete code showing how I try to do a potentially 'dangerous' operation, catching an exception object if that operation causes an error and then running the rest of the code only if no error occurred (so the value of the `no_error` variable is true):

SafeTaxCalc

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

    double subtotal = 0.0;
    double tax = 0.0;
    double grandtotal = 0.0;
    boolean no_error = true;

    try {
        subtotal = Double.parseDouble(subtotalTF.getText());
    } catch (Exception e) {
        JOptionPane.showMessageDialog(this,
                "--- Oops! Could not convert input to double! ---"
                + "\nError class: " + e.getClass()
                + "\nError message: " + e.getMessage() );
        no_error = false;
    }
    if (no_error) {
        tax = subtotal * TAXRATE;
        grandtotal = subtotal + tax;
        taxTF.setText(Double.toString(tax));
        grandtotalTF.setText(Double.toString(grandtotal));
    }

}
```

Exception Types

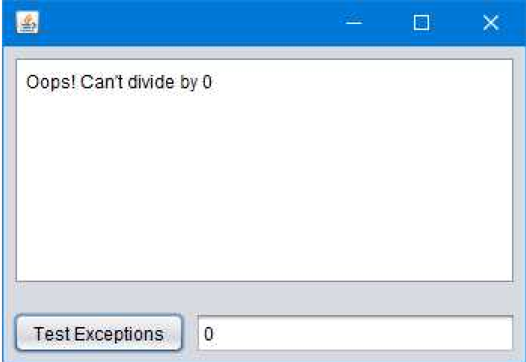
It is also possible to catch specific *types* of exception. Here is an example:

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 when a division operation is tried, 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 and the message "xxx is not a valid integer!" is shown.

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 type of exception such as an `ArithmeticException` or a `NumberFormatException` has already been caught, this last 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");
    }
}
```

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

More On Exceptions

There are many subclasses of the `Exception` class. I don't list them all here because there are so many of them and they are, in any case, already fully documented by Oracle. If you need to trap specific exception types, refer to the documentation of all available classes here:

https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html

Oracle also has online tutorial on Exceptions:

http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

Chapter 10 – Files and Serialization

In this chapter we shall look at some ways of dealing with files on disk and reading and writing data to and from files.

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. There are, in fact, many ways of writing and reading data to and from files using Java.

The *WriteList* project from Chapter 9 used a `PrintWriter` object to save lines of text to a file. `PrintWriter` is one of a number of 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 of linear sequences of bytes or as well-defined data-structures representing objects of various types. 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 enable 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:

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. 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.

File Pointers

A file pointer is not, as you might imagine, a pointer (the sort of direct memory reference that is common in the C language but not in Java) directly into a file on disk. What is referred to as a 'file pointer' is, in fact, a piece of data that stores a position or 'offset' into a file. The `JavaRandomAccessFile` class even has a `getFilePointer()` method which returns a `long` integer value which represents an offset (a number of bytes) in a file at which the next read or write operation occurs. When programmers talking about 'moving' the file pointer, they really mean updating the stored offset – thereby changing the 'position' which it refers to.

If you have saved 'chunks of data' or 'records' each of a fixed size, you can locate the start of a specific record 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. Each character has the size of 1 byte – so I can seek to a specific character position, like this:

RAFile

```
file.seek(20);
```

Now, from position 20, I can read in a specific number of bytes (where a byte is a piece of memory big enough to store a character). I've create a 20-byte array like this:

```
byte[] chars = new byte[20];
```

I read bytes into the `chars` array like this:

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

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 a new number (say 20) to the number of bytes that were read in previously (`bytesread`) in order to move the file pointer to this new offset (20 bytes after the end of the bytes that were read previously):

```
file.seek(20+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:

RAFile (NewJFrame.java) – readFileBtnActionPerformed()

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

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

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

When you close a file, any memory that was in use is cleaned up (in the jargon, any allocated resources are released) and the file is no longer available for reading and writing operations.

If you want to run this program, make sure you have a text file called '*test.txt*' in the *C:/temp/* directory. If it is in some other directory, be sure to edit this line in the source code (note that forward '/' is the standard directory delimiter in Java):

```
String pathName = "C:/temp/test.txt";
```

The '*test.txt*' file should contain a few lines of text. This is the text I've used:

```
I wandered lonely as a cloud
That floats on high o'er vales and hills,
When all at once I saw a crowd,
A host, of golden daffodils;
```

Now run the program and click the 'Read File' button. This is what it displays:

```
File pointer at start of file
chars = I wandered lonely as
File pointer at position #20
chars =  a cloud
That float
File pointer at position #40
chars = s on high o'er vales
File pointer at position #1000
End Of File!!! -- No more data to read!
```

Let's look at how this works. I begin by setting the file pointer to the start of the file, at position 0:

```
file.seek(0);
```

I read in 20 characters:

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

I assign those characters to a string, `s`:

```
s = new String(chars);
```

These are the first 20 characters which are displayed in the text area of my form:

```
I wandered lonely as
```

I now specifically set the file pointer to position 20:

```
file.seek(20);
```

I read and display another 20 characters starting from position 20:

```
 a cloud
That float
```

Here, the twenty characters include the space before the 'a' plus a carriage return character and a linefeed character after 'cloud'. Next I use the stored value, `bytesread` (which here is 20 because 20 bytes were read in the previous operation) and add 20 to it. That moves the file pointer to position 40 (right after the word "float"):

```
file.seek(20+bytesread);
```

This time when I read 20 characters I get this:

```
s on high o'er vales
```

I'll leave you to work through the code on your own. Try modifying the offset values or changing the size of the `chars` array to understand exactly how they work. You might want to rewrite the code so that a variable called `lastpos` accumulates the total values of bytes read (adding together the values of each file-read operation that is stored in the `bytesread` variable) so you always know the last accessed position in a file.

Be sure to use your IDE's debugger. For example, in NetBeans I can put a breakpoint by clicking in the numbered margin to put a red marker on a line of code so that execution pauses on that line:

```
117             }
                file.seek(20+bytesread);
119             ta.append("\nFile pointer at position #"+(20+bytesread));
```

I can then use the debugging tools to examine the values of variables. In NetBeans I start the debugger by selecting *Debug|Debug Project*. When I click the button on the form, the debugger stops at my breakpoint.

I can now hover my mouse over a variable name such as `s` or `chars` and 'drill down' to see the data they contain. I do that by hovering my mouse over the variable name until a 'debug hint' pops up:

```
file.seek(20+bytesread);
ta.append("\nFile pointe  chars = (byte[]) #1653(length=20)  read));
bytesread = file.read(chars);
```

I click the '+' sign in the debug hint to view the details. Here, for example, I can see that the first 20 bytes read from the file include the carriage return and linefeed characters (ASCII values 13 and 10) read into index 8 and 9 of the `chars` array:

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

Name	Value	Type
chars	#1652(length=20)	byte[]
[0]	32	byte
[1]	97	byte
[2]	32	byte
[3]	99	byte
[4]	108	byte
[5]	111	byte
[6]	117	byte
[7]	100	byte
[8]	13	byte
[9]	10	byte

Debugging

A detailed explanation of the debugging tools of any specific IDE (such as NetBeans) is beyond the scope of this book. You should, however, take the time to study your IDE's debugger in depth. Refer to your IDE's help, online documentation or user guide for more information. Debugging is invaluable for understanding how your programs work – or why they *don't* work!

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 (a storage area for data waiting to be processed) 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. Take my adventure game as an example. In order to save the game state, I would have to save each `Room` object from the map one by one, being sure to save each `Treasure` object that is present in each `Room`, and also the `Treasure` objects that are in the player's inventory.

Then, when I wanted to reload a previously 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 correct `Room` and reassemble the player's inventory (treasures that had been collected when the game was last saved).

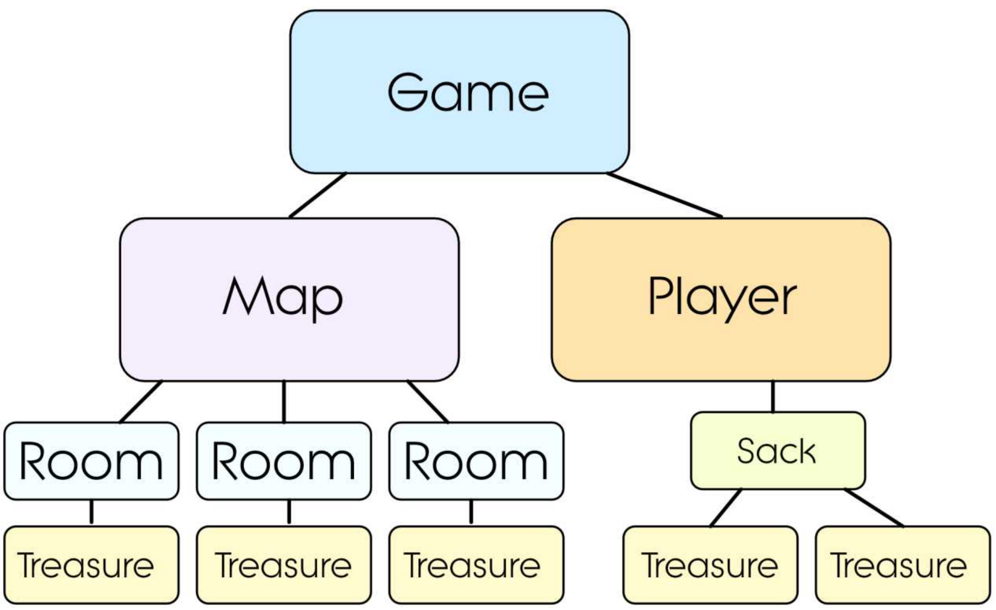
Each time the game is played, the game 'state' would change. That means that the positions of all the `Treasure` objects would need to be saved. And when the game is reloaded, those treasures would need to be put back into the correct places. This sounds like a programming nightmare. Fortunately, Java 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 boxes, sacks 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. You can think of this network of objects as a sort of tree-structure with one object at its root and other objects 'branching off' from it.



The diagram above shows a simplified view of the sort of tree-structured network of objects we may need to save. Here a game object contains a map (`ArrayList`) object and a player (`Actor`) object. The player has a sack which contains two treasures. The map contains three rooms each of which contain one treasure.

Of course, as the game progresses, this 'game state' will change. The player may take more treasures and put some of them into the sack while dropping other treasures into different rooms. When the game is saved all these complex relationships need to be saved. When it is reloaded, the saved state has to be restored. In a real game there would be far more treasures and rooms than I've shown here. At first sight, the problem of saving and restoring a game like that may look extremely difficult to solve.

In fact, it turns out that I can save or 'serialize' the entire tree of objects simply by serializing the object at the *root* of the tree – here that's the *game* object – and all the other objects in the rest of the tree (here, every other object in the game – the map, the rooms, the treasures and so on) will be serialized automatically!

The Serializable Interface

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:

Adventure4

```
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);
```

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*":

Adventure4 (AdventureForm.java)

```
private void saveGame() {
    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");
    }
}
```

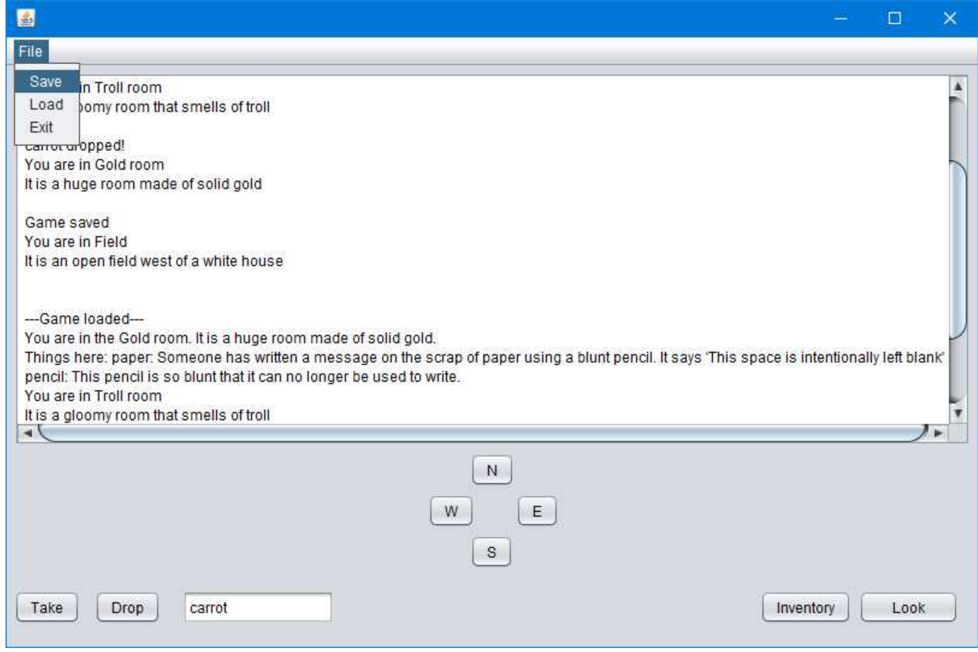
Loading a game is just as easy. I create a `FileInputStream` object to read from the same file to which the objects were previously written. I pass this to the `ObjectInputStream` constructor and I then read in data using the `readObject()` method whose return value is cast to the `Game` class.

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:

Adventure4 (AdventureForm.java)

```
private void loadGame() {
    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.



The *Adventure4* game project lets you take and drop objects by entering a name (such as "carrot") into a field and clicking the *Take* or *Drop* buttons. Move around the map by clicking the *N*, *S*, *W* and *E* buttons. To save and load a game use the items shown on the *File* menu.

Save the game at some point. Then move around the map some more and take and drop some other objects. Now restore your previously saved game. You will see that the game is restored to its state *as it was 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.

Where Next?

Now that we've arrived at the end of this book, you should have a solid understanding of the Java language and how to use it to create programs that can do calculations, work with text, create complex class hierarchies, save and load files and deal with errors. In order to continue developing your Java programming skills you now need to work on a programming project of your own.

You may want to try writing some sort of tools – say to navigate or organise files on disk. Or maybe you would be interested in doing mathematical calculations or writing a text editor, creating some sort of data-organising utility or programming a game. If you want something to get started with, you may want to take a look at the simple adventure game projects which I've provided (see the Appendix). These provide you with the basic functionality of a game and you can try to extend it to make a more complex game of your own.

Whatever you do next in Java programming, I wish you the very best of success. And I hope that this *Little Book Of Java* has helped you along the way!

Appendix

The Adventure Game Projects

You will find four little adventure game projects in the code archive. The *Adventure1* project begins by creating some fundamental game classes: `Actor` (the player), `Room` (the locations) and `Thing` (any type of object in the game). Both `Actor` and `Room` descend from `Thing`. The `Game` class itself defines the entire game.

In *Adventure2*, I add on a `Treasure` object. This also descends from `Thing`. It is used to create the objects found in rooms and which the player is able to take and drop. The player moves around a map, from one room to another, in response to direction buttons on the main form of the program. *Adventure3* adds on generics so that the map is now declared to be an `ArrayList` typed to hold `Room` objects only and so the 'cast' found in the previous project is no longer needed:

Adventure2 – moveTo()

```
moveActorTo(anActor, (Room) map.get(exit));
```

Adventure3 – moveTo()

```
moveActorTo(anActor, map.get(exit));
```

Adventure4 adds on the ability to take, drop and look at objects and to serialize data so that a game can be saved and restored.

The Little Java Book Of Adventure Game Programming

If you really want to write an adventure game, I have a whole book devoted to the subject! *The Little Java Book Of Adventure Game Programming* explains how to open, close, take, drop and look at objects. It shows you how to put objects into containers and how to write recursive methods to calculate the combined masses of `Treasure` objects (so the player can only take a specific number and weight of objects at a time). It provides code to parse adjectives so that the user can distinguish between similar object types. For example, if there are two Elvish swords, one big silver sword and one small golden sword, the user can enter the command: "Take the small golden Elvish sword". The

book also shows how to create 'traditional' types of text adventure which run at the command prompt and take commands in the form of short sentences.

```
You find yourself on the bridge of the HMS Starcrash,
an elite-class starship of Her Majesty's Royal Fleet.
There are no people on the bridge.
There are three boxes on the floor.
What do you want to do?
(Enter q to quit)
> take the box
Which of these do you want to take?
small cardboard box
big wooden box
tiny onyx box
> look at the tiny onyx box
It is a tiny box made of exquisitely-carved onyx (open)
There is something in it.
> look into the onyx box
The tiny onyx box contains:
gold ring (in the tiny onyx box)
> take the gold ring
You take the gold ring from the tiny onyx box
> close tiny box
You close the tiny onyx box
> w
Ready Room: This is a small but neat room. There is a cupboard on the wall.
> open the cupboard
```

This shows one of the games from *Little Java Book Of Adventure Game Programming*.

The *Little Java Book Of Adventure Game Programming* explains...

- How to write Interactive Fiction (IF) games
- Creating class hierarchies
- Compare different way of creating maps
- Commands such as: take, drop, look at X, put X into Y,
- Putting objects into nested levels of containers (sacks, chests etc.)
- Using lists and dictionary structures
- How to save and load multiple named games.
- Recursion
- Parsing user commands
- Matching adjectives with objects
- Distinguishing similar objects (e.g. "small gold ring" or "big silver ring")
- Calculating mass (so player can only carry a specified size or weight)
- Designing a command-line game to run at the command prompt
- Write your own games with the free BIFF game framework

IDEs/Editors

Here are a few (though there are others) IDEs and editors that support Java:

NetBeans
https://netbeans.apache.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 Books and Courses

For all the latest news on Bitwise Books and Courses, be sure to bookmark these sites…

For our full range of paperback books and eBooks plus free downloads, be sure to visit the **Bitwise Books** web site:
http://www.bitwisebooks.com

For video-based programming tutorials, go to the **Bitwise Courses** web site:
http://www.bitwisecourses.com

Visit the *Facebook* page of Bitwise Books & Courses:
https://www.facebook.com/BitwiseCourses

Subscribe to our *Twitter* feed:
https://twitter.com/bitwisecourses

Subscribe to our *YouTube* Channel:
https://www.youtube.com/BitwiseCourses?sub_confirmation=1

A Message from the Author

Thank you for reading my book. I hope you've found it useful – and, with luck, even enjoyable! I've been programming, and writing about programming, since the early 1980s. In the series of 'Little Books' I hope to be able to pass on some of the knowledge that I've gained over the years and provide readers with the sort of books I wish had been available (but weren't!) when I was learning to program.

Writing these books, and all the sample code that goes with them, takes a great deal of time and effort and, while I enjoy writing them – because, after all, programming is something I love – it is really hard for me for my books to compete against books produced by big publishing companies.

You can help me!

Reviews are incredibly important to me. They let potential readers know if my books have been useful to other people so that they can decide whether they might also be useful to them. I would therefore be really grateful if you could take a few moments to leave a short review on Amazon. You can do that by clicking one of the links on the next page, using your phone to read the QR code or (on other Amazon sites) by clicking the 'Write a Customer Review' button on the book page.

Thanks again for reading my book!

Huw Collingbourne

Leave a Review

Amazon (USA):

https://tinyurl.com/1gorjxki
or
https://www.amazon.com/review/create-review?asin=1913132145

Amazon (UK);

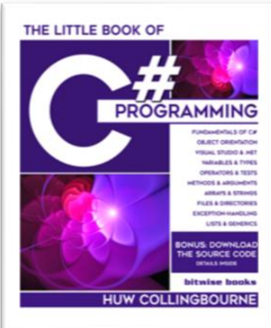
https://tinyurl.com/3x9qaz6k
or
https://www.amazon.co.uk/review/create-review?asin=1913132145

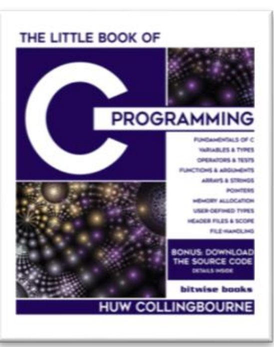
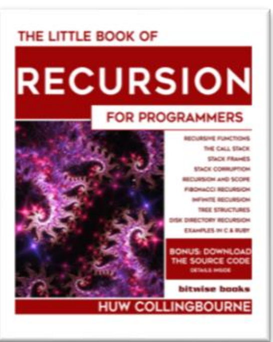
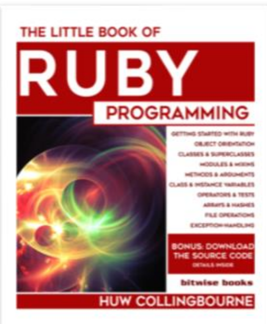
More Little Programming Books…

The Little Book Of Java Programming is one of a series of '*Little Books Of* …' for programmers. In each *Little Book* we aim to give you *just the stuff you really need* to get straight to the heart of the matter without all the fluff and padding.

We know that there is plenty of information online about standard code libraries, so we don't fill the pages of these books by duplicating that information. Instead, we aim to explain the really important details that you need to gain a solid understanding of each subject and start hands-on programming as quickly as possible.

Some other '*Little Books Of* …' are:

<table>
<tr>
<td>
</td>
<td>The Little Java Book of Adventure Game Programming
Advanced object orientated techniques:
• Create command line games
• Parse user input
• Open, close and lock containers
• Put treasures into containers
• Object properties and adjectives
• Traverse networks recursively</td>
</tr>
<tr>
<td>
</td>
<td>The Little Book of C#
A beginners guide to the C# language:
• Object Orientation
• Classes and Methods
• Methods & Arguments
• Arrays & Strings
• Exception-handling
• Lists & Generics</td>
</tr>
</table>

<table>
<tr>
<td>
</td>
<td>The Little Book of C
A beginners guide to the C language:
• Fundamentals of C
• Variable & Types
• Operators & Tests
• Functions & Arguments
• Arrays & Strings
• User-Defined Types</td>
</tr>
<tr>
<td>
</td>
<td>The Little Book of Recursion
Understanding recursion in C:
• Recursive Functions
• The Call Stack
• Stack Frames
• Stack Corruption
• Recursion & Scope
• Tree Structures
• Disk Directory Recursion</td>
</tr>
<tr>
<td>
</td>
<td>The Little Book of Ruby
Beginners guide to the Ruby Language:
• Getting Started with Ruby
• Object Orientation
• Classes & Superclasses
• Modules & Mixins
• Methods & Arguments
• Class & Instance Variables
• Operators & Tests
• Arrays & Hashes
• File Operations
• Exception-handling</td>
</tr>
</table>

You can also download a number of useful free resources from the Bitwise Books web site: http://www.bitwisebooks.com