

Object-Oriented Programming

PROGRAMMING WITH OBJECTS IN JAVA

DR. SEÁN RUSSELL

Contents

Contents	1
1 Introduction	1
1.1 Structure of a Computer	1
1.2 Life of a C Program	2
1.3 The Java Virtual Machine	3
1.4 Big Differences Between Java and C	4
1.5 Syntax of Hello World in Java	5
1.6 Components of a Java Program	5
1.7 Compiling and Executing Java Code	6
2 Java Programming Language Syntax	9
2.1 Primitive Data Types	9
2.2 Data Types for Text	12
2.3 Names	12
2.4 Syntax of Java	13
2.5 Syntax Errors	18
2.6 Practical Exercises	21
3 Objects, Classes and Methods	23
3.1 Object-Oriented Programming	23
3.2 Objects and Classes	24
3.3 Classes	24
3.4 Objects	28
3.5 Object References	31
3.6 Packages	33
3.7 Application Programmer Interface	34
3.8 API Use Exercises	36
3.9 Practical Exercises	37
4 Object-Oriented Programming	41
4.1 Why Object-Oriented Programming	41
4.2 The start of a basic arcade game	41
4.3 Cohesion	50
4.4 Game Completion Tasks	51
5 Encapsulation	53
5.1 Visibility	53
5.2 Encapsulation	55
5.3 Public Interface	57
5.4 Class/Static Variables	59
5.5 Constant Variables	60
5.6 Class Methods (static methods)	60
5.7 Enumerated Types	62

5.8	Practical Exercises	64
5.9	Game Completion Tasks	65
6	Interfaces	67
6.1	Code Reuse	67
6.2	Interfaces	69
6.3	Interfaces As Types	72
6.4	Polymorphism	74
6.5	Practical Example	76
6.6	Updated Interfaces	82
6.7	Practical Exercises	84
6.8	Game Completion Tasks	85
7	Inheritance	87
7.1	What is Inheritance?	87
7.2	Changing inherited functionality	92
7.3	Changing the Functionality Inherited from Object	95
7.4	Inheritance in Interfaces	97
7.5	Abstract Classes	97
7.6	Final Classes and Methods	100
7.7	Immutable Objects	100
7.8	Inheritance Example	102
7.9	Practical Exercises	104
7.10	Game Completion Tasks	106
8	Nested Classes	109
8.1	Inner Classes	109
8.2	Anonymous inner classes	111
8.3	Static Nested Classes	112
8.4	Game Example	112
9	Text Input and String Processing	115
9.1	Text Input	115
9.2	Input Checking	118
9.3	Dialogs	121
9.4	String Processing	122
9.5	Using a Scanner to Process A String	125
9.6	Practical Exercises	126
10	Graphical Input and Output	129
10.1	Drawing on the Screen	129
10.2	Using the mouse to interact	135
10.3	Interacting using the keyboard	139
10.4	GUI Game Example	141
10.5	Practical Exercises	146
10.6	Game Completion Tasks	149
11	Testing	155
11.1	Testing	155
11.2	Traditional Testing	155
11.3	Unit Testing	156
11.4	JUnit	157
11.5	Testing With IO	161
11.6	Code Coverage	161
11.7	How Much Code Coverage?	162

11.8 Practical Exercises	163
12 Exceptions	167
12.1 Errors	167
12.2 Stack and Stack Traces	169
12.3 Error Recovery	170
12.4 Exceptions	172
12.5 Recovering From Errors	173
12.6 Checked Exceptions	174
12.7 Exceptions and Unit Tests	177
12.8 Practical Exercises	177
13 Streams and File I/O	179
13.1 Files	179
13.2 Streams	181
13.3 Character Streams	184
13.4 Buffered Streams	186
13.5 Data Streams	188
13.6 Object Streams	191
13.7 Game Example	194
13.8 Practical Exercises	198
13.9 Game Completion Tasks	198
14 Advanced Topic - Concurrent Programming/Threads	201
14.1 Concurrent Programming	201
14.2 Creating Threads	202
14.3 Executing Threads	203
14.4 Controlling Execution	204
14.5 Thread Safety	209
14.6 Use In our Game	211
14.7 Practical Exercises	211
15 Advanced Topic - Generic Programming	213
15.1 Problems With Data Structures	213
15.2 Generics	214
15.3 Defining Generic Classes and Interfaces	215
15.4 Generic Methods	217
15.5 Reasoning About Generic Types	218
15.6 Why Use Generics	219
15.7 Generics and Primitive Types	221
15.8 Practical Exercises	222

Chapter 1

Introduction

This book is to accompany a student learning object-oriented programming, in particular we are using Java. The book assumes that you are already familiar with procedural programming such as python or C. Revising procedural programming would be a good idea if you have forgotten. You should already be able to solve problems of moderate difficulty using C.

During the course of this class, you will be required to write compile and test many different Java programs. There will be a particular focus on testing your own code, throughout this book. This is because it is very difficult to know if the code you have written is correct unless you have tested it. For most simple problems you should be able to read the code and know the output without executing the code. The more practice that you complete the better you will be at this. This is good practice, and generally the more programs that you write the easier you will find completing the course and getting a good grade.

It should be noted that this book and module are not all of the information that you should know about object-oriented programming in the Java language. The topics that we cover here are the most important to enable you to learn the basics of programming in Java. It is a good idea to continue practising your skill in other classes and projects, and generally learning more on your own time. The most successful students are those that spend time completing projects that they find interesting and that challenge them to learn more difficult and advanced techniques.

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the structure of computers, and the life cycle of C programs
- ☐ be able to compile and execute a basic Java program

1.1 Structure of a Computer

The process of creating your own programs in Java is slightly different from the same process in C. To understand the difference between the two we will talk the main components of a computer and how they work together. The primary components of a computer are the central processing unit (CPU), short term memory (RAM) and the long term memory (Hard Drives).

The CPU is the brain of the computer. CPUs perform most of the calculations that take place within a computer. CPUs work at a very basic low level executing code that would be difficult to understand for a human programmer. This code is called machine code. The example of machine code on the next page is represented in hexadecimal (base 16) and contains 9 instructions for the 8-bit CPU in a Commodore 64.

Example: Machine code represented in hexadecimal

```
1 78A9808D1503A92D8D14035860EE20D04C31EA
```

These instructions are almost impossible for a human to understand, but perfect for a computer. Humans, if they were working on programming at this level would work with an assembly language. The example below is the same instructions as the machine code represented in a way that is more understandable to humans. Each part of the instruction has meaning and is represented here.

Example: The same code represented in assembly language

```
1 SEI
2 LDA #$80
3 STA $0315
4 LDA #$2D
5 STA $0314
6 CLI
7 RTS
8 INC $D020
9 JMP $EA31
```

It is possible to program in an assembly language, but it is very difficult. For this reason, higher level programming languages were developed. These languages (including C, Java and Python) are converted from the text we write into a series of instructions represented in machine code. The CPU can **only** understand the machine code. This is what all of our programs in C were compiled into.

Memory is used to remember pieces of data for short periods of time. Memory contains all of the programs executing on a computer as well as parts of the operating system too. Every variable and constant that we use in our programs are stored here during execution. All of this information is stored in circuitry where each component represents a single bit of information. These components are grouped together (usually into 32 or 64 bit groups) and given an address. This means that when the computer wants a piece of information from memory it will ask for the piece at location x. Here x will be a number between 0 and the number of groups in the memory.

There are several types of long term storage, hard drives are the most common type. Typical examples are magnetic hard disk drive (HDD), which spins a magnetic disk containing many pieces of data, solid state drives (SSD) which store data within complex circuitry and various optical disc drives (ODD) which used light to store information on optical discs.

This is where we store all of the data on our computers. Particularly, this is where programs are stored when they are **not** being executed.

1.2 Life of a C Program

There are many other components in a computer system. These 3 are enough for us to have an understanding of the basics life of a computer program. A computer program starts off as a simple text file that we write according to the rules of a language. When we are finished editing the file we save it somewhere on a hard disk such as E:\program.c.

The file contains all of the instructions for what the program is supposed to do, but the CPU cannot understand the instructions as text. Before they can be executed, we must convert the instructions into machine code. This is done by the **compiler**. The **compiler** takes a text file as input and converts the text inside into a series of instructions that the CPU can understand. The compiled file is then stored on the hard disk with a name such as E:\program.exe.

When we execute this program, the machine code is loaded into memory and executed by the CPU one instruction at a time. Any variables or data stored by the program are also stored in memory and remembered as long as the program is being executed.

The basic steps for compiling and running our program look something like this:

Example: Compile and execute commands in C

```
1 E:\>gcc p.c      // compile the program to p.exe
2 E:\>p.exe        // run the program p.exe
```

Problems with C Programs

There are some small problems with working with C programs. The biggest problem is because computer systems can be very different from each other. Two computers using different types of CPUs cannot execute the same programs. This is because the different CPUs can use slightly different basic instructions. Therefore if I compile a program for my computer, the instructions are specifically generated for my CPU (an Intel CPU). The instructions available on an ARM CPU are not the same, so when I try to execute this program on that CPU it cannot execute some or all of the instructions. To get my program to work, I need to compile it again on the ARM CPU.

Additionally, two computers using different operating systems cannot execute the same programs. Whenever we perform many simple operations (such as print to the screen) we are actually using parts of the operating system called **libraries** to complete this task. If I compile a program for my operating system, (Mac OS X) it uses libraries from the operating system to execute. If I try to execute this program on a Windows computer, those libraries do not exist and my program cannot execute.

1.3 The Java Virtual Machine

Java is designed to remove these problems and make it much easier to develop code for any computer system. This is done by creating an interpreter called the **Java Virtual Machine JVM**. The JVM is basically a custom created set of instructions and libraries that a Java program can use when executing. For each type of CPU and operating system a different JVM exists. Each JVM is designed so that it can understand and execute the same instructions. The programs that we write are compiled into these instructions. Compiled Java is known as **bytecode**. No matter what system we execute our program on, the correct instructions are executed and the correct libraries are used.

Because Java is compiled into bytecode, we cannot just execute the program that is created. It contains instructions, but they cannot be understood by the CPU in our computer. Instead we have to use the JVM to execute the program we have compiled. This is shown in figure 1.1, where the Java code is compiled to bytecode and then the bytecode can be executed by the JVM on any operating system. The basic steps for compiling and executing a Java program are something like this:

Example: Compile and execute commands in Java

```
1 E:\>javac P.java  // compile the program
2 E:\>java P        // run the program
```

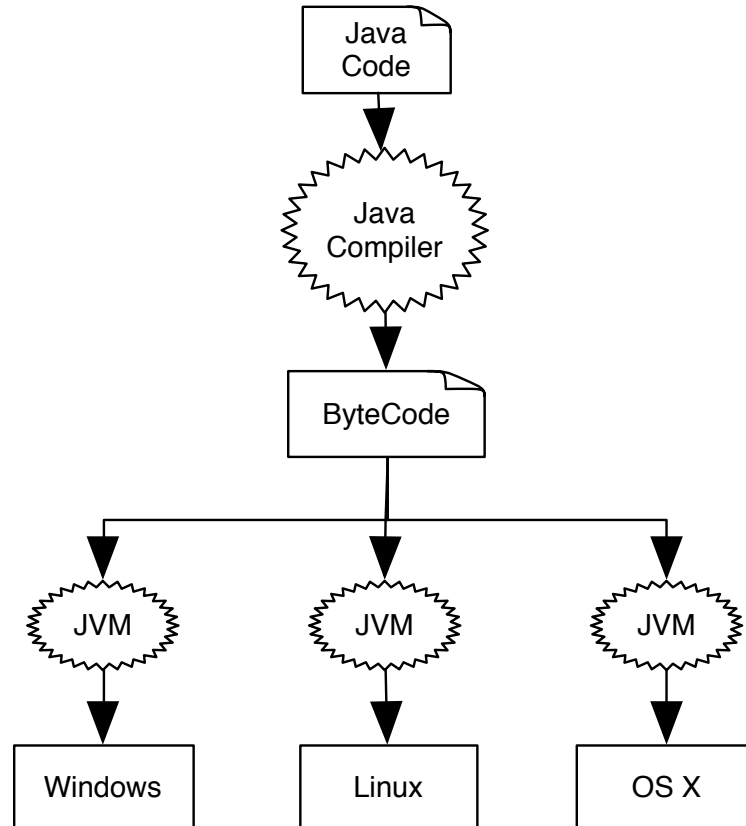


Figure 1.1: A representation of bytecode compiled for multiple operating systems

1.4 Big Differences Between Java and C

There are some more differences between Java and C that we will learn about throughout the course. These generally make Java an easier language to write complex programs in.

- Java does not use pointers
- Java manages memory for us
- Java is based on the idea of classes and objects

In C the idea of pointers can be quite confusing. What does it mean to reverence or dereference a variable? When should i use `*`? when should i use `&`? Java solves this problem by making objects (we will talk about these later) the same. In reality **all** objects are pointers called **references**, this way we never have to treat them differently. We never need to reference or dereference anything. This simplifies the process of working with references.

In C managing memory can be quite difficult. If you forget to free up some memory that you have allocated, then that memory can not be used again until the program is finished. In Java, memory is automatically allocated for us. When we are no longer using that piece of memory, Java will automatically free it for us. This functionality is called the **Garbage Collector**.

The biggest difference is that we have think and program in a different way. The purpose of this book (and class) are to learn about object-oriented programming. This requires us to think about our programs in a different way.

1.5 Syntax of Hello World in Java

In this section, we will look at the syntax of a very basic Java program. This is compared to the same program in C. As Java is a more complicated programming language, many components of the program will not be explained until we have covered more of the course.

The first program that is written by most programmers is a simple program called ‘Hello, world’. This usually just prints a small greeting to the screen of the computer. In C, this looks something like this:

Example: Hello World in C

```
1 #include<stdio.h>
2 int main(){
3     printf("hello world!");
4 }
```

The same program written in Java is a little more complicated and looks something like this:

Example: Hello World in Java

```
1 public class Hello {
2     public static void main(String[] args){
3         System.out.println("hello world!");
4     }
5 }
```

Now there are a lot of things that have to be declared in a Java program before it can be executed. But this program performs the same function as the C version. The first line declares a class called **Hello**. We will be doing this a lot and we are going to look at exactly what this means in Chapter 3. The second line declares a method named **main**. A method is a function that is inside a class. This method is usually called the **main method** and is the same as the main function in C. This definition will always be the same, for now we just need to remember it. The third line contains the only statement in the program. This just uses a method named **println** (just like **printf**) to print some text to the screen. The part **System.out** tells Java where it can find the method **println**. The last two line only contain the matching braces that belong with the first two lines.

1.6 Components of a Java Program

There are a number of basic components to a Java program. The most basic is the **class**.

- Every Java program must contain at least one class
- Many Java programs will be made up of many different classes.

Within one of the classes there must be a **main** method. The main method is used to start the execution of a Java program.

Naming of Classes

We will talk about classes a lot during this module. Basically each class should contain a set of related methods or functions. Classes are given names that describe the purpose of the class. In the example, we called the class **Hello**, because it was a class that contained the functionality to say hello. The name of a Java class **must** match the name of the file that it is stored in. So the file containing the class **Hello** must be stored in a file named **Hello.java**. This is done so that it is easy for the JVM to find code that is separated into different classes.

1.7 Compiling and Executing Java Code

Compiling and executing Java code on the command prompt/command line/terminal is similar to C.

1. First we must change to the correct directory using the `cd` command
 - `cd e:\java`
2. When we want to compile a Java file we use the command **javac** followed by the name of the file
 - `javac Hello.java`
3. When we want to execute a Java program we use the command **java** followed by the name of the **class**
 - `java Hello`

Integrated Development Environments


An Integrated Development Environment (IDE) is an application with a lot of tools that are useful for developing software. IDEs usually include:

- Text editor
- Compiler
- Syntax highlighting
- Built in program execution
- Code suggestions
- Debugging

There are a number of different IDEs, but we will be using the eclipse IDE - <http://www.eclipse.org>

Basics of using Eclipse

Eclipse is a very large and complicated program. This section will introduce some basics about how it works. To use eclipse you must first select a **workspace**. This is just a folder on your computer where eclipse will save all of your code and programs. Eclipse also stores some small files with data about the code that you have written. Select a location that is easy to get to because you will usually have to submit your work.

Eclipse groups programs together into projects. You cannot program using eclipse without a project. To create a new project click the new icon () located at the top left. Then select **Java project** from the drop down menu. In the **Project name** box type a name for your project and then click **Finish**. This will create a new folder in the workspace that will contain your Java files. You should choose an easy name such as **OOP-Week1** for your projects so that it is easy to tell what each project contains. Inside this folder there will be another folder named **src**, this is where you can find your Java files. You cannot have two projects with the same name, because you cannot have two folders with the same name.

Once you have a project created we can now start creating Java classes. To create a Java class click on the new icon again. Then select **class** from the drop down menu. In the window that appears we have to input some information. In this window we should enter the following information:

1. The location we want to put the file (this should be the folder named **src** in our project) in the **Source folder** box

2. The name of the class in the **Name** box
3. We can get eclipse to automatically add a main method if we select the tick box named `public static void main(String[] args)`
4. Finally click Finish and the Java file will be created!

This will open a new file that has been created in our project. Once we have added some code to this we will need to test it by executing the program. There are different ways to execute code in eclipse. Here are a few steps to complete the task.

1. In the main window of eclipse right-click on the file you want to execute
2. in the drop down menu go down to the **Run As** menu and select **Java Application**

The program will be executed and the output will appear in a tab named **console**

Chapter 2

Java Programming Language Syntax

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ be able to recognise the primitive data types in Java
- ☐ be able to use the String data type in Java
- ☐ understand the rules for identifier names in Java
- ☐ be able to explain what an expression is
- ☐ be able to determine the type of an expression in Java
- ☐ be able to use the basic syntax of Java
- ☐ recognise some of the most common errors shown in Java

2.1 Primitive Data Types

Primitive data types in C and Java are very similar. So learning to write basic programs in Java should be very easy. The main difficulty will come later when we learn how to program in an **object-oriented** way.

- Most primitive variables are the same in both languages
- The main difference between the two is the size of the data type
- We also have some new data types

Structs in C are replaced by classes and objects. But classes and objects are much more powerful. We will learn about these in chapter 3.

There are only two basic types of data, integers and floating point numbers. Everything else is based or built using these two types.

Integer types

Integer types are those that represent whole numbers such as 1, 2, 3 and 123412. It is important to remember that when performing mathematical operations with only integers the result will be an integer. For example $12 / 5$ would give us the result 2.

Comparison of integer types

Name in C	Size in C	Name in Java	Size in Java	Value Range
char	1 byte	char	2 bytes	0 to 65,535
byte	1 byte	byte	1 byte	-128 to 127
int	≥ 2 bytes	short	2 bytes	-32,768 to 32,767
long	≥ 4 bytes	int	4 bytes	-2^{13} to $2^{31} - 1$
long long	≥ 8 bytes	long	8 byte	-2^{63} to $2^{63} - 1$

Example: Declarations of integer types

```

1 char c = 'a';
2 byte b = 100;
3 short s = 1234;
4 int i = 35;
5 long l = 12345L;

```

Real/floating point types

Floating point types are not whole numbers such as 1.6, 2.3 and 1233.43425. The result of any mathematical operation will be a floating point number if there is at least one floating point number in the calculation.

Here we can see that the floating point types in Java are exactly the same as the floating point numbers used in C. These are defined in a standard known as IEEE 754. The float type is the equivalent of the single-precision 32-bit IEEE 754 floating point, and the double type is the equivalent of the double-precision 64-bit IEEE 754 floating point.

Generally we will use a double whenever we need to use a floating point number. Every real number we put in our program is automatically a double unless we add the character 'f' after the number. This is shown in the example below. The only reason to use a float is if you need to save memory in large arrays of floating point numbers.

Comparison of floating point types

Name in C	Size in C	Name in Java	Size in Java
float	== 4 byte	float	== 4 bytes
double	== 8 bytes	double	== 8 bytes

Example: Declarations of floating point types

```

1 float f = 2.5f;
2 double d = 235.78;

```

The boolean type

Java adds one type that is designed to make it easier to work out logical problems. This type is called boolean, after the mathematician and logician George Boole. George Boole was the first professor of mathematics at Queen's College, Cork (now University College Cork (UCC)) in Ireland. His work is often credited as setting the foundations for all computers.

The boolean data type is used for logic. It can only represent two values; **true** or **false**. This type is actually represented as an integer type within the JVM but cannot be used in any mathematical operations.

Boolean is the type of any result of a comparison such as `x == 0`. We can use all the same conditional operators as in C. e.g. `||` (OR) + `&&` (AND). As we will see in section 2.4, conditional statements such as `if` and `while` require the boolean type in their conditions.

Example: Declarations of boolean types

```
1 boolean t = true;
2 boolean a = x < 45;
3 boolean b = y == 8;
4 boolean c = a || b;
5 boolean d = c && b;
```

Standard operators

The standard operators for Java are the same as in C. As we can see in the examples below, they can be used to perform calculations with literal values (values we type into the code) or using variables that already exist within the code.

- **x + y** Add two values

```
int r = 1 + 4;
int r = a + b;
```
- **x - y** Subtract one value from another

```
int r = 6 - 2;
int r = a - b;
```
- **a * b** Multiply two values

```
double ans = 4.5 * 4;
double ans = a * b;
```
- **a / b** Divide one value by another

```
double ans = 4.5 / 4;
double ans = a / b;
```
- **a % b** Modulus operator, calculates the remainder of the integer division `a \ b`

```
int ans = 1234 % 4;
int ans = a % b;
```

For boolean values we have a different set of operators. These can be used to combine logical values in different ways. It is recommended that if you are using more than one operator in an expression that you use brackets to be sure the operations are happening in the correct order.

- **x || y** OR operator

```
boolean r = a || b; boolean r = x < 0 || x > 100;
```

`r` is true if either `a` **or** `b` are true
- **x && y** AND operator

```
boolean r = a && b;
boolean r = x > 0 && x < 100;
```

`r` is true if both `a` **and** `b` are true
- **!a** NOT operator

```
boolean r = !a;
```

`r` is true if `a` is false

```
boolean r = !(x > 0) ;
```

`r` is true if `x` is NOT greater than 0

2.2 Data Types for Text

In C if we wanted to store or manipulate a sequence of characters (i.e. text), we would use an array of chars. In Java we use a type called **String**. The string type is not like the primitive data types we have looked at so far, String is an object. Because we have not learned what exactly this means, we will ignore this for now. We will treat String just like a primitive data type. We will learn about Strings and other objects later.

Note

It is important that you remember that String starts with an upper case S. This is a common compiler error that can be easily avoided.

Example: Declarations of String variables.

```
1 String name = "Sean";  
2 String message = "hello world!";
```

Operators for text

There is only one basic operator of Strings. This is the **concatenation** operator, which combines two Strings together. This functionality is very useful for outputting complicated messages to the screen.

- **x + y** Adds two strings together to create a new String
String r = "hello " + "world";
String s = r + "!";

In this example, the variable r will contain "hello world" and s will contain "hello world!".

The operator can be used for more than just concatenating String, it can also concatenate a string with other types of data. This can be any of the primitive types that we have seen so far. For example if i concatenate the String "Cost: " with the number 45, the number is converted to a String "45" and then added to the end of the other String. This gives us the result "Cost : 45"

This can also be applied multiple times in the same expression. Java will perform the concatenations one at a time from left to right.

Example: Concatenation of Strings

```
1 String c = "Cost: " + 5;  
2 String message = "Hello, that will cost " + 5 + " RMB";
```

The result of the expression on line 1 is "Cost: 5", and the result of the expression on line 2 is "Hello, that will cost 5 RMB". The second example is a little more complicated than the first. The concatenation operation is applied two times, first the String and 5 are combined into "Hello, that will cost 5" and then this new String is concatenated with "RMB" to give us the final result.

2.3 Names

The name we give for anything in Java is called an **identifier**. This includes the names of **classes**, **variables**, **interfaces** and more. There are a lot of rules about what is allowed as an identifier in Java.

Here are the rules specifically identifiers.

- Identifiers can be made up of letters, (upper-case and lower-case), digits and the underscore (-)
- Identifiers cannot start with a digit
- Identifiers cannot contain symbols
- Identifiers cannot contain spaces
- You cannot use reserved words as an identifier (for, while, class...)
- Identifiers are **case sensitive**

If we do not follow any of these rules, our programs will not compile. This means that we must obey them always. There are also a number of conventions, these are similar to the rules but you do not have to obey them. It is good programming practice to follow these conventions when naming variables.

- Variable names should be descriptive
- Variable names should start with a lower-case letter
- If there is more than one word in your variable name, you should use what is called camel case
 - This is where the first letter of every word except the first is a upper-case letter

Example: Variable Declarations

```
1 int height = 178;  
2 int boxWidth = 40;  
3 String reallyLongVariableName = "Example of camel case"
```

2.4 Syntax of Java

Syntax is a word used in programming to describe the **grammar** of a programming language. Humans are much smarter than computers. When two people speak we need grammar to help us understand each other, but we do not need to have perfect grammar to be understood. A computer can only understand the instructions we give it if the grammar is **perfect**. Even if we make a mistake as small as forgetting a single character, the computer cannot understand, but a human would.

When we make a grammatical error while programming it is called a **syntax error**. These errors can be confusing and will prevent our program from being compiled. Later we will look at some examples of typical syntax errors that are often made by novice programmers.

Statements and expressions

The terms statement and expression are easily confused. However, it is important that you understand the difference between the two. A **statement** is a piece of code that does something. Usually a statement has a semi-colon (;) at the end, although this is not the case in more complicated statements such as if, while and for. Here are some examples of the basic statements that we will see and use:

Variable declaration: This is a statement that creates a new variable. The syntax is exactly the same as in C. **Syntax:** `type name;`. For example `int number;` or `String message;`

Variable assignment: This is a statement that gives a new value to a variable. Again the syntax is exactly the same as in C. **Syntax:** `name = expression;`. For example `number = 7;` or `message = "Hello"`

Calling a method: This is a statement where we use a method to perform some task. **Syntax:** `object.methodName(parameters);`. For example `System.out.println("Hello");`

An expression is a piece of code that can be calculated to give us a result. An expression will not always be followed by a semi-colon, but it is often part of a statement that is. Expressions may or may not contain operators. Here are some examples of the basic expressions that we will see and use:

Example: Different expressions in Java

```
1 1 + 3.0
2 number
3 number + 2
4 "Hello, my name is " + name
5 Math.min(a, b)
6 x < 100
```

Each of the above lines contains an expression. Expressions can be something as simple as a single number or variable name or something as complicated as calculations containing many variables, method calls, numbers and operators. The important part is that each component of the expression can be reduced to a value.

Lets examine the examples above. In the first expression we have two numbers and an operator, obviously both of the numbers can be reduced to values and additionally the operator `+` can be reduced to a value by applying it. As such we get the result 4.0. In the second example, we have only a single variable name. This can be reduced to a value by looking up the value stored in the variable. In the third example we perform the same step and then apply the addition operator. The fifth example can be confusing because it calls a method. This is an example of a method that performs some calculation and returns some result. Because a result is returned, it is allowed in an expression.

Because an expression gives us a result and because all components of the expression have a type, this means that the result of an expression will have a type. If we want to use the result of an expression or store it in a variable we need to know what type the result is. There is some simple logic that we can use to calculate what type the result of an expression is.

- If all of the components are the same data type, then that is the type of the expression. For example in the expression `7 + 7 + 67 + 9` all components are `int` so the type of the result is `int`
- If all of the components are of similar type, but not exactly the same data type, pick the **largest type**. For example if we have an expression that contains the integer types `byte`, `short` and `int`, then we choose the largest type which is `int`.
- If the components are different types, integers and floating point numbers and strings, then ask yourself can I save this as a `X` for each type. For example if we have an expression containing an `int` and a `double`, we would ask the following questions. Can we store this as an `int`? No if the answer is a real number then we would lose the information after the decimal point. Can we save this as a `double`? In this case, yes.

Lastly we have conditional expressions. These are any expression that can be calculated to `true` or `false`. The following are examples of conditional expressions; `true`, `x < 100` or `x < 100 && x > 0`. In Java the type of all conditional expressions is **boolean**

Questions

For each of the following expressions, what is the correct type?

1. `12 + 6.7` _____
2. `500 / 10` _____
3. `5 * 4.5f` _____
4. `"That will cost you " + 1 + 10 + " RMB"` _____
5. `123453L / 1234` _____
6. `100 < 56` _____
7. `false || (true && false)` _____

If statements

The if statement in Java is almost the same as in C. The only difference is that the condition must be an expression with the type **boolean**. This means we cannot use a number as the condition, for example `if(1){...}` will not compile.

Syntax: If statement in Java

```
1 if (booleanExpression) {  
2     code  
3 }
```

Example: An Example of an if statement in Java

```
1 if (x > 5) {  
2     System.out.println("hello");  
3 }
```

If-else statements

The syntax of the if-else statement is the same as in C.

Syntax: If-else statement in Java

```
1 if (booleanExpression) {  
2     code  
3 } else {  
4     other code  
5 }
```

Example: An Example of an if-else statement in Java

```
1 if (booleanExpression) {  
2     System.out.println("hello");  
3 } else {  
4     System.out.println("goodbye");  
5 }
```

While loop

The syntax of the while loop is the same as in C.

Syntax: While loop in Java

```
1 while (booleanCondition) {  
2     repeat some code  
3 }
```

Example: An Example of an while loop in Java

```
1 while (x > 5) {  
2     System.out.println("hi");  
3     x++;  
4 }
```

For loop

This is where there is a small difference between the for loop in C and in Java. In Java it is possible to declare a temporary variable within the initialisation section of the for loop. This means when creating a for loop we do not need to create an additional variable first. This variable is then only in scope within the for loop.

Syntax: For loop in Java

```
1 for (initialise variable; booleanExpression; counter) {  
2     repeat some code  
3 }
```

Example: An Example of an for loop in Java

```
1 for (int i = 0; i < 5; i++) {  
2     System.out.println("Hi");  
3 }
```

In this example the variable `i` is declared in the for loop and does not need to exist before.

Switch Statement

The switch statement is similar to a series of if-else statements. We pass a single variable to the statement and it will choose the correct case to begin executing. The main difference is that all code after the selected case will be executed unless a **break** statement is added between each case.

Example: Syntax of switch statement in C

```

1 switch(expression) {
2     case constant-expression :
3         statement(s);
4         break;
5
6     case constant-expression :
7         statement(s);
8         break;
9
10    default :
11        statement(s);
12 }
```

Switch statements are very similar in Java, however they are generally more useful because instead of just using an integer they can also use Strings and enumerated types (We will learn about these later). The syntax is exactly the same in Java as it is in C. The only difference is what type of expression can be used passed to the statement.

Arrays

Arrays in Java can be a little different than they are in C. This difference is mostly in how an array is declared and extra information that is stored in an array. The first big difference is that arrays are objects, we will learn a lot more about objects in chapter 3.

First let's examine the syntax of declaring an array in Java and C. In C, the array declaration statement `int arrayName[arraySize];` declares a variable called `arrayName` that contains an array of `arraySize` ints. Here the variable is declared and the memory is allocated in a single statement. In Java, the statement `int[] arrayName;` only declares a variable called `arrayName` that can contain an array of ints.

Array bracket positioning

The big difference here is the positioning of the brackets in the declaration. In Java they should be placed beside the type so it is clear before reading the variable name that this is an array.

Actually creating an array in Java is more complicated, both declaration and construction statements are required. Construction of an array uses the syntax `new type[arraySize]`, this allocates the memory for an array of `arraySize` of the defined type. To then combine this with the declaration we end up with the following syntax: `type[] arrayName = new type[size];`. For example, `int[] numbers = new int[10];` declares and constructs an array of 10 ints called `numbers`.

Using the array is unchanged from C, if we want to store the value 123 in element 0 of an array called `numbers`, then the code is `numbers[0] = 123;`. Also if we want to access the value stored in element 7 of the array called `numbers` we use the expression `numbers[7]`. This can then be used in any calculation or simply printed to the screen.

Array list initialisation

Just as in C, it is possible to declare and construct an array as well as insert values all in a single line of code. The type of the array is declared as normal, however we do not declare the size of

the array. The size of the array is calculated automatically based on the number of items we are inserting.

Syntax: Array list initialisation

```
1 type name = {comma, separated, values};
```

Array size

The best added piece of functionality is that the array knows its own size. We do not need to remember the size of an array separately. The size of an array is stored in a special instance variable called `length`. To find out the size of an array we use the syntax `arrayName.length`. For example, `numbers.length` would give us the value 10.

This value is most useful when we are using arrays and loops together. A loop to visit every element in an array can be written based on a template. The following code can be used to loop through any array no matter what size, simply by replacing `arrayName` with the name of your array.

```
for(int i = 0; i < arrayName.length; i++)
```

Obviously you will have to add the actual code of the loop yourself, depending on what you are trying to achieve.

2.5 Syntax Errors

We have learned that when using programming languages we must use perfect syntax, or our program will not compile. When programming we will make many mistakes, the most common of these will be using the syntax of the language incorrectly. These are called **syntax errors**. Syntax errors are shown whenever we try to compile our programs. It is important to remember any syntax errors that we see and more importantly remember what caused them. This way the next time we see the error we will already know what the problem and solution are.

Sometimes the error messages we see are not very easy to understand. It is common that a compiler will show many errors all caused by a single mistake. The best practice when programming is to solve one problem at a time and compile again. You might notice that by solving a single error you have also solved many other errors. It is generally easier if you compile your code after every line you write. This means you will see the error as soon as you have made the mistake. In IDEs like eclipse this will happen automatically, you may only need to save the file and it will be compiled and the errors will be highlighted.

Typical syntax errors

Here is a list of some of the most common syntax errors, we will have a look at the results of these.

- File name and class name **do not match**
- Misspelled a keyword
- Misspelled a variable name
- Incorrect variable name
- Forgotten semicolon
- Forgot to import a library

Recreate each of the following errors and compile the file on the command line. The output of the compiler will be in Chinese. Record the first error in the space after each.

File name and class name do not match**Example: FName.java**

```
1 public class Hello {  
2     public static void main(String[] args){  
3         System.out.println("Hello, Sean");  
4     }  
5 }
```

Example: Output of the compiler

```
1 FName.java:3: error: class Hello is public, should be declared in a  
    file named Hello.java  
2 public class Hello {  
3     ^  
4 1 error
```

Misspelled a keyword**Example: MSpell.java**

```
1 public Class MSpell{  
2     public static void main(String[] args){  
3         System.out.println("Hello, Sean");  
4     }  
5 }
```

Example: Output of the compiler

```
1 MSpell.java:3: error: class, interface, or enum expected  
2 public Class MSpell{  
3     ^  
4 MSpell.java:4: error: class, interface, or enum expected  
5 public static void main(String[] args){  
6     ^  
7 MSpell.java:6: error: class, interface, or enum expected  
8 }  
9 ^  
10 3 errors
```

Misspelled a variable name**Example: MSpell2.java**

```
1 public class MSpell2{
2     public static void main(String [] args){
3         String varName = "Sean";
4         System.out.println("Hello , " + varname);
5     }
6 }
```

Example: Output of the compiler

```
1 MSpell2.java:6: error: cannot find symbol
2 System.out.println("Hello , " + varname);
3                               ^
4   symbol: variable varname
5   location: class MSpell2
6 1 error
```

Incorrect variable name**Example: MSpell3.java**

```
1 public class MSpell3{
2     public static void main(String [] args){
3         int 2cool = 34;
4         System.out.println("Hello , " + 2cool);
5     }
6 }
```

Example: Output of the compiler

```
1 MSpell3.java:5: error: not a statement
2     int 2cool = 34;
3     ^
4 MSpell3.java:5: error: ';' expected
5     int 2cool = 34;
6     ^
7 MSpell3.java:6: error: ')' expected
8     System.out.println("Hello , " + 2cool);
9     ^
10 MSpell3.java:6: error: illegal start of expression
11     System.out.println("Hello , " + 2cool);
12     ^
13 4 errors
```

Forgotten semicolon**Example: ForgetS.java**

```

1 public class ForgetS{
2     public static void main(String [] args){
3         String name = "Sean"
4         System.out.println("Hello , " + name);
5     }
6 }

```

Example: Output of the compiler

```

1 ForgetS.java:5: error: ';' expected
2     String name = "Sean"
3                     ^
4     1 error

```

Forgot to import a library**Example: ForgetI.java**

```

1 public class ForgetI{
2     public static void main(String [] args){
3         Scanner in = new Scanner(System.in);
4         String name = "Sean";
5         System.out.println("Hello , " + name);
6     }
7 }

```

Example: Output of the compiler

```

1 ForgetI.java:5: error: cannot find symbol
2     Scanner in = new Scanner(System.in);
3                     ^
4     symbol:   class Scanner
5     location: class ForgetI
6 ForgetI.java:5: error: cannot find symbol
7     Scanner in = new Scanner(System.in);
8                     ^
9     symbol:   class Scanner
10    location: class ForgetI 2 errors

```

2.6 Practical Exercises

1. For each of the errors listed above, create a Java class containing the error and compile it to see the error for your self.
2. Write a Java program that prints the message "Hi my name is XXXXX", where XXXXX is replaced by your name.

3. Write a Java program that prints JAVA to the screen using * to create large letters e.g.

```
*****  *****      *      *      *
*        *          * *    **    *
*****  *****  *****  * * *
      * *      *      *      * **
*****  *****  *      *      *      *
```

4. Write a Java program that prints "hello" to the screen 5 times, on the same line and separated by a space.
5. Write a Java program that prints "hello" to the screen 5 times, each on a different line (don't use \n).
6. Write a Java program that prints all of the numbers between 0 and 100 inclusively on a single line separated by spaces.
7. Write a Java program that prints only the even numbers between 0 and 100 inclusively on a single line separated by spaces.
8. Write a Java program that prints only the numbers between 1 and 500 that are divisible by both 3 and 5. All numbers should be printed on a single line and separated by spaces.
9. Write a Java program that prints all of the prime numbers less than 1000. Each number should be printed on a single line and separated by spaces.
10. Write a Java program that prints all of the prime numbers less than 2000. There should be 10 numbers printed on each line and they should be separated by tabs.

Chapter 3

Objects, Classes and Methods

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the syntax of a Java class
- ☐ be able to define a class
- ☐ be able to define methods for a class
- ☐ be able to define constructors for a class
- ☐ be able to call constructors to create an object
- ☐ be able to call methods on an object
- ☐ understand the concept of the implicit parameter
- ☐ be familiar with how object references are stored and copied in Java
- ☐ understand the structure of Java code in packages
- ☐ be able to source relevant information in the Java API

3.1 Object-Oriented Programming

When writing large programs, it is easier for programmers to complete the program if it is broken down into a number of smaller parts. In C we used functions to break down our programs, so instead of writing all our code in a single main method we could write smaller pieces as a function and use them. This idea is developed much further in object-oriented programming. Here is a quote from Steve Jobs explaining the idea of object-oriented programming.

Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here.

Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."

You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you

don't have dollars in your pocket. Yet I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high level.

3.2 Objects and Classes

In object-oriented programming, larger programs are broken down into classes. Each class usually contains a set of **related** functionalities. In the example above, there might be a cleaner class that has the functionality to clean clothes, a taxi class that can take people from place to place and an assistant class that will use the taxi and cleaner objects to get your clothes cleaned.

Each class can be broken down into a number of methods. There may be a single method for each piece of functionality or there may be many. In our example we might have high level methods called `cleanClothes` in the assistant class or `driveToLocation` in the taxi class. The methods might also be very low level such as `heatWater`, `addSoap` and `dryClothes` in the cleaner class or `increaseSpeed`, `turnLeft`, `decreaseSpeed` in the taxi class.

The general idea is that if we are able to break our problem down into a number of classes, then each class is broken down into methods. Having broken down our problem into such small pieces, each method will be smaller and easier to write. Then our job is to put all of the smaller pieces together in the correct sequence.

Classes

Classes are similar to structs in C. A class is a collection of variables, know as **instance variables** (also called fields, data members, members), just like a struct in C. A class can also contain its own methods, which can perform calculations with the variables contained in the class. A class tells us what information an object can remember, an object remembers the actual values.

Objects

An object is an **instance** of a class. An instance is where we give values to the instance variables within a class. We can create many objects based on the same class, just like we can have many variables based on the same struct. Each object can have different values for its instance variables.

For example we can define a class called A that can remember two int values b and c. Each object based on class A will have values for b and c.

Difference between objects and classes

The distinction between classes and objects can be difficult to understand. The main thing to remember is that the idea of object-oriented programming is based on how humans see the world. In the real world we can think of a class like an idea. Lets examine the idea of a 'chair', it is something that we can sit on, it has legs and a back. There are billions of real chairs in the world, each of these could be thought of as an instance of the idea 'chair'. In this way objects are the real versions of the ideas that are classes. The reality is a little more specific, but this basic idea will help you understand how objects and classes are related.

3.3 Classes

Every object is based on a class. For example the object "Hello, World!" is based on the String class. This is a very important point because it is the class which determines what functionality an object has. Every object based on the String class has the same methods and instance variables. The only difference between two objects based on the String class is the values stored in their instance variables. Every single object based on the same class can have different values.

Designing classes

Last year you learned how to use a struct to group variables together to make something useful. Here is an example of a struct designed to store a date as three integer values; **day**, **month** and **year**.

Example: Struct representing a date in C

```
1 struct Date{
2     int day;
3     int month;
4     int year;
5 };
```

This struct allows us to store three values using a single variable. This makes it much easier to work with values that are more complex.

In Java we can do perform the same function using a class. Here the **Date** class contains three **instance variables**; **day**, **month** and **year**.

Example: Date class represented in Java

```
1 class Date{
2     int day;
3     int month;
4     int year;
5 }
```

Adding functionality to classes

The big difference between using structs in C and classes in Java is that we can write methods inside a class. This is not possible with functions and structs, we could write them in the same file but they would not be connected. Lets look at an example of using functions with a struct in C. Lets say I want to write a function to change the date to the next day, we will call this function **incrementDay**. In order for the function to know which variable it is supposed to increment the day for, the variable has to be passed as a parameter of the function. So we end up with something like this **void incrementDay(struct Date* d){}**. The code in the function can then perform the calculations and change the values of the variables inside the struct d.

In Java, we keep the functionality and data in the same place. Because the methods and instance variables are both inside the class, we do not need to pass a Date object as a parameter.

Example: Date class with unfinished methods added

```
1 class Date{
2     int day;
3     int month;
4     int year;
5     void incrementYear() {}
6     void incrementMonth() {}
7     void incrementDay() {}
8 }
```

Lets take these examples one step further and actually implement one of the methods/functions.

Example: Date struct and increment month function in C

```

1 struct Date {
2     int day;
3     int month;
4     int year;
5 };
6 void incrementMonth(struct Date* d){
7     d->day = 1;
8     if (d->month < 12) {
9         d->month++;
10    } else {
11        incrementYear(d);
12    }
13 }

```

Here we can see that in order to implement the functionality of incrementing the month, a struct has to be passed as a parameter. Actually a pointer to a struct has to be passed so that we can change the original value, however because we no longer have to think about pointers lets ignore this. The first thing that happens when the month is changed is that the day counter returns to 1. This is changed in `d` using `d->day = 1;`. We then check if the value for month is less than 12 (before December) and either increment the value of month in `d` or pass `d` to another function to increment the year.

The important point to note is how we have to access the variables inside the date struct in order to see or change them. Now lets have a look at the same example in Java.

Example: Date class with implemented increment month method added

```

1 class Date {
2     int day;
3     int month;
4     int year;
5     ...
6     void incrementMonth() {
7         day = 1;
8         if (month < 12) {
9             month++;
10        } else {
11            incrementYear();
12        }
13    }
14 }

```

Here we can see that there are some significant differences between the two examples. Most importantly, there is no parameter for this method. How then does the method know what the values of day, month and year are? Because the method and the instance variables are part of the same class, the method will always change the values for the object that it is inside. This way we can simply used `day` to access or change the value of day and so on.

Lets look in more detail at the process of adding methods to classes. In Java, methods can only be defined inside of classes. So every method will belong to a class. Defining a new method is very similar to defining a function in C. We need to complete the following steps;

- specify the return type
- specify the name

- specify the parameters required
- add the code

Lets look at the steps involved in adding a method to increment the year in our Date class. What is the return type? We have two choices here, this method is doing some work for us by changing the date object so it does not need to return an result, however we could have it return the value of year after it has been changed. So we have a choice between `void` and `int`. What is the name of the method? Just like with naming variables method names should be descriptive, so we will name the method `incrementYear`. What parameters does the method require? None, all of the variables it needs are inside the object already. Parameters are specified in the exact same way in Java as they are in C. So we end up with a method signature that looks like `void incrementYear()`.

Now we need to add the code. The easiest way to do is to break this down into what happens to each of the values when the year is changed. The first part is the easiest, they value of year increases by 1. When the year changes we go back to the start, so the value of month and day will go back to 1.

Example: `incrementYear` method for the Date class

```
1 void incrementYear() {  
2     day = 1;  
3     month = 1;  
4     year++;  
5 }
```

Constructors

We have seen the basics or how to design a class that can store data, and we have seen how we can add some functionality to change that data. But we haven't seen how to actually create and use objects based on this class. Whenever we want to create an object we need to use a **constructor**. There is automatically a constructor for every class that we write, but these are not very good and we usually replace it with more suitable constructors.

We can construct an object using the keyword `new`. If I want to create a object based on the Date class, the code to use the constructor is `new Date()`;. Here `new` tells us we are creating a new object, `Date` tells us what object the class is based on and inside the brackets we pass any parameters that are required.

Usually when we create an object, it is then stored in a variable. This is done in exactly the same way we would store an `int` in a variable. When we want to store an `int` variable called `n`, we would use the code `int n`;. But what about objects? Because every object is based on a class, we use the name of the class as the type of the variable. So if we want to store an object based on the Date class called `d`, we would use the code `Date d`;. If we combine this with code for constructing an object we end up with `Date d = new Date()`;

When we have created an object, what values will the instance variables have? If we do not give them a value, then they will all be the default value (This is 0 for numbers). This is not very useful so Java allows us to add new constructors to our classes. In each constructor, we can specify the parameters we require to create the object. In our date example, it would make sense that we require a day, month and year to create a Date object.

Adding a new constructor is like adding a new method, the main differences are that we do not need to specify a return type and the name must match the name of the class. This means that the only things we really need to do is to specify the parameters and add the code. In this example the parameters we require are three integers, named `day`, `month` and `year`. So between the brackets we add `int d`, `int m`, `int y`.

Example: Unfinished constructor for the Date class

```
1 Date(int d, int m, int y){  
2  
3 }
```

Parameter names and scope

You notice here that the three parameters are named d, m and y. This is to make it easier to tell the difference between them and the instance variables that belong to the class. If the parameters were named day, month and year, extra code would be needed to tell the difference between the parameters and the instance variables. It is always a good idea to use different names so your code is easier to understand.

Now that we have added a constructor to the class, we need to pass matching parameters to the constructor when we are creating a Date object. For example, if I wanted to create a date object to represent my birthday I would use the code `Date birthday = new Date(21, 7, 1985);` Now the values passed will go to the parameters d, m and y respectively.

But what do we do with these values when the object is created? This requires code to define what should be done with each of the parameters. We want to store the first value in the instance variable day. This value is represented by the parameter d, so we use assignment to store the value of d in the instance variable day. The code required for this is `day = d;`. A common mistake is to reverse the order of the parameter and the instance variable, in this case the value passed to the constructor will not be stored in the object. For the other two variables we perform the same steps, `month = m;` and `year = y;`.

Example: Finished constructor for the Date class

```
1 public Date(int d, int m, int y){  
2     date = d;  
3     month = m;  
4     year = y;  
5 }
```

3.4 Objects

Having looked at how classes are defined, constructors and functionality added, let us have a look at how objects are used. An object is a complex value that we can change by accessing its instance variables and calling its methods. But how do we know what instance variables and methods an object has? This is defined in the class that the object belongs to.

Changing instance variables

Every class specifies the instance variables that you can access and the methods that you can call for an object of that type. This means that every object based on the Date class will have the same instance variables, and methods. So for each one we can set the value of day, month or year and call methods such as `incrementYear`.

As we have already seen how to create objects using constructors, let's now look at how to change the instance variables of an object. Once we have an object constructed, we can start using it in our program. Let's say that I want to change the values of day, month and year in a date object, first I need to be able to access the variables. The syntax for this is `objectName.variableName`. Remember that every object based on the date class contains instance variables with the same name, this

means we must tell Java which object we are using. To do this we put the name of the variable we are using in the `objectName` portion.

For example, if I have a date object named `today`, I could set the date to 7 using the following code; `today.day = 30;`. To set month and year would be the same `today.month = 5;` and `today.year = 2018;`. After this code has executed, the values of the instance variables in `today` will have changed to the values we have set.

It is generally considered bad programming practice to change an object using its instance variables. Instead we would normally use a method to change the values. Because we are writing the methods, this means that we have control over how the variable is changed. Think of a simple example, `today.day = 42;`, here because we have directly changed the value of `day` we were able to give it a value that does not make any sense (There are no months with 42 days in them). Instead it should be changed using a method that we have implemented in the class, this method can then ensure that only correct dates can be entered.

Calling methods

We have studied how to define methods in classes, but we have not used any of these methods. When we use a method, we say that we **call** the method. Methods cannot be called unless we have an object to call it. The steps required to call a method are not very complicated (with practice);

- Get a reference to the object you wish to use
- We need to know the name of the method we want to use
- We need to know the parameters that are required by the method

Once we have this information we use the following syntax

Syntax of calling a method

`objectName.methodName(parameters)`

Note a method call will always be followed by brackets, even when there are no parameters

Lets look at some examples assume that we have a date object named `today` and we want to change the date by putting it forward a single day. We already know the name of the object we want to use (`today`), based on the methods that the class has we want to call the `incrementDay` method and this method requires no parameters. This means we end up with the code `today.incrementDay();`.

Lets study another example that we have already use every time we printed something. `System.out.println("Hello, World");`. Here the name of the object is `System.out`, this is a special object that is automatically created for us in every program. The name of the method is `println` and we have a single parameter, the string `"Hello, World"`.

We have seen how to change the instance variables of an object directly and indirectly using methods. It is important to remember that only the object we have named was changed. As there can be many objects based on the same class, the instance variables of all other objects base on this class will not be changed. Compile and execute the following example (Once you have completed the Date class).

Example: Example of multiple date objects

```
1 Date today = new Date(7, 4, 2016);
2 Date tomorrow = new Date(8, 4, 2016);
3 for(int i = 0; i < 100; i++){
4     today.incrementDay();
5 }
6 System.out.println(today.day + "/" + today.month + "/" + today.year);
7 System.out.println(tomorrow.day + "/" + tomorrow.month + "/" + tomorrow
    .year);
```

The output shows that because the increment day method was only called for the object today, the instance variables of the object tomorrow are the same as when it was created

Calling Methods from inside the class

When calling methods, we usually need to use the name of the object. This is only the case outside the class. When writing code inside the class we do not need to include the name of the object, this is because it was already used when the method was called. For example, in the method `incrementMonth` the method `incrementYear` is called without any object name put before it. This is because the object will stay the same and was supplied when `incrementMonth` was called, or when the method that called `incrementMonth` was called.

This can be simplified if we make use of a keyword in Java that represents the current object. **this** is a keyword for representing the **implicit parameter**. The implicit parameter really just means this object, the one that was used to call the method. For example, assuming we replace `incrementMonth()` with `this.incrementMonth()`, there will be no change in how the code executes (because Java already knew this) but the process is a little clearer to us. But what does **this** mean? **this** is the object that was used to call the method. if we have the code `today.incrementMonth()`; , then **this** in `incrementMonth` just means the object `today`. If we have the code `birthday.incrementMonth()`; , then **this** in `incrementMonth` just means the object `birthday`.

Whenever we use an instance variable or method inside our class, Java is assuming that **this**. is put before it. We can add this if it makes it easier for us to understand, but it is not necessary for Java to understand. The following example shows the `incrementMonth` method implemented using **this**.

Example: Using implicit parameter in incrementMonth

```

1 class Date {
2     int day;
3     int month;
4     int year;
5     ...
6     void incrementMonth() {
7         this.day = 1;
8         if (this.month < 12) {
9             this.month++;
10        } else {
11            this.incrementYear();
12        }
13    }
14 }
```

More complex methods

It is often necessary to compare two objects of the same class. In C we would write a function that takes two parameters and then performs the calculation. In Java, we do not need to pass a parameter to access our own instance variables, but we do need a parameter to access the instance variables of another object. Lets look at an example of checking to see if two date objects contain the same date.

Firstly, we need to know what the return type of this method is. As we are asking a question with a true/false answer the return type will be boolean (either they are the same date or they are not). Secondly we need to know the name of the method, `sameDate` describes the functionality pretty well. Lastly we need to know what parameters the method should required. We wish to compare two date objects, but we already have access to the instance variables of one, this means we only need a single date object as a parameter. So our method will start as `boolean sameDate(Date d){}`.

Next we need to write the code for the method, this comes down to asking how we know when two times are the same? When they have the same value for day, month and year. But how do we tell the difference between the instance variables of the two objects? Because both objects have the same instance variables we need to use names to tell the difference. In the method above we have a parameter named `d`, we can use this to identify the instance variables of the parameter e.g. `d.day`. For the instance variables of the other object we do not need to use any name (we already have access to them), we can just use the `day`.

This can sometimes be confusing, when we only have one parameter, how can we compare two objects? This is possible because the other object is the **implicit parameter**. This basically means that when we are calling the methods of an object, we have access to the instance variables it contains, so it is like it has been passed as a parameter.

Example: Method comparing two date objects

```
1 boolean sameDate(Date d){  
2     if (this.day == d.day && this.month == d.month && this.year == d.year)  
3     {  
4         return true;  
5     }  
6     return false;  
7 }
```

Here we can see that when comparing the two objects, we are comparing the `this.day` against `d.day`. This shows the implicit parameter (represented by `this`) compared to the actual parameter (represented by `d`).

Let's look at an example of calling the `sameDate` method. Because we are comparing two dates, we must first create two date objects. To call the method, first we need to choose one of the objects to use. This is normally easy because when calling a method we would normally want to change a single object, but here we have the choice of two. In reality it does not matter in this case if we use `a` or `b`, the result will be the same. If we call the method on the object `a`, the parameter must be the object `b`. This allows us to compare both objects and the result is stored in the variable `r`.

3.5 Object References

Previously, we learned that Java does not use pointers. Instead Java uses references for the same purpose. The major difference is that **every** object variable is a reference. This means that we never have to reference or dereference anything. All objects are treated the same way. However, it is important that we understand the difference between how primitive values are treated and how references are treated in Java.

When an item is constructed in Java, it is stored somewhere in memory. When we store this object in a variable, we are only storing the reference to the memory location. As such it is often easier to think of object variables as addresses.

Copying a primitive variable

An important point to note about this is that we cannot copy an object in the same way as we can copy a primitive variable. Let's have a look at what happens when we copy a primitive variable. Consider the following code:

Example: Copying a primitive variable

```
1 int x = 120;
2 int y = x;
3 y++;
```

When we execute the first line, we create a location in memory with the name `x` and in that location we store the value 120. When we execute the second line, we create a location in memory named `y` and we copy whatever value is stored in the location with the name `x`. This means that now `y` contains the value 120. Finally, when we execute the third line the value in location `y` is increased by 1. As the end, the location `x` contains the value 120 and the location `y` contains the value 121. This is what we would expect to happen when the code executes.

Copying an object variable

Consider the following code:

Example: Copying an object variable

```
1 Date t = new Date(08, 04, 2016);
2 Date b = t;
3 b.incrementDay();
```

When the first line of code is executed, two things happen. First, a new date object is created and stored somewhere in memory (lets say address 1234). Secondly, the address of the object we have just created is stored in a location with the name `t`. So the value of the variable `t` is really the address of our date object.

When the second line of code is execute, we create location in memory with the name `b` and we copy the value of the variable `t`. The value of `t` as we already know is actually the address of the object we created. This means that both `t` and `b` are remembering the same address.

Lastly when we execute the final line of code, Java looks up the address stored by the variable `b` (address 1234). It calls the method `incrementDay` on the object in address 1234, this causes the day to change. If the objects had been copied, then the variable `t` would lead to an object with the date 8/4/2016 and the variable `b` would lead to an object with the date 9/4/2016. However, because there is only one object and both variables contain the same address, the date for both is 9/4/2016.

Copying an object

We now know that we cannot copy object variables in the same way as primitive variables. But how can we copy an object? We need to construct a new object. We can then copy the value of each instance variable to the new object (or pass them as parameters in the constructor).

Example: How to copy an object in Java

```
1 Date t = new Date(8,4,2016);
2
3 Date b = new Date(t.day, t.month, t.year);
```

Null references

References work in the same way as pointers, they simply like the variable to the location in memory. However, a reference will not have a value until one is assigned to it. This is usually done during the construction process when we are creating our own objects. If we create an object variable without assigning it a value, what does it point to and what will happen if it is used?

Java uses a special value called `null` when a reference has not been linked to an object in memory. For example, simply declaring an object variable, e.g. `String name;`, creates a reference to this null value. The reference does not actually point to any information. If the variable is used before a value is assigned the program will fail. This type of error is called a `NullPointerException`, and is one of the most common errors experienced by programmers in object-oriented programming.

3.6 Packages

When working on very large programs, we can end up with a lot of classes (and other stuff too). Because each class is stored in its own file, this means that we can end up with a large number of files. This can be difficult to organise, so usually related code is grouped together to keep the structure neater. For example, we might group together all of the code that is responsible for creating a user interface separately from all of the code that manages the database and all of the application code.

In Java packages are used to group code together. Each package is a folder where we put related code together. We can even break code in a package into smaller packages. This helps to keep a large number of files sorted in the right places, so we can find and modify them easily.

When we group files into packages we have to put the file into the correct folder. For example if we have a package called `examples`, then any file in this package must be stored in a folder named `examples`.

Syntax of a package declaration

```
package packageName;
```

If we want a Java file to belong to a package, then we declare this at the start of the that file. If we have a class named `Test` and we want it to be a part of the package `examples`, then the first statement in the file `Test.java` should be `package examples;`. This also implies that the file `Test.java` should be stored in the folder `examples`. Most compilers will enforce this by showing an error when a package declaration and file location are not the same. In eclipse this will be done automatically for us.

If there are still too many files in a package we can separate the files into smaller packages. These packages are called **sub-packages**. For every lecture I prepare a lot of examples, if I put them all into the `examples` package there will be a lot of files. I can put my files from week 1 in a sub-package called `week1`. This means my declaration would be `package examples.week1;`. Any file in this package would be stored in a folder `week1` which is in a folder called `examples`.

Packages in Java library

Java uses the same idea for storing all of its libraries. There are a large number of packages and sub-packages, but the code contained inside each package is usually related. Here are some examples of packages and the code that they contain;

Package	Description
<code>java.io</code>	All of the Java code that supports input and output
<code>java.util</code>	All of the Java utilities, such as data structures and algorithms
<code>java.text</code>	Java code for processing and formatting text
<code>java.lang</code>	The core functionality of the Java language

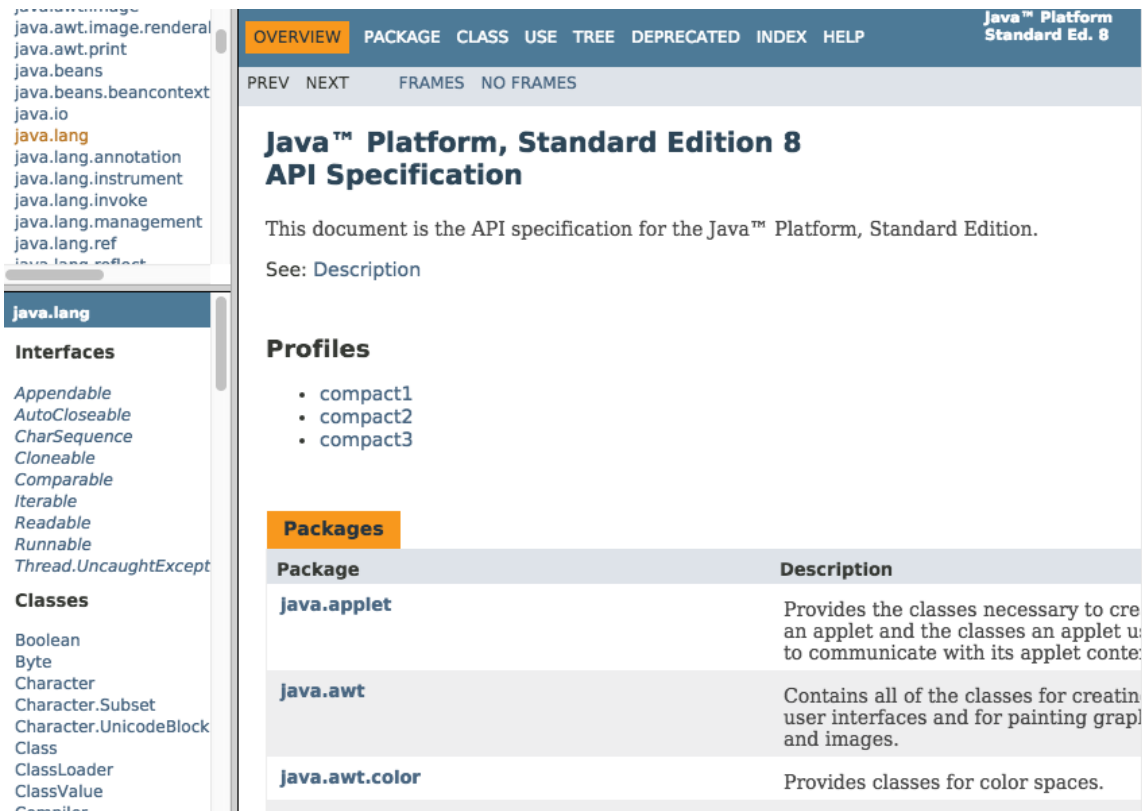


Figure 3.1: The basic layout of the Java API

3.7 Application Programmer Interface

The application programmer interface (API) is the list of all of the libraries made available in Java. It is called the application programmer interface because it is the interface between the application programmers (us) and the system programmers (people who design the Java language). The API contains a huge amount of information about all of the classes that make up the Java Language. For each class it also lists their **instance variables** and **methods**. There is also a description of each class and of each method. The API is available here <https://docs.oracle.com/javase/8/docs/api/>, although this address may change with newer versions of the language. There is also a version of the API where much has been translated into Chinese, however these are not always updated to the latest version.

Figure 3.1 shows the basic layout of the API documentation. There are three main components, in the top left is a list of packages contained in the library. When you want to read the documentation of a part of the library first select the package from this list. Once you have selected a package, all of the components of that package will be displayed below the package list. This includes class, interfaces and other Java files. From this list you select the class that you want and the documentation will be opened in the main part of the screen.

This documentation contains all of the information that we need to use any class in the library. For each class the documentation contains;

- A description of the functionality of the class
- A table to tell us what instance variables are in the class
- A table to tell us what constructors are in the class
- A table to tell us what methods are in the class

Constructors	
Constructor and Description	
Rectangle()	Constructs a new Rectangle whose upper-left corner is at (0, 0) in the coordinate space, and whose width and height are both zero.
Rectangle(Dimension d)	Constructs a new Rectangle whose top left corner is (0, 0) and whose width and height are specified by the Dimension argument.
Rectangle(int width, int height)	Constructs a new Rectangle whose upper-left corner is at (0, 0) in the coordinate space, and whose width and height are specified by the arguments of the same name.
Rectangle(int x, int y, int width, int height)	Constructs a new Rectangle whose upper-left corner is specified as (x,y) and whose width and height are specified by the arguments of the same name.

Figure 3.2: Constructors listed in Rectangle class

- A detailed **description** of each constructor and method

When we want to use one of the classes we have found in the API, we must complete the following steps.

1. We have to **import** the class into our code
2. We have to **construct** an object based on the class
3. We have to look up the descriptions of its methods

Importing a class

There are many classes in the API, when we choose to use one we must tell Java specifically which class we want to use. There are often many classes with the same name so we will need to specify the package and the name. This is specified at the beginning of the file after the package statement.

Syntax of import statement

```
import packageName.ClassName;
```

Assume that we have to complete a problem involving 2D coordinate geometry. We want to know if certain points are contained within certain rectangular areas. We have discovered a class in the package `java.awt` named `Rectangle`. This class contains some useful methods that we can use to solve the problem. At the beginning of the file we add `import java.awt.Rectangle;`. Now the class is available to be used in our code.

Constructing the object

As this is a class that we have not used before, we do not know how to construct it. The API contains all of the information we need to do this so we look at the constructors section of the Rectangle class. Figure 3.2 shows the constructors that are listed for the Rectangle class.

As we can see there are several options to choose from. In our problem we want to create a number of rectangles at different locations and with different sizes. As such the final constructor is the most suitable. We can see that the constructor requires 4 int parameters to construct a `Rectangle` object. This gives us code that looks something like this: `Rectangle r = new Rectangle(5, 5, 10, 20);`. Based on the description, this creates a `Rectangle` object at the position (5, 5) with a width of 10 and height of 20.

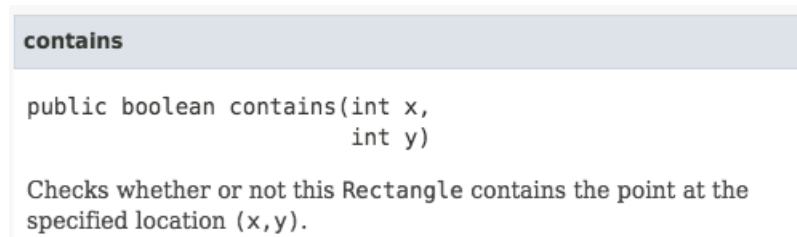


Figure 3.3: Contains method listed in Rectangle class

Using the object

Our problem requires us to determine if a number of points are inside a number of rectangular areas. We need to find a method that would be useful for this task. After searching through the methods in the API, we find a method called `contains`. Figure 3.3 shows the information available about the method `contains`. Remember all we need to know to call a method is a reference to the object (which we created in the previous step), the name of the method (which is shown in the documentation) and the parameters which we know will be two ints representing a set of x and y coordinates. Now we are ready to call the method and find out if the point is inside the rectangle.

```
boolean ans = r.contains(10, 10);
```

3.8 API Use Exercises

These exercises are all based on the idea of locating the required class in the

1. Define a class named `StringExamples`. This class should contain a main method with the following statements;
 - Declare a string variable named `myString` and assign it some value.
 - Print the contents of `myString` to the screen
2. Add statements to the main method of the `StringExamples` class, that complete the following actions;
 - Declare an integer variable named `stringSize`
 - Call the `length` method on the variable `myString` and store the result in the variable `stringSize`
 - Declare a string variable named `myMessage`.
 - Assign the value "Contains " the the variable `myMessage`
 - Use the concatenation operator (+) to combine the string `myMessage` and the variable `stringSize`. The result should then be stored in the variable `myMessage`.
 - Use the concatenation operator again to combine the string `myMessage` and the value " characters". The result should then be stored in the variable `myMessage`.
 - Print the value of `myMessage` to the screen
3. Add statements to the main method of the `StringExamples` class, that complete the following actions;
 - Declare a string variable named `secondString`.
 - Using the `replace` method on the variable `myString`, replace one of the words in the string with another word. The result should be stored in the variable `secondString`.
 - Declare a string variable named `secondMessage`.
 - Assign the value "Contains " the the variable `secondMessage`

- Using the concatenation operator, add the length of `secondString` to the end of the `secondMessage`. The length should be calculated directly and concatenated, not stored in a variable.
 - Use the concatenation operator again to combine the string `secondString` and the value " characters". The result should then be stored in the variable `secondString`.
 - Print the value of the variable `secondString`
4. Define a class named `ShapeExamples`. This question is based around the use of the `Rectangle`, `Dimension` and `Point` classes in the package `java.awt` in the API. This class will require import statements to use these classes. This class should contain a main method with the following statements;
- Declare a rectangle variable named `box`.
 - Using the keyword `new`, construct a rectangle object in the location (5, 15) and with a width and height of 10.
 - Declare a boolean variable named `inside`.
 - Use the method `contains` in the `Rectangle` class on the variable `box` to determine if the point (7, 8) is inside it. The result should be stored in the variable `inside`.
 - Using an if statement, print one of the following messages. "The point is inside the box" or "The point is outside the box"
 - Declare a dimension variable named `mySize`.
 - Construct a dimension object with the width and height 15 and 21, store this object in the variable `mySize`.
 - Using the methods `getWidth` and `getHeight` to access the width and height of the variable `mySize`, print both values to the screen.
 - Declare a point variable called `myLocation`
 - Construct a Point object with the location (55, 66) and store the object in the variable `myLocation`
 - Using the methods `getX` and `getY` to access the coordinates of the variable `myLocation`, print both values to the screen.
 - Declare a rectangle variable called `secondBox`.
 - Construct a rectangle object by passing the variables `mySize` and `myLocation` to the constructor. The object should be stored in the variable `secondBox`.
 - Declare two Point variables named `p1` and `p2`.
 - Construct two point objects with the locations (63,75) and (75,63) and store the objects in the variables `p1` and `p2` respectively.
 - Use the method `contains` in the `Rectangle` class on the variable `secondBox` to determine if the points `p1` and `p2` are inside it.
 - For each of the points, one of the following messages should be printed. Either "p1/2 is inside the box" or "p1/2 is outside the box"

3.9 Practical Exercises

1. Define a class in Java with the name `Time`.
2. Add three instance variables to the class named `hours`, `minutes` and `seconds`. These variables should use an appropriate integer type.
3. Define a class named `TestTime`. Add a main method to the class containing the following statements;

- Create an object based on the `Time` class named `morning`.
 - Set the value of each of the instance variables of `morning` to the time you awoke this morning.
4. Define a method in the `Time` class called `printTime`, the method should take no parameters and not return any data (return type is void). The method should print out the time in the format "`[HH:MM:SS]`" where HH is the number of hours, MM the number of minutes and SS the number of seconds. Each of the values should be printed with a leading 0 if the value is less than 10.
 5. Add another statement to the main method of the `TestTime` class that calls the `printTime` method for the time object `morning`.
 6. Add a constructor to the `Time` class. This constructor should take no parameters and set the value of all instance variables to 0;
 7. Add a second constructor to the `Time` class. This constructor should take three integers as parameters and use them to set the initial value of the instance variables hours, minutes and seconds.
 8. Add the following statements to the main method of the `TestTime` class;
 - Create a second object based on the `Time` class named `now`. The initial values of the instance variables should be set using the constructor, not individually and represent the current time.
 - Call the method `printTime` for the object `now`.
 9. Define a method in the `Time` class called `incrementHour`. This method should take no parameters. When the method is called, the time within the object should be changed to the start of the next hour. Remember that the value of hours cannot be above 23.
 10. Define a method in the `Time` class called `incrementMinute`. This method should take no parameters. When the method is called, the time within the object should be changed to the start of the next minute. Remember that the value for minutes cannot go above 59 and when this happens the hour should be changed. (Hint: this should be done by calling the `incrementHour` method)
 11. Define a method in the `Time` class called `tick`. This method should take no parameters. When the method is called, the time within the object should change by one second. Remember, that the value for seconds cannot go above 59 and when this happens the minute should be changed. (Hint: this should be done by calling the `incrementMinute` method)
 12. Add the following statements to the main method in the `TestTime` class;
 - Add a loop which will execute 5000 times.
 - Inside this loop call the `tick` method for the object `morning`.
 - Call the `printTime` method for the object `morning`.
 - Change the number of times that the loop is executed until the value of `morning` is the same as the value of `now`
 13. Define a method in the `Time` class called `sameTime`. This method should take a single `Time` object as a parameter. The method should return true if the instance variables in the object and the parameter are the same value and false if they are not.
 14. Add the following statements to the main method in the `TestTime` class;
 - Create two time objects, named `start` and `finish`. The initial values of the instance variables should be set using the constructor, not individually (Make up some random time values).

- Add a while loop that will continue executing as long as **start** and **finish** are not the same time.
 - Inside the loop you should call the tick method for the object with the earlier time.
 - Additionally you should count the number of times that the loop is executed.
 - Print out the number of seconds between the times **start** and **finish**.
15. Define a class in Java with the name **DateTime**.
 16. Add two instance variables to the class named **date** and **time**. The type of date should be the Date class that was specified earlier in the chapter and the type of time should be the Time class specified in the earlier questions.
 17. Add a constructor to the **DateTime** class. This class should take six parameters in the following order day, month, year, hour, minute, second. Using these parameters, both the date and time instance variables should be constructed.
 18. Define a method in the **Date** class called **printDate**, the method should take no parameters and not return any data (return type is void). The method should print out the date in the format "(DD/MM/YYYY)" where DD is the date, MM the month and YYYY the year. Each of the values should be printed with a leading 0 if the value is less than 10.
 19. Define a method in the **DateTime** class called **printDateTime**, the method should take no parameters and not return any data (return type is void). The method should print the date and time to the screen by first calling the **printDate** method for the date object and then the **printTime** method for the time object.
 20. Define a method in the **DateTime** class called **sameDateTime**. This method should take a single **DateTime** object as a parameter. The method should return true if the date and time are the same and false if they are not.

Chapter 4

Object-Oriented Programming

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand why we use object-oriented programming
- ☐ be able create multiple objects from the same class
- ☐ understand the relationship between classes and objects

4.1 Why Object-Oriented Programming

Chapter 3 introduced **how** to define classes and **how** to construct objects, this chapter will highlight **why** objects and classes are used. One of the main reasons for using object-oriented programming is **code reuse**. The idea is that code can be written in a way that can be used many times without changing it. Another reason for using object oriented programming is that it makes code easier to understand, maintain or change.

This chapter will show a number of examples that highlight the benefits of object-oriented programming, and are a little more fun than the normal problems that are normally set as assignments. By designing some parts of an old arcade game, some practical reasons for the use classes, objects and other object-oriented programming techniques are shown and how these can make code much easier to manage.

The basic functionality of the game should allow the player to perform the following actions;

- Move our ship
- Fire our laser
- Destroy incoming enemy ships

As the book progresses, the features of this game will be increased and changed to make the game more fun, or simply to make it easier to program. For each of the examples, you should perform the same steps in your copy of the game. At the end of the chapter and many others, there will be some additional tasks to be completed in order to demonstrate understanding of the concepts covered. To complete the game fully, you should perform each of these tasks before starting the next chapter.

4.2 The start of a basic arcade game

The game we are creating is called Space Invaders, it was one of the first really successful arcade games ever created. Before we can finish the game there are a lot of topics that we must learn about in order to complete some of the more difficult tasks. In this chapter, we will first begin programming the game in a procedural way (like C) instead of in an object oriented way.

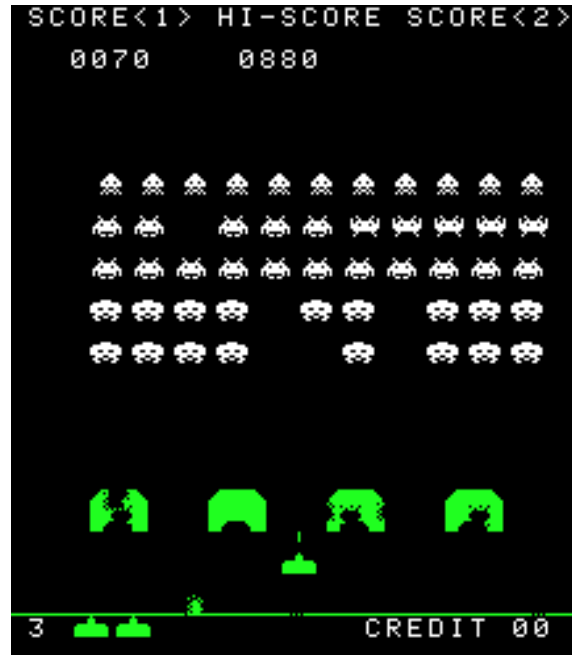


Figure 4.1: The original Space Invaders arcade game

Figure 4.1 shows the original arcade game. We will be creating a game that is very similar but is different in some small ways. For example, instead of the player having a number of lives, we will have a value for health. We will simplify the drawing of many of the shapes such as bullets and the defensive bunkers and change how damage happens to these. We will also remove the animations of the enemy ships to simplify the drawing process.

The first class to be defined is called **Game**, this will contain a main method and the basic logic of the game. To the class some instance variables are added to hold information like the location of the players ship and his score. In this situation, these instance variables will work just like global variables as long as we are working with a single class.

Representing basic information

Example: Variables required for knowing the players location and health

```

1 class Game {
2     int playerX = 50;
3     int playerY = 400;
4     int playerHealth = 20;
5 }

```

Next, the information about the enemy ships has to be represented. In the simple version, this information could be represented by variables for each of the ships with names like **enemy1X**, **enemy1Y**,... This is shown in the first example, and is not very easy to work with. Too many variables make it very easy to make mistakes. If more ships are added to the game new variable would have to be added for each of the things we need to remember.

Example: Variables required to remember the location of 2 enemies

```

1 int enemy1X = 70;
2 int enemy1Y = 50;
3 int enemy2X = 90;
4 int enemy2Y = 50;

```

A simpler solution would be to have an array for each piece of information. One array would contain the x coordinates of each enemy, another for the y coordinate of each enemy and so on. Adding a new enemy, now only requires changing the size of the array and adding values. This is a much easier solution to maintain, all that has to be remembered is that the first enemy is represented by index 0 of each of the arrays and so on.

Example: Enemy locations remembered in a collection of arrays

```

1 class Game {
2     int playerX = 50;
3     int playerY = 400;
4     int playerHealth = 20;
5
6     int numberOfEnemies = 3;
7     int [] enemyX = { 70, 90, 110 };
8     int [] enemyY = { 50, 50, 50 };
9 }

```

Testing our representation

The Game class that has been defined performs no actions yet, It only stores some information. The only thing that can be tested is that the information stored in the variables is the same as it should be. To do this some test code is added to the class, this code simply prints out the current value of each of the variables. This is done in a separate method that can be called to check our code is correct. The difficult part of testing code in this way is that in order to know if the code is correct or not, we have to know what output the code should produce.

Example: Testing the basic data stored in the game

```

1 class Game {
2     // variables declared and given values
3     void test() {
4         System.out.println("Player (" + playerX + ", " + playerY + ") has
5         health " + playerHealth);
6
7         for (int i = 0; i < numberOfEnemies; i++) {
8             System.out.println("Enemy " + i + " located at (" + enemyX[i] + ", "
9             + enemyY[i] + ")");
10        }
11    }
12
13    public static void main(String[] args) {
14        Game g = new Game();
15        g.test();
16    }
17 }

```

The example shows a main method (this is where our program starts), which creates an object based on the Game class and then calls a method on that object called test. The test method contains code to print the contents of all of our variables to the screen. When we look at the output, we see a line of text for the player and each enemy ship showing their location.

Example: Output of our test code

```

1 Player (50, 400) has health 20
2 Enemy 0 located at (70, 50)
3 Enemy 1 located at (90, 50)
4 Enemy 2 located at (110, 50)

```

To verify that the above test code is correct is easy. The result is compared to the expected result, for example the first line is supposed to print the location of the player and their health and in the output we can see that this is correct. Testing at this low level is not very common, but the process of displaying this information can be useful for seeing the result of any operations defined.

Adding functionality

The current code simply represents the player and the enemy ships. One of the requirements for the main player is to kill all enemy ships, so next we are going to add some code to get rid of one of the enemy ships when the player hits it. When the player shoots an enemy ship we need to remove the information about that ship from the arrays. This functionality is defined as a method that can remove the information about the ship *i* from each of the arrays. Because this code is defined as a method, it can be used many times when a ship must be removed.

Example: Ship details removed from both arrays when dead

```

1 void removeShip(int m){
2     // code to remove the location of enemy m from the arrays
3     if(m < numberOfEnemies && m >= 0){
4         for(int i = m; i < numberOfEnemies -1; i++){
5             enemyX[i] = enemyX[i+1];
6             enemyY[i] = enemyY[i+1];
7         }
8         enemyX[numberOfEnemies-1] = 0;
9         enemyY[numberOfEnemies-1] = 0;
10        numberOfEnemies--;
11    }
12 }

```

The code here is a little complicated here but fully understanding this is not the point, just seeing that we have implemented it is enough. For the curious, I will explain the code in detail. Because the size of an array cannot be changed, when a ship is removed there will be a blank space left in its place. First every ship location is copied back one space in the array. So if there are ships 0, 1, and 2 and ship 1 is removed, then ship 2 will be copied into the position of ship 1 and the number of enemies will be reduced. Finally, the last ship in index `numberOfEnemies` is removed by replacing its data with 0 and reducing the number of enemies. This is done because the last ship appears twice in the array after being copied. All other ships have the duplicate values replaced when the next ship is copied.

Testing the functionality

Once new functionality is added to a program, it is always a good idea to test that it works correctly. Assuming that you assignment works correctly and not testing it is a great way to get bad grades.

More statements are added to the test code already defined so that we can test the result of this method. First the `removeShip` method is called, then the process of printing out the details of the ships is repeated. If the process is completed correctly, there should be only two ships printed after the remove method is called.

Example: Code for testing the remove ship functionality

```

1 void test() {
2     System.out.println("Player (" + playerX + ", " + playerY + ") has
      health " + playerHealth);
3     for (int i = 0; i < numberOfEnemies; i++) {
4         System.out.println("Enemy " + i + " located at (" + enemyX[i] + ", " +
      enemyY[i] + ")");
5     }
6     removeShip(1);
7     for (int i = 0; i < numberOfEnemies; i++) {
8         System.out.println("Enemy " + i + " located at (" + enemyX[i] + ", " +
      enemyY[i] + ")");
9     }
10 }

```

Changing functionality

Changing requirements is a very common problem in software engineering. After some time working on the game, it is decided that players and enemy ships should have cooldown on their weapons. This means that all ships should have an integer number that counts down to when they can fire their gun again. This prevents players and enemies firing too quickly. This change requires that the code we have already written is changed. First we add the new instance variables, then we update our test code so that the value of cooldown is printed for every ship.

Example: Variables added to represent cooldown

```

1 int playerCooldown = 10;
2 int[] enemyCooldown = {15, 20, 0};

```

When we run the test there is a problem, the table below shows the information contained in the arrays both before and after enemy 0 is removed.

Before	After
enemyX: {70, 90, 110}	enemyX: {90, 110}
enemyY: {50, 50, 50}	enemyY: {50, 50}
enemyCooldown: {15, 20, 0}	enemyCooldown: {15, 20, 0}

We can see that for the two remaining enemies, they no longer have the correct value for cooldown. The cooldown value was not removed when enemy 0 was killed. This is because we forgot to update the `removeShip` method. This means that any time we add, remove or change the variables we use to represent the game, all of the functions will have to be checked to make sure they are correct. This is adding difficulty to our work so we decide that we are going to use object-oriented programming to solve our problem.

Recreated using OOP

Instead of having a different array for each piece of information about the enemy ships and individual variables for the player, we are going to design a class to represent this information. Every ship in

the game, including the player, has the same information (location, health and weapon cooldown). We can write a class that contains an instance variable for each of these pieces of information. Because we keep the same information about the enemy ships and the player, we can use the same class to represent both.

Example: The Ship class

```
1 class Ship {  
2     int x;  
3     int y;  
4     int cooldown;  
5     int health;  
6     Ship(int x1, int y1, int c, int h){  
7         x = x1;  
8         y = y1;  
9         cooldown = c;  
10        health = h;  
11    }  
12 }
```

The ship class simply groups together values for x, y, health and cooldown. It also provides a constructor to make it easier to create ship objects. More importantly, this changes how we represent the player and enemy ships in our Game class. To represent the player we only need a single variable, a ship object to store all of the players information. To represent the enemy ships, we only need a single array, each element will hold a ship object (which will store all of the information about a single ship).

Example: Variables used to represent player and enemy ships

```
1 Ship player = new Ship(50, 400, 20, 20);  
2  
3 int numberOfEnemies;  
4 Ship[] enemyShips;
```

The array of ships was declared by the above code, but we have not added any enemy ships to the array. This is because, instance variables are added directly to the class, but executable code (like adding an element to an array) needs to be within a method or constructor. Now we need to add a constructor to the Game class that adds the information about the enemy ships. The Game constructor does not require any parameters, because we already know the information we need to create the enemy ships, for each ship we pass in the required parameters and store the created object in the array.

Example: The constructor for the Game class

```
1 Game() {  
2     enemyShips = new Ship[numberOfEnemies];  
3     enemyShips[0] = new Ship(70, 50, 15, 5);  
4     enemyShips[1] = new Ship(90, 50, 20, 5);  
5     enemyShips[2] = new Ship(110, 50, 0, 5);  
6 }
```

Updating our tests

Now that we have changed how we represent the enemy information, we need also to update how we test that it works correctly. Previously we just printed the values of each variable, now the variables are contained inside classes. That means that instead of printing the variable `playerX`, we print `player.x` (which is the instance variable `x` inside the object `player`). Instead of printing out the array element `enemyX[i]`, we print `enemyShips[i].x`.

Example: Test updated for the object-oriented design

```

1 class Game {
2     // variables declared and given values
3     void test() {
4         System.out.println("Player (" + player.x + ", " + player.y + ") has
           health " + player.health);
5
6         for (int i = 0; i < numberOfEnemies; i++) {
7             System.out.println("Enemy "+i+" located at (" + enemyShips[i].x +
           ", " + enemyShips[i].y ") has health "+ enemyShips[i].health );
8         }
9     }
10    ...
11 }

```

Killing a Ship

Now that we have updated the code, we need to change the `removeShip` method. The structure of the code will stay exactly the same, but now instead of changing many arrays containing different values (`x`, `y`, `cooldown`), we only need to change the array containing the ship objects. Each ship already contains its own `x`, `y`, `health` and `cooldown`. When the array is changed all of this information is changed too.

Example: Updated `removeShip` method

```

1 void removeShip(int m) {
2     if (m < numberOfEnemies && m >= 0) {
3         for (int i = m; i < numberOfEnemies - 1; i++) {
4             enemyShips[i] = enemyShips[i + 1];
5         }
6         enemyShips[numberOfEnemies - 1] = null;
7         numberOfEnemies--;
8     }
9 }

```

This code is now easier to understand and to edit if required.

New Functionality

We have created some code for representing ships, both the player and the enemy. The next task we need is to add code to represent the bullets that these ships can fire at each other. Before we create the class, we need to think about what needs to be represented about each individual bullet

- Each bullet has a location (`x` & `y`)
- Each bullet is moving in a direction, up or down

- Each bullet has an amount of damage it can do

Later we will add more functionality, but for now we will implement these abilities in a class.

Example: Representing the attributes of a bullet

```

1 class Bullet{
2     int x;
3     int y;
4     boolean up; // true for up, false for down
5     int damage;
6     Bullet(int x1, int y1, boolean u, int d){
7         x = x1;
8         y = y1;
9         up = u;
10        damage = d;
11    }
12 }

```

The x and y coordinates are easily represented as integers. To represent the direction of movement we use a boolean, this variable can have only two possible values true and false. We can use true to represent a bullet that will move up the screen (fired by the player) and false to represent a bullet that moves down the screen (fired by an enemy). Finally, we use another int to represent the power of the bullet. All of these values must be given a value when the bullet is created, so they must be required by the constructor.

Now that we have created the bullet class, we need to add the functionality to the ship class to fire bullets. We will add a method for this, the method takes no parameters and will return either the bullet object or null if a bullet cannot be fired yet. The only problem we have is that we must know if the ship belongs to the player or to an enemy, this is how we will know if the bullet should move up the screen or down the screen. This must be added to the Ship class. We can again use a boolean value, true if this is the player and false if it is an enemy. This can then be used to tell the direction a bullet should travel.

Example: Ship class with fire method

```

1 class Ship {
2     int x, y, cooldown, health;
3     boolean player;
4     Ship(int x1, int y1, int c, int h, boolean p){
5         x = x1;
6         y = y1;
7         cooldown = c;
8         health = h;
9         player = p;
10    }
11    Bullet fire() {
12        Bullet b = null;
13        if(cooldown == 0) {
14            b = new Bullet(x,y, player, this);
15        }
16        return b;
17    }
18 }

```

This code simply checks if the `cooldown` value has reached 0 and if so creates a new bullet at the same location as the ship. Later when we know the details of the user interface, we can adjust

the position of the bullet so that it appears at the correct point of the ship. Important parts to note are that for the third parameter representing the direction the bullet should travel, we pass the `player` variable. This means that when the ship represents a player (value is true) the bullets will travel up the screen and when the ship is an enemy (value is false) the bullets will travel down the screen. Finally the last parameter to the constructor is a reference to the Ship object that fires the bullet. In this case we are using the implicit parameter to represent that the object running this code is the one who fired the bullet.

Did it hit?

Now both types of ships have the ability to fire bullets, what we need to check is if they have hit or not. We will add this code to the ship class, it will be responsible for checking if it was hit by a bullet or not. To do this we need to have an idea of the size of the ship, later when we design the interface we will do this more accurately, but for now we will use an assumed height and width for all ships.

Example: Method for checking if bullet has hit a ship or not

```
1 boolean isHit(Bullet b) {  
2     if(b.x >= x && b.x <= x + 10 &&  
3         b.y >= y && b.y <= y + 10) {  
4         return true;  
5     }  
6     return false;  
7 }
```

This method from the ship class assumes that the height and width of the ship are both 10. It then uses greater than and less than to check if the x and y coordinates of the bullet are inside the rectangle represented by the points (x, y), (x+10, y), (x+10, y+10) and (x, y+10). If the bullets coordinates are within this range, the value true is returned (this counts as a hit) otherwise it is a miss and false is returned.

Finally, we will add some methods to the classes to make our testing easier. These methods will print details about the values contained inside the instance variables of the different objects.

Example: Output details of ship objects

```
1 void printShip() {  
2     if(player) {  
3         System.out.println("Player located at (" + x + ", " + y + ") with  
4         health " + health);  
5     } else {  
6         System.out.println("Enemy located at (" + x + ", " + y + ") with  
7         health " + health);  
8     }  
9 }
```

Example: Output details of bullet objects

```
1 void printBullet() {  
2     if(up) {  
3         System.out.println("Bullet located at (" +x+ ", " +y+ ") moving up");  
4     } else {  
5         System.out.println("Bullet located at (" +x+ ", " +y+ ") moving down")  
6     };  
7 }
```

These methods can be used on the variables such as `player` or `enemyShips` to output the details easily.

4.3 Cohesion

One very important concepts in object oriented programming is **cohesion**. Cohesion in programming is about how much the elements of a component belong together. We are going to look at cohesion in terms of our classes. Thus, cohesion measures the strength of relationship between pieces of functionality within a given class. Classes are usually described as “high cohesion” or “low cohesion”. In a class with high cohesion, all of the functionality is closely related.

Classes with high cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

If we consider our example, these classes that we have defined so far are highly cohesive. The Ship class only has functionality relating to the ships represented in the game. The bullet class represents only contains functionality directly related to bullets. The ship class contains functionality for storing details about an individual ship (location, health and weapon cooldown), as well as some functionality to check if the ship has been hit by a bullet or not.

The only exception to this is the Game class. This class is intended to manage the overall game and as such has functionality storing information about all ships and managing this information by removing enemy ships after they are hit. However, it also contains methods for testing the the functionality is working correctly. This gives the class functionalities that are quite different. Normally, testing code is removed when a program is finished but in this case we are going to move all of this code to another class `TestGame`. This results in two more cohesive classes.

Example: Testing code moved to a separate class

```

1 public class TestGame {
2     public static void main(String[] args) {
3         TestGame t = new TestGame();
4         t.test();
5     }
6
7     public void test() {
8         Game g = new Game();
9         g.player.printShip();
10        Bullet pB = g.player.fire();
11
12        for (int i = 0; i < g.numberOfEnemies; i++) {
13            g.enemyShips[i].printShip();
14
15            if(g.enemyShips[i].isHit(pB)) {
16                System.out.println("hit");
17            } else {
18                System.out.println("miss");
19            }
20
21            Bullet b = g.enemyShips[i].fire();
22            if(b != null) {
23                b.printBullet();
24            }
25        }
26    }
27 }

```

But how does this make it easier to program? The main reason is this, it makes it easier at a high level to understand what each class in the system does and how they relate to each other. For example, because we know that the ship class is only responsible for maintaining information about individual ships, if we are modifying code related to the levels in the game, we know the ship class will probably not be effected by the changes that we have made.

Additionally, as a general rule highly cohesive classes will not grow to large and be difficult to maintain or understand. When a class is getting too large this might be a sign that we are being too broad in our definition of related functionality, and might want to separate the class into several more closely related classes.

Conclusion

We have seen that object-oriented ideas can be used to simplify code itself as well as the process of updating and testing it. As we learn about more object-oriented techniques, we will add more to this example.

4.4 Game Completion Tasks

Figure 4.2 shows a representation of the defensive bunker structure. It is made up of 27 individual bricks of differing sizes. These individual bricks must be represented separately and can be destroyed by either fire from the player or from the enemy ships.

1. Define a class called **Bunker**. This class will be used to represent the defensive structures that the player ship can hide under.

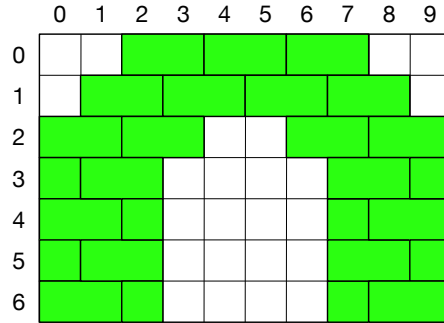


Figure 4.2: Representation of a defensive bunkers structure

2. Add instance variables to the **Bunker** class called **x** and **y**. These will be used to represent the location of the bunker on the screen.
3. Define a class called **Brick**. This class will be used to represent individual bricks in the defensive structures that the player ship can hide under.
4. Add instance variables to the **Brick** class called **x** and **y**. These will be used to represent the location of the brick on the screen.
5. Add an instance variable to the **Brick** class called **width**. This will represent the width of this brick.
6. Add an instance variable to the **Bunker** class called **bricks**. This variable should be an array to remember all the individual brick objects.
7. Define a constructor in the **Bunker** class. The constructor should require two parameters that specify the x and y coordinates of the bunker. These values should be remembered in the variables **x** and **y**. Additionally, the array **bricks** should be created to hold 27 bricks, but the individual brick objects will not be created yet.

Chapter 5

Encapsulation

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the meaning of access specifiers
- ☐ be familiar with the 4 access specifiers and their effect
- ☐ understand the concept of encapsulation
- ☐ understand the use of getter and setter methods
- ☐ to be able to implement getter and setter methods
- ☐ be able to identify the public interface of a class
- ☐ be able to tell the difference between a static variable and an instance variable, and understand how the use of either effects a class
- ☐ understand how to define and call static methods

5.1 Visibility

In large programming projects, a programmer is usually only working with a small part of the overall program. The programmer will make use of the classes that others have provided by calling their methods and other programmer will make use of his code by calling its methods. When others are using our classes we want to make sure that they are using it in the way that we designed them.

Access levels

Java uses **access level modifiers** to give other programmers different levels of access to different parts of our code. Access levels can be applied to variables, classes and everything else we define in our programs. There are 4 access levels:

- public
- protected
- default (sometimes called package)
- private

The access level that we apply is usually called visibility. The define exactly what other programmers can see when they are using our code. To set the access level of something we have defined, we just need to put the correct keyword before it.

- `public int day;`
- `protected int year;`
- `int month;`
- `private int hour;`

Note

The level of access called default is when **no** access modifier is defined

The following table shows, for each access level, where the item can be accessed from:

Modifier	From the same class	From the same package	From a subclass	From everywhere
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

public

When we define something as public, this means that it can be accessed, used or changed by any other object in the entire program. For example if I have a reference to a Ship object (and all of its instance variable are declared public), then I can simply change the value of any of these instance variables. I can set the x coordinate of the player to -12312, his weapon cooldown to 9000 or make any other changes that don't make sense.

It is pretty obvious that a ship should not have a value for x below 0. So we should probably not declare all instance variables as public.

protected

When something is declared as protected, this means that it can only be accessed from within the same class, package or by a subclass of this class. We will be learning about what exactly a subclass is in chapter 7, so we can fully understand protected then.

default

When no access specifier is defined, this means that it can be accessed from within the same class or package. This is often called package level visibility because that is where it can be accessed from. This is what we have declared all of our variables as so far. This can be a useful level of visibility when the program has a good package structure.

private

Private is the most restrictive access level. When something is defined as private, it can only be accessed, changed or used from within the same file. If we declare the instance variables of the Ship class as private, then other programmer working on the game can only access or change the values by calling the methods that we have defined.

Defining access levels

There are 4 levels of access that we can allow for every instance variable, class or method. But how do we know which level should be used? It comes down to a single question, who needs to see or change this?

Classes and Constructors

When it comes to classes, we will mostly declare them as public. The biggest reason for this is that, when we create a class it is supposed to be used by other classes in different parts of the program. There are types of classes that you might declare as protected or private, but we will not cover these until later in the book.

Methods

Methods are often declared with different access levels. If we define a method that we expect others to use then it will be declared public. If we define a method that we do not want others to use, but we would like to use it in other parts of our code, then we declare it default. If we write a method that is only use from within our class then it should be declared as private.

Instance variables

When declaring instance variables of a class, we think backwards. Most instance variables should be declared as private. Only where an instance variable needs to be used by another class would we use any other access level.

This may seem unusual, but it makes it easier for programmers to work together. This idea is called information hiding, where we hide our instance variables and only allow them to be accessed or changed using methods we have written. This means that we keep control over the way they can change. Information hiding is part of an idea know as **encapsulation**.

5.2 Encapsulation

Encapsulation in programming languages is the combination of two ideas, the grouping together of instance variable and methods into classes and the ability to restrict access to some of the objects components. We have already studied defining classes containing both instance variables and methods, lets examine the idea of restricting access.

This process is known as information hiding, and the main purpose behind it is the idea of a **black box**. A black box is a device, system or object which can be viewed in terms of its inputs and outputs, without any knowledge of its internal workings. Almost anything might be referred to as a black box: a transistor, algorithm, or the human brain.

A very simple real world example of this is a car. I know the idea of how a car works, I can provide the input to make the car move, stop, turn, etc.. But when I press the accelerator, I do not know how the engine performs the task of moving.

This idea is often used in programming, consider the following example where a game is being developed by a large group of programmers. One group is responsible for the code that defines a ship and what it can do. Another group may be working on the overall game logic. The second group do not need to know how the ship information is stored, they only need to know what method they can call to access or change this information. This means that I could completely remove the instance variable `x` from the class and store the value in a file or data based instead. When a `getX` method is called, i read the value from the file, when a `setX` method is called, i write the parameter into the file.

Example - Bullet class

Lets apply these ideas to the Bullet class. The main change is that all of our instance variables are declared as private, and the class and methods as public.

Example: Access levels applied to the bullet class

```
1 public class Bullet {  
2     private int x, y;  
3     private boolean up; // true for up, false for down  
4     private int damage;  
5     public Bullet(int x1, int y1, boolean u, Ship o) {  
6         x = x1;  
7         y = y1;  
8         up = u;  
9         origin = o;  
10    }  
11    public void printBullet() {  
12        ...  
13    }  
14 }
```

What is the effect of this change? Other classes can no longer change the values of our instance variables, but they also have no access to them. This means that the method `isHit` in the ship class will no longer work. So how can we let other parts of the program know the values without allowing them to change? This can be done by adding new methods to the Bullet class.

Getter methods

A getter method, sometimes called an accessor method, is a simple method that gets the value of an instance variable and returns it. Getter methods are only required when your instance variable are not accessible. In the bullet class, the `x` and `y` instance variables are not accessible from outside the class, so we add getter methods to the class to return these value.

Example: Getter methods for the name instance variables in bullet class

```
1 public int getX() {  
2     return x;  
3 }  
4 public int getY() {  
5     return y;  
6 }
```

What is important here is that the values of `x` and `y` cannot be changed, we can only find out what they are. This is all that we need for the `x` variable, as it is not supposed to be changed. The `y` variable is different, the value of `y` will change as the game progresses, it will travel up or down the screen. In this case a getter method may not be enough.

Setter methods

Now we need to be able to change the value of `y`. This can be done using a setter method, sometimes called a **mutator** method. A setter method is a little more complicated than a getter method. We define a method that we can call when we want to change the value. At a basic level, a setter method will take a parameter, and use this value to change the instance variable. The reality can sometimes be a little more complicated, if we take the example of the `y` coordinate of a bullet, this should be changed, but only by the amount specified in the game and in the correct direction. This means that the method does not need a parameter, and will use some code to decide the new value.

Example: Setter method for the y instance variable

```

1 public void move() {
2     if(up) {
3         y = y - 2;
4     } else {
5         y = y + 2;
6     }
7 }

```

This example shows a small change to the value of y when this method is called. Depending on the value of the variable up, the value will either be increased or decreased to move it up or down the screen.

Using getters and setters

Now we have added getter and setter methods to our bullet class we can fix the errors we created in the ship class by making these instance variables private. Every reference to the instance variables x or y has to be replaced with the appropriate getter or setter method.

Example: Use of getter and setter methods in place of directly accessing the values

```

1 public boolean isHit(Bullet b) {
2     if(b.getX() >= x && b.getX() <= x + 10 &&
3         b.getY() >= y && b.getY() <= y + 10) {
4         return true;
5     }
6     return false;
7 }

```

In this example, and all others, accessing an instance variable is easily replaced. For the instance variables x and y, we simply replace any occurrence of accessing the value with calling the getter method, e.g. `.x` is replaced with `.getX()`. Replacing the changes of the value is a little more difficult, because we must replace a variable assignment with a method call. For example, if we changed the value of x `.x = expression;` then we would replace it with `.setX(expression);`. This is not required for this example.

5.3 Public Interface

Through applying the ideas of encapsulation, we have essentially split our classes into two separate parts. The first part is what we allow other programmers to see and use, this is called the **public interface**. Here interface means the point where two systems meet and interact. Because we have hidden some of our code, other programmers can only interact with the code we allow them to. The second part is the actual way in which all the code is written, this is called the **private implementation**.

The public interface of a class is all of the methods, constructors and instance variables that are declared as public. The public interface of the Bullet class is very simple;

- `public Bullet(int x, int y, boolean u, int d)`
- `public int getX()`
- `public int getY()`

- `public void move()`
- `public void printBullet()`

These are the only methods that can be used to interact with a bullet object, we can create it, find out its location, move it and print it's details. Lets apply the same idea to the Ship class, we end up with a very small public interface.

- `public Ship(int x, int y, int c, int h, boolean p)`
- `public void printShip()`
- `public Bullet fire()`
- `public boolean isHit(Bullet b)`

This allows other programmers the ability to create ship objects, fire bullets, check if bullets have hit this ship and print out their details. But there is no way for them to know the location of the ship or their current level of health or weapon cooldown. Obviously we may need to add getters for these methods, but do we need to add setters?

This choice is based on how the variables will be used. Is it likely that other objects will change the location of the ship? No, we do not need to add a setter method for `x` or `y`. How is the cooldown variable likely to change? It will usually go down by a pre-set amount over time and is likely to have constraints placed on the possible values such as not going below 0 or above 100. As such it will not have a direct setter method but will be changed by some other methods. Similarly the value of health will drop when we are hit by a bullet, but there should not be unrestricted access to the value.

Lets examine the idea of taking damage, the primary purpose is that the health of the ship will be reduced by some amount. To know how much, we will need a parameter to tell us. The parameter should not be negative, and the minimum value for health is 0. This method will print a message let us know what happened.

Example: Method for reducing the health of a ship

```

1 public void takeDamage(Bullet bullet) { // this is how much damage we
    should take
2     String message;
3     if (bullet.getPower() > 0) { // it must be a number greater than 0,
        healing will be done by another method
4         if (health - bullet.getPower() > 0) { // our health should not be a
            negative number, so we check what the answer will be first
5             health = health - bullet.getPower(); // if it is positive we
                store the value
6             message = "Ship has taken " + bullet.getPower() + " damage,
                health is now " + health;
7         } else {
8             health = 0; // if it is negative we are dead
9             message = "Ship is destroyed.";
10        }
11    } else {
12        message = "Damage cannot be a negative value";
13    }
14    System.out.println(message);
15 }
```


5.4 Class/Static Variables

Having studied the idea of encapsulation and how it applies to the definition of instance variable, classes and methods, we will now look at **class** variables. We know that an instance variable has a separate value for each object based on a class, class variables share a single value over all objects based on the same class. That means that when I have 5 objects based on the same class, they will always have the same value for a class variable and when one object changes the value it is changed in all of the others.

This is because a class variable is only created in memory once, whereas an instance variable will be created once for every object created. All objects use the same piece of memory for the class variable, so it is impossible for the values to be different.

The syntax for declaring a class variable is to add the keyword **static** before the type of the variable. Class variables can have any level of visibility. Class variables are not very common, on their own, but let's look at an example where one might be used. Let's assume that there are many components of the game that contain ships and all of these except the player are enemies. We would like a statistic showing the total number of enemies in the game. How can we calculate this? Ships may be created by different programmers working on different levels.

This can be calculated using a class variable. First we add a static class variable to the class of type `int` called `numShips` and give it the initial value of 0. This is where we will store the answer, we will make it private so it cannot be changed by other programmers. Second, we add a small bit of code to the constructor of the class, every time a ship object is created we increment the class variable `numShips`. When the first ship is constructed the value is changed to 1, when the second object is constructed the value is changed to 2 and so on. Lastly we add a getter method to return the value after all of the code has executed.

Example: Code added to ship class

```
1 private static int numShips = 0;
2
3 public Ship(int x, int y, int c, int h, boolean p) {
4     ...
5     numShips++;
6 }
7
8 public int getNumShips() {
9     return numShips;
10 }
```

Using a class variable externally

When a class variable is public or default, it can be accessed directly from other classes without an object based on that class. Because class variables are connected to the class and not to each individual object, these variables are referenced using the class name, and no object is required to access or change the value. For example, to access the above class variable (if it was public) we would use the syntax `Ship.numShips`.

There are many examples of useful class variables that we can use when performing calculations. For example, when performing calculations related to circles or spheres, we could use the value of the class variable `PI` from the `Math` class e.g. `double area = Math.PI * radius * radius;`. Additionally, `MAX_VALUE` and `MIN_VALUE` define the largest possible value and smallest possible value for many of the different primitive values. These are stored in the classes `Integer`, `Float`, `Double` etc.

5.5 Constant Variables

Constant variables are variables that can only be assigned a value once. This means that once the value of the variable has been set, it will never change over the lifetime of the program. The syntax for declaring a constant variable is to add the keyword **final** before the type of the variable.

If code within the program attempts to change the value of this variable, it will be highlighted as a compiler error. This can be useful for preventing mistakes in code where a variables value should not change. Using final variables can also prevent some problems from happening when writing more complex programs that can perform multiple operations at the same time (we will be looking at this in chapter 14).

The most common use of the final for variables, is in combination with static. This defines a variable that is connected to the class and cannot be changed. These types of variables are often referred to as **constants**. They are very useful in programming for representing facts that do not change. We can rely on the use of constants in our calculations because the value will always be the same.

A good example of useful constants, would be to represent values such as the screen bounds. This could then be used in multiple locations, for example in a **move** method of any class that will be displayed on the screen. In this example, the value of bounds of the screen are stored as constants in the **GameScreen** class. This constants are then used in any method that changes the position of the ship, to check if the new poistion is inside the bounds of the screen.

Example: Move method using constants from the **GameScreen** class

```

1 public void move(int xm, int ym) {
2     int x1 = this.x + xm;
3     int y1 = this.y + ym;
4     if(x1 > 0 && x1 + this.width < GameScreen.SCREEN_WITDH && y1 > 0 &&
5         y1 < GameScreen.SCREEN_HEIGHT){
6         this.x = x1;
7         this.y = y1;
8     }
9 }
```

The use of the constant ensures that methods will always be consistent. If it is decided that the maximum value for x should be 100 instead of 512, we do not need to search through the code and find and change everywhere 512 is used. Instead, we just change the definition of the constant, this value is then automatically used in all methods.

Variable naming for constants

When a variable is declared constant (static and final) it is generally shown in how the variable is named. Constant variable names are declared using only upper-case characters. Rather than use camel case as we do in instance variables, every word in the name of a constant is separated by an underscore (_). For example **SHIP_WIDTH**, **SCREEN_HEIGHT**, **SCREEN_WIDTH** etc...

5.6 Class Methods (static methods)

It is also possible to associate a method with a class rather than an object. Because class methods are not connected to objects, it is not possible to use instance variables within a class method. A class method is declared by adding the keyword **static** before the return type of the method. All access levels can be used with class methods, but public is the most common.

Lets add some useful methods to our class that can be used from anywhere in our code. In many games, the result of an action performed is partly random and partly based on the other values.

Because we will want to use this functionality in many places we will create a new class with static methods that we can use. In older games these random values were calculated by throwing a number of dice. We will put this functionality in a class called `Dice`.

As the methods we declare will be as class methods, they will not be able to access the value of instance variable. All of the variables in the `Dice` class will have to be declared as class variables. The only variable we need is an object based on the `Random` class from the package `java.util`, which we will use to roll our dice.

Example: Class simulating the roll of dice

```

1 public class Dice {
2     private static Random rng = new Random();
3
4     public static int roll1D6() { // name means roll 1 6 sided dice
5         int d = rng.nextInt(6); // generate a number between 0 and 5
6         return 1+d; // add one to the value so the range is between 1 and 6
7     }
8
9     public static int roll2D6() { // roll 2 six 6 dice and add the values
10        int d = roll1D6() + roll1D6(); // call the method and add the
11        results
12        return d; // return the calculated answer
13    }
14 }

```

Now we are required to call this method in order to calculate our random values. Just as with class variables, class methods are called in a different way. We do not need an object in order to call a class method. For example, if we want to get a random number between 1 and 6, we would use the syntax `Dice.roll1D6()`. Here the **name of the class** (`ClassName.methodName(parameters)`) is used instead of the name of the object, as is the case with instance methods (`objectName.methodName(parameters)`).

Example: New private method added to calculate the effects of our actions

```

1 private int randomEffect(int effect){
2     int r = Dice.roll2D6();
3     if(r == 2) {
4         effect = 0; // terrible attempt, no effect
5     } else if(r < 6){
6         effect = (int) (effect * .6); // weak attempt, reduced effect
7     } else if(r > 8 && r < 12){
8         effect = (int) (effect * 1.5); // strong attempt, increased effect
9     } else if(r == 12){
10        effect = effect * 5; // perfect attempt, big effect
11    }
12    return effect;
13 }

```

In this example, we simply call the method and get access to a random integer in the correct range. This method can then be called to find out the effect of an attempt to heal or to attack an enemy.

The main method

We have now studied all of the required theory to explain the definition of the main method that we have use in all of our programs.

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

public - This is the access level of the method. This must be public because the program will be started by the JVM, which is outside of our code.

static - This is a class method. This must be a class method because when the program is started there are no instances of any classes.

void - This is the return type of the method. You should have known this before you started.

main - This is the name of the method. This must always be the same or the JVM does not know how to start the program.

String[] args - This is the parameters that are passed to the program. This is the same as the equivalent in C, however it is not optional in Java.

Testing

Class methods are also useful for testing purposes where we do not want to create a new object just to call some method. The following example shows how to call a class method to begin the testing process. Because the main method is in the same class as the static/class method `test`, we do not need to use the name of the class when we call it.

Example: Testing the game with class methods

```
1 public class TestGame {
2     public static void main(String [] args) {
3         test();
4     }
5
6     public static void test() {
7         // code for testing
8     }
9 }
```

5.7 Enumerated Types

There are many different types of data that can be represented in Java, this includes the primitive types such as `int` and types defined by interfaces or classes such as `String`. We have seen that we can create new types by defining classes or interfaces, but it is also possible to define them using an enumerated type. An enumerated type, usually called an **enum**, is a type with a fixed list of possible values. These values are specified when the enum is defined.

Syntax: Declaration of an enumerated type

```
1 enum EnumName { list of possible values }
```

Enums can be defined separately within their own file, or within another class. However they cannot be declared inside a method within a class. The name of the enum can be any identifier, and this becomes the name of the enum type. We generally use the same naming conventions for the name of an enum as we do for the name of a class. Each value within the list of possible values must also be an identifier, and these are separated by commas. By convention, because the possible values of an enum are constants, they are given names with uppercase characters.

For example, if we were defining a class to represent cards in a game. We might store the suit of a card as a string, i.e. "diamond", "club", "heart" or "spade". However, it is possible that the variable we are using can store any value of string. The value stored may be different than we expect and may cause problems in our program. Instead, if we use an enumerated type, the only values that can be stored in the variable are correct.

Example: The Suit type defined as an enum

```
1 public enum Suit { DIAMOND, CLUB, HEART, SPADE }
```

Now in our card implementation we can represent the suit of the card using `Suit cardSuit;`. It is only possible that the variable `cardSuit` can store one of the possible values defined in the `Suit` enum. To give the variable a value we must tell it which value to be `cardSuit = Suit.DIAMOND;`. Notice that we have used the possible value `diamond` as a constant (we did not create it like an object using the `new` keyword). Additionally, because the possible value is inside the `Suit` enum, we must use the name `Suit.DIAMOND` and not simply `DIAMOND`.

Adding functionality to enums

Within the implementation of Java, enums are actually represented as a class. Now this is not very important to how we define and use enums, with the exception of one ability. Because classes can have methods, so can enums. Enums already contain methods that can be useful such as `compareTo`, which allows us to determine the ordering of two of the same enum, `equals` which allows us to check if two enums are the same, `toString` which tells us the name of the possible value stored in this variable and `ordinal` which tells us the unique number associated with this possible value.

Lets look at an example for the game. There are three distinct types of enemy ships that attack in the space invaders game. Each of these types of ship has a different shape, height and width and are worth a different amount of points when destroyed.

Example: Basic AlienType enum

```
1 public enum AlienType {  
2     A, B, C  
3 }
```

Next, we will add some instance variables and methods to these types. To do this we need to add a little more functionality to the enum. When adding variables or methods to an enum, the syntax is a little different;

Syntax: Declaration of enum with method and variable

```
1 enum EnumName { list of possible values;  
2     type instanceVariable;  
3     EnumName(params) { ... }  
4     public Type methodName(params) { ... }  
5 }
```

Firstly, the list of possible values must be followed by a semicolon, following the list of possible values we can declare instance variables and can add constructors and methods.

Example: AlienType enum with methods and variable

```

1 public enum AlienType {
2     A(10, 7, 10), B(8, 8, 20), C(10, 8, 15);
3     private int width;
4     private int height;
5     private int score;
6
7     private AlienType(int w, int h, int s) {
8         width = w;
9         height = h;
10        score = s;
11    }
12
13    public int getWidth() {
14        return width;
15    }
16
17    public int getHeight() {
18        return height;
19    }
20
21    public int getScore() {
22        return score;
23    }
24 }

```

In this example, we associate an integer value with the width, height and score of each of the alien ship types. The values are set using the constructor, which must be **private**, during the listing of the enumerated types. Additionally, getter methods are also provided to allow the values of the variables be accessed.

5.8 Practical Exercises

1. Define a class named **Counter**, containing a single class variable named **count**, which is an **int**.
2. Define a method named **increment** in the class **Counter**. This method should take no parameters and return no value. The method should increase the value of **count** by one.
3. Define a method named **reset** in the class **Counter**. This method should take no parameters and return no value. The method should increase the value of **count** to 0.
4. Define a method named **getCount** in the class **Counter**. This method should take no parameters and return an **int**. The method should tell us the value of the variable **count**.
5. Define a class named **Test**, this class should contain a main method. In the main method the following operations should be performed.
 - Declare and construct an array of counter objects of size 4.
 - Construct a counter object for each element of the array.
 - Iterate through the array and call the increment method once for each object.
 - Iterate through the array **in reverse** and print the value of the counter in each element.

Your output should look something like this:

Count of index 3 = X

```
Count of index 2 = X
Count of index 1 = X
Count of index 0 = X
```

6. Define a class named **MathHelper**.
7. Define a class method named **min** in the class **MathHelper**, this should take three ints as parameters and return whichever value is the smallest.
8. Define a class method named **max** in the class **MathHelper**, this should take an array of integers as a parameter and return whichever value is the largest.
9. Add the following statements to the main method in the **Test** class.
 - Use the **min** method in the class **MathHelper** to determine the smallest of three values. Store this value in a variable and then print it to the screen.
 - Declare an array of integers using list initialisation containing 5 to 10 numbers.
 - Use the **max** method in the class **MathHelper** to determine the largest value in the array. Store this value in a variable and then print it to the screen.

5.9 Game Completion Tasks

1. Modify all of the classes so that they are declared as public and all of the instance variables are declared as private. Add getters and setters where required. Add visibility keywords to all of the methods in the classes too.
2. Add a constant to the **Bunker** class called **BRICK_SIZE**, give it the initial value of 5. This will be the width in pixels of our smaller bricks.
3. In the **Game** class, add an array of **Bunker** objects and construct it in the constructor to hold 4 bunkers.
4. In the constructor of the **Game** class, create 4 bunker objects at the following coordinates and remember them in the array. (100,350), (200, 350), (300, 350), (400, 350)

Chapter 6

Interfaces

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ be able to declare and use interface types
- ☐ understand the concept of polymorphism
- ☐ be able to convert between interface types and classes

6.1 Code Reuse

One of the biggest issues with the development of software is the cost. This cost is reflected in the number of hours programmers spend coding to produce a program. In order to increase productivity, programmers want to be able to reuse software components in multiple projects. However, changes are often required to code in order to make its reuse in the future possible.

This chapter introduces an important strategy for separating the reusable part of a program from the parts that are different each time it is used. This strategy is based on interfaces. An interface is a description of the actions that an object can do... for example when you flip a light switch, the light goes on, you don't care how, just that it does. In object oriented programming, an interface is a description of all functionality that an object must have in order to be an "X". Think of the example of a light, anything that "behaves like" a light, should have a `turnOn` method and a `turnOff` method.

Common code

It is often possible to make a functionality available to a wider set of inputs by focusing on the important operations that the functionality requires. We can use an interface to describe these important operations that be common in our code.

Lets look at an example of a class that is designed to calculate the average and maximum of a set of numbers. This class would require a method for each of the following pieces of functionality:

- A method to add new numbers to the set
- A method to get the average value
- A method to get the maximum value

Example: Implementation of data set calculator

```
1 public class DataSet {
2     private double sum;
3     private double max;
4     private int count;
5     public void add(double x){
6         sum += x;
7         if(count == 0 || x > max){
8             max = x;
9         }
10        count++;
11    }
12    public double getAverage(){
13        if(count == 0){ return 0;
14        } else { return sum / count; }
15    }
16    public double getMax(){ return max; }
17 }
```

The above example class contains all of the necessary method to implement the required functionality for a data set of double values. It is very useful, but it cannot be used for anything except a set of doubles. We could not use it to find the same information about bank accounts or coins without making a lot of changes to the class. If we made the required changes to make this class work for bank accounts or coins, then it will no longer work for doubles.

Example: DataSet class modified to work for bank accounts

```
1 public class DataSet {
2     private double sum;
3     private BankAccount max;
4     private int count;
5     public void add(BankAccount x){
6         sum += x.getBalance();
7         if(count == 0 || x.getBalance() > max.getBalance()){
8             max = x;
9         }
10        count++;
11    }
12    public double getAverage(){
13        if(count == 0){ return 0;
14        } else { return sum / count; }
15    }
16    public BankAccount getMax(){ return max; }
17 }
```

Example: DataSet class modified to work for coins

```

1 public class DataSet {
2     private double sum;
3     private Coin max;
4     private int count;
5     public void add(Coin x){
6         sum += x.getValue();
7         if(count == 0 || x.getValue() > max.getValue()){
8             max = x;
9         }
10        count++;
11    }
12    public double getAverage(){
13        if(count == 0){ return 0;
14        } else { return sum / count; }
15    }
16    public Coin getMax(){ return max; }
17 }

```

The two previous examples show the data set class modified, the max is now stored as bank account or coin objects and the parameters or return types of most of the methods have changed. It is obvious that these can no longer be used to work on doubles. However, it is clear from the structure of the code that the algorithm for the data analysis is the same for all types of data. Only the details of the actual **measurement** are different. In the original, we could use the value of the parameter, in the bank account example we needed to use the method `getBalance` to access the value and in the coin example we had to use method `getValue`.

What we would like is a single class that provides this functionality to any object that can be measured. Lets assume that the designers of the various classes agree on a single method to use. The method `getMeasure` will be used to access the balance in the bank account and the value in the coin. Now we can use the exact same algorithm to implement the data set functionality by using `getMeasure`.

Example: Code section of algorithm using getMeasure

```

1 sum += x.getMeasure();
2 if(count == 0 || x.getMeasure() > max.getMeasure()){
3     max = x;
4 }
5 count++;

```

6.2 Interfaces

This section of code shows how the algorithm can be implemented when we know what method to use for different classes. This assumes that we either had access to change the code, or could ask the original creators to do so. But we have a bit of a problem with the rest of the class, what type should the variables `x` and `max` be? Ideally, the type would be any class that contains the method `getMeasure`, but there is no way to specify this in Java (and most other OOP languages).

In Java we are required to define an interface to solve this problem. The basic functionality of an interface is to specify methods that are required by this interface. The syntax for declaring an interface is:

Example: Syntax of an interface declaration

```
1 interface InterfaceName{  
2     // requirements are specified here  
3 }
```

Lets examine an real example that can help us solve our problem. We define an interface called **Measurable**, in the requirements we will specify a method called **getMeasure** that returns a value of the type double.

Example: The measurable interface

```
1 public interface Measurable {  
2     double getMeasure () ;  
3 }
```

This declaration specifies that any object that want to take the place of **Measurable** must contain the a **public** method with the same name, return type and parameters. In general, interfaces can require many methods to be implemented. Just like a class, an interface in Java has an associated access level, here in this case we have specified that this interface is public. We do not need to specify the access level of any of the methods in an interface, because these are all automatically public.

Interfaces and classes

The definition of interfaces and classes can seem similar, both are declared in a file with the same name and both contain the details of methods. However there are several important differences;

- Methods in an interface cannot be implemented, this is called an abstract method.
- All methods and variables in an interface are automatically public
- You cannot create an instance of interface
- Interfaces cannot contain instance variables, all variables in an interface are constants (declared as static and final)

Implementing an interface

Now that we have defined the **Measurable** interface, we can rewrite the **DataSet** class. The max will be stored as the type **Measurable** and this will also be the type of the parameter of **add** and the return type of **getMax**.

Example: Data set class changed to handle anything measurable

```

1 public class DataSet {
2     private double sum;
3     private Measurable max;
4     private int count;
5     public void add(Measurable x){
6         sum += x.getMeasure();
7         if(count == 0 || x.getMeasure() > max.getMeasure()){
8             max = x;
9         }
10        count++;
11    }
12    public double getAverage(){
13        if(count == 0){ return 0;
14        } else { return sum / count; }
15    }
16    public Measurable getMax(){ return max; }
17 }

```

We note that from the code above, the only parameter that the data set will accept is the type **Measurable**. But how does Java know if a class we are using is measurable or not. It can't, we have to specify that there is a relationship between this class and interface. The relationship that we use for interfaces is "behaves like". In our example a bank account is "behaves like" measurable, because the bank account class contains the required method. When a class is "behaves like" particular interface and specifies this relationship we say the class **implements** the interface.

This relationship must be specified in the class, the syntax of the relationship is:

```

public class ClassName implements InterfaceName{

```

For our bank account example, that would look like:

```

public class BankAccount implements Measurable{

```

Once we have specified this relationship in the class, it will not compile if it does not contain the required method. If the return type is different, the name misspelled, the parameters changed or the access level incorrect, the class will no longer compile. This does not mean that the code must be fully working, just that the method is correctly defined.

Implementing multiple interfaces

In the real world, an object can "behave like" a large number of things. For example, the game chess is "behaves like" a game because it is something you can play, it is also "behaves like" a learning tool because playing chess improve your reasoning abilities. This same idea also applies to the relationship between classes and interfaces, for example a bank account might implement both the measurable interface and also the taxable interface.

Implementing multiple interfaces in a single class is done by defining a list of interfaces separated by commas rather than a single interface name. There can be any number of interfaces specified in the list. The syntax for this is:

Example: Syntax of implementing multiple interfaces

```

1 public class ClassName implements IfName1 , IfName2 , ... {
2     ...
3 }

```

Here is an example of the BankAccount class implementing both the Measurable and Taxable interfaces:

Example: Implementing multiple interfaces

```
1 public class BankAccount implements Measurable, Taxable {  
2     ...  
3 }
```

Common code expressed as interface

Interfaces are at their most useful, when they are used to express what different classes have in common. This common functionality can be exploited to save time implementing the same functionality for all of these classes. The interface class does not know what classes implement it, neither does any class that uses the interface. As such the data set class could be used to find the average of a set of student results or the max of a number of game scores without needing to be changed.

In practice, one programmer will write functionality that works for a particular interface. He makes the interface and code available and any programmer who wants to use that code only needs to implement the interface. A large amount of common programming problems are solved in this way.

6.3 Interfaces As Types

Our data set class will now function for an class that implements the measurable interface. Lets look specifically at how this is happening, the method add requires a measurable type as a parameter and the bank account class “behaves like” a measurable type. This means that the method will actually accept any type that “behaves like” measurable, as such passing a bank account object to the method is allowed.

When an object of any type that “behaves like” measurable is passed to the method, the type of the object is forgotten. All of the code in the method do is use methods that are specified in the measurable interface. For example, even if the bank account has methods such as `getAccountNumber` or `getInterest`, we can only use them if the same method also appears in the measurable interface. It is also possible to directly store an object as any of the interface types that it implements.

Example: Using a bank account as a measurable type

```
1 BankAccount ac = new BankAccount(12345);  
2 Measurable meas = ac;  
3  
4 double money1 = meas.getMeasure(); // correct  
5 double money2 = meas.getBalance(); // error
```

This example shows the reference of the bank account type being stored as the measurable type. Once this is done we cannot use the measurable variable to call any of the methods that are not available in its interface. As such calling the method `getMeasure` works, but calling the method `getBalance` will not compile.

Example: Attempting to store a string as a measurable type

```
1 Measureable m = new BankAccount(1000); // ok  
2 Measureable m2 = new String(); //not ok
```

Using types like this can only be done if the relationship is defined. Because the bank account class states that it implements the measurable interface (and implements the required method) we

can store a bank account object in a measurable type variable. However, if this relationship does not exist for the class, then storing an object as an interface type is not possible. The example above shows first a bank account being stored as the measurable type, this is allowed because the relationship exists between the class and the interface. Secondly, the code attempts to store a string object as the measurable type, however the same relationship does not exist so this code does not compile.

What can be difficult to remember is that the actual object in memory is always a bank account object, it is only the type of the variable that determines what methods we can call. It is possible to change the type of the reference that we are using, such that it matches the type of the object (or some other type that is compatible). This process is called **type casting**.

Type casting can be applied to any expression, such as converting doubles to int, but is more useful when dealing with object types. The syntax of type casting is the following **(NewType)expression**. Here the expression is either the result of some calculation or the name of some variable, the NewType is whatever type we would like that expression to be. The result is then usually store in a variable of the type NewType, which can then be used to call any method of the NewType class or interface.

Example: Typecasting from measurable to bank account

```
1 Measureable m = dataset.getMaximum();  
2 BankAccount b = (BankAccount) m;
```

In this example, we have used the dataset to analyse a number of bank account. After we have finished we want to find the largest bank account, however the return type of the method is Measureable. We know that the object is actually a bank account object so we apply a typecast to the variable m and store the result in a variable b. Here both m and b are just references and both are pointing to the same location in memory, the only difference is the type of the reference. Once this is complete the variable b can be used as a bank account object.

Safer Typecasting

In the previous example, we were able to typecast the variable m because we knew that only bank account objects we put into the data set, therefore only a bank account object could be returned. However, this is not often the case, sometimes we do not know exactly what type of data is returned by the method. In these situations typecasting can cause serious problems in our programs, if we incorrectly typecast something to the wrong type our program will fail.

Because of this danger, Java provides a way for us to check that we know the type before we attempt to typecast it. The operator **instanceof** can be used to check if the typecast we want to apply will be allowed. Lets assume that we have a variable of the type X and we would like to typecast it into the type Y. The boolean operator instanceof can be used to check this, e.g. X instanceof Y will return true if the object X either actually is a Y object or behaves like an interface Y. The code can be used in the following way to safely typecast.

Syntax: Using instanceof to safely typecast

```
1 Y newVar;  
2 if(X instanceof Y){  
3     newVar = (Y)X;  
4 }
```

The instanceof operator will return true if the typecast is allowed and false if it is not, when combined with an if statement like this it will prevent your program from crashing. However, if the typecast is not performed then the variable newVar will have a null reference.

Example: Using instanceof to safely typecast

```
1 BankAccount account;  
2 if(max instanceof BankAccount){  
3     account = (BankAccount)max;  
4 }
```

Here we see an example where an object `max` is first checked to see if it is compatible with a bank account object and then typecast to that type. Again, if the type of `max` is not compatible, then the conversion will not be performed and the reference `account` will contain `null`.

6.4 Polymorphism

Assume that we have been asked to develop a program that will manipulate a number of different shapes in two-dimensional geometry. We might want to define an interface to capture what is common between these different shapes. The shapes will all be represented by a number of attributes such as `x`, `y`, `width`, `height` etc., however different shapes will represent this in different ways. A circle will be represented by its centre and a radius, a rectangle by the location of one point and its width and height. These are not common to all shapes. However, all of the shapes can be moved by some distance in the `x` and `y` axis, this functionality is common to all of the shapes in a two-dimensional environment. Secondly, all shapes will have an area which can be calculated. Lastly, for each shape, we can check if it contains a particular point.

Our shape interface will specify the requirements that objects that behave like shapes should be able to move, calculate an area and determine if a point is contained inside their boundaries.

Example: The Shape interface

```
1 public interface Shape{  
2     public void move(int x, int y);  
3     public double getArea();  
4     public boolean contains(int x, int y);  
5 }
```

As noted earlier, each of the classes will be implemented in a different way. Lets consider how the Circle class might be implemented. A circle is generally represented as a point and a radius, this means that we need three integers to represent its location and size. These integers should be private and passed to the object through a constructor.

Example: Circle implementation of the Shape interface

```

1 public class Circle implements Shape {
2     private int x, y, radius;
3     public Circle(int x1, int y1, int r){
4         x = x1;
5         y = y1;
6         radius = r;
7     }
8     public void move(int x1, int y1){
9         x = x + x1;
10        y = y + y1;
11    }
12    public double getArea(){
13        return Math.PI * radius * radius;
14    }
15    public boolean contains(int x1, int y1){
16        return false; // To be completed
17    }
18 }

```

Now let's consider how we would implement a rectangle shape. A rectangle has 4 corners, any of these would work to represent the location, it is easiest to choose the corner closest to the origin (only positive values will be used). Knowing this point the location of the rectangle can then be calculated knowing only the width and height. Again, these four values should be private and passed using a constructor.

Example: Rectangle implementation of the Shape interface

```

1 public class Rectangle implements Shape {
2     private int x, y, width, height;
3     public Rectangle(int x1, int y1, int w, int h){
4         x = x1;
5         y = y1;
6         width = w;
7         height = h;
8     }
9     public void move(int x1, int y1){
10        x = x + x1;
11        y = y + y1;
12    }
13    public double getArea(){
14        return width * height;
15    }
16    public boolean contains(int x1, int y1){
17        return false; // To be completed
18    }
19 }

```

We have seen two different implementations of the shape interface, both of which implement methods in different ways. If we have a variable of the type Shape, What version of the `getArea` method is called?

To answer this question it is important to understand that there are no objects in Java with the type Shape. Shape may be the type of the reference, but the actual object located in memory will be a circle or rectangle object (or some other implementation of the shape interface). It is often

not possible to know what type the actual object is by simply reading the code, this can only be known by executing the code.

Dynamic method lookup

Dynamic method lookup, sometimes called late binding, is where the actual type of the object and the correct method to use is done while the code is being executed. This process allows us to write code where we do not know exactly what type of object will be use, but we can know that the correct version of a method will be executed. The use of dynamic method lookup allows a programming technique called polymorphism.

Polymorphism

The term “polymorphism” comes from the Greek polys, “many, much” and morph, “form, shape”, approximately this means many forms. This relates to the fact that an object could contain any of the types that are related to it by a “behaves like” or “is a” relationship.

The main benefit of polymorphism is that all implementations of an interface can be treated in the same way. For example if we want to calculate the area of all of the shapes in an array, we can loop through the array and call the `getArea` method. Each object will use the correct method, but we do not need to specifically know what type of object it is. This simplifies code and makes it much easier to understand.

Interfaces and Enumerated Types

In the last chapter, we studied the idea of enumerated types and we saw that they can contain methods and variables. Enums can also implement interfaces. Normally, this is only used to have different enums implement a single interface so we can use polymorphism to store different enumerated types in the same variable.

6.5 Practical Example

Lets look at an example of the use of interfaces from our game. There will be a number of actions we might wish to perform for a number of different types of objects. Foe example, we will want to check if objects have been hit by a bullet or not, this includes ships as well as bunkers. We will want to move a lot of objects, this includes ships and bullets. Additionally, we might want to draw objects on the screen, ships, bullets, bunkers etc.

These different tasks all need to be applied to different types of objects. It would be inefficient to write code separately to perform each action for each different object type. Instead we will define a number of interfaces that capture what is common about the different objects.

Hittable

The first interface will define the methods common to all of the objects that can be hit by bullets. Primarily this includes the ship objects, bullet objects as well as the bunkers that we have not yet defined. Common code here is firstly checking if a bullet has hit or not, secondly inflicting the damage on the object, and thirdly telling us if the object has been destroyed or not. Finally, objects that we destroy may increase the score of the player so there will be a method to tell us how many points the object is worth.

This gives us an interface with only four methods that need to be defined.

Example: The Hittable interface

```
1 public interface Hittable {  
2     boolean isHit (Bullet b);  
3     void takeDamage (Bullet b);  
4     boolean isDestroyed ();  
5     int getPoints ();  
6 }
```

Firstly, every object that behaves like hittable must be able to tell us true or false if a particular bullet has hit it. Secondly, it must be able to tell us true or false if it has been destroyed. Finally it must tell us how many points it was worth.

The first class we can implement the Hittable interface for is the Bullet class. Although it is unlikely, two bullets may hit and destroy each other, as such we should check for this. Firstly, we have to decide the size of the bullets. This is a value we can change later but for now we will define a constant to represent the width of a bullet in our calculations. Secondly, to simplify our calculations, we are going to make use of some of the built in classes in Java. Specifically, we will use the Rectangle class in the `java.awt` package. This includes methods for determining if there is overlap (they hit) between two rectangles (our bullets). Finally, we need to have a way of knowing if the bullet was destroyed or not, this can be remembered by adding a new instance variable called `hit`. When the bullet has been hit it is destroyed.

Example: Bullet class implementing the Hittable interface

```

1 import java.awt.Rectangle;
2
3 public class Bullet implements Hittable {
4     ...
5     public static final int BULLET_WIDTH = 5;
6     private Rectangle box;
7     private boolean hit;
8
9     public Bullet(int x1, int y1, boolean u, int d) {
10         ...
11         box = new Rectangle(x, y, BULLET_WIDTH, BULLET_WIDTH);
12         hit = false;
13     }
14     public void printBullet() { ... }
15     public int getX() { ... }
16     public int getY() { ... }
17     public void move() { .. }
18     public int getPower() { ... }
19
20     public boolean isHit(Bullet r) {
21         if (this != r) {
22             if(box.intersects(r.box)) {
23                 return true;
24             }
25         }
26         return false;
27     }
28     public boolean isDestroyed() {
29         return hit;
30     }
31     public int getPoints() {
32         return 0;
33     }
34     public Rectangle getHitBox() {
35         return box;
36     }
37     public void takeDamage(Bullet b) {
38         hit = true;
39         b.hit = true;
40     }
41 }

```

The important parts in this implementation are:

- Line 11: The construction of the rectangle object representing the area of the bullet object
- Line 21: Here we check that the address of *r* is not the same as this object. This makes sure that I am not comparing the location of this bullet to itself, this would always be a hit.
- Line 22: Here we use the `intersects` method in the rectangle class to check if there is any overlap between the areas that define these two bullets.
- Lines 38 & 39: Here the `hit` boolean is set to true for both bullets, this means they will be classified as destroyed and no longer part of the game.

The second class we can implement this interface for so far is the ship class. We have already written the code to check if a bullet has hit the ship or not, but we should now change it to check

for the larger shaped bullets we are now using. This can be done by adding a rectangle object to represent the shape of the ship and using the same intersect code as in the bullet class. After this we only need to add the other two methods to implement the interface.

We can implement `isDestroyed` by checking if the value of the variable `health` is equal to 0, if this is true we have been destroyed, otherwise we have not been destroyed. To simplify this method we return `health == 0`, this will be calculated when the method is called and returned as a boolean value. Finally, we have to calculate the score. If the player is destroyed we get no points, however we know that alien ships have different types and these determine how many points we get. This has not yet been implemented in the class so for now we will just return 5 for an alien ship.

Example: Ship class implementing the Hittable interface

```

1 public class Ship implements Hittable {
2     private int x, y, cooldown;
3     private boolean player;
4     private int health;
5     public static final int SHIP_WIDTH = 10;
6     private Rectangle box;
7     public Ship(int x1, int y1, int c, int h, boolean p) {
8         ...
9         box = new Rectangle(x, y, SHIP_WIDTH, SHIP_WIDTH);
10    }
11    public Bullet fire() { ... }
12    private void takeDamage(Bullet bullet) { ... }
13    public void printShip() { ... }
14
15    public boolean isHit(Bullet b) {
16        if(box.intersects(b.getHitBox())) {
17            takeDamage(b);
18            return true;
19        }
20        return false;
21    }
22
23    public boolean isDestroyed() {
24        return health == 0;
25    }
26
27    public int getPoints() {
28        if (player) {
29            return 0;
30        }
31        return 5;
32    }
33 }

```

Later we will implement this interface in the bunker class.

Movable

Next we want to represent what is common about the objects that can be moved. The movable objects will all have a different action performed when they are moved. For instance the players ship will look for keyboard interaction from the user, alien ships will move along a predictable path and bullets will move in a straight line. What is common then about these is only the fact that they can be commanded to move. So we will only add one method that takes no parameters and returns no value, this will just change the position of the object.

Example: The Movable interface

```

1 public interface Movable {
2     void move();
3 }

```

For now, we will only implement this in the Bullet class. We must cover more topics before we can implement the interface in other classes. What already have the move methods defined, but this does only updates the y coordinate and does not change the rectangle object we use to check if we have hit something.

Example: Bullet class implementing the Movable interface

```

1 import java.awt.Rectangle;
2
3 public class Bullet implements Hittable, Movable {
4     ...
5     public Bullet(int x1, int y1, boolean u, int d) {
6         ...
7         box = new Rectangle(x, y, BULLET_WIDTH, BULLET_WIDTH);
8     }
9     public void move() {
10         if (up) {
11             y = y - 2;
12         } else {
13             y = y + 2;
14         }
15         box.y = y;
16     }
17 }

```

Here all we have changed is that the y variable in the rectangle object is also updated when the move method is called.

Drawable

A final piece of common code that we can identify is objects that can be drawn on the screen. These objects will all be located somewhere on the screen and have a predefined shape. As the overall look and feel of the game is classic and pixelated we will define the shapes of the objects in terms of a number of rectangles. These rectangles can then be drawn on the screen by our UI code when we complete this.

Example: The Drawable Interface

```

1 import java.awt.Rectangle;
2
3 public interface Drawable {
4     Rectangle[] getShape();
5 }

```

Figure 6.1 shows the typical layout of the bunkers in the space invaders game. The individual rectangle in the picture are highlighted with black outlines as they can be individually hit and destroyed by bullets. This means we will also have to think about making the bunkers and the individual bricks hittable too.

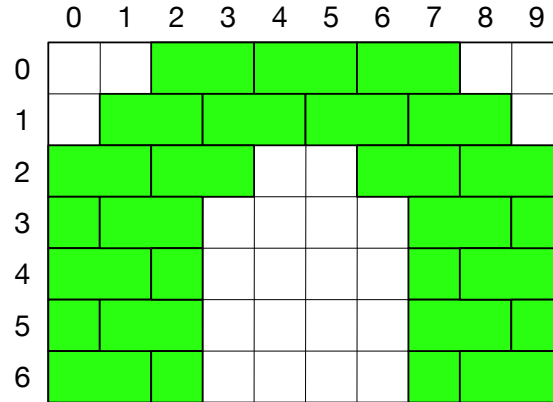


Figure 6.1: Typical layout of the defensive bunkers in the game

Therefore our first step will be to implement the hittable interface in the **Brick** class. This will be left as a task to complete.

Once the **Brick** class is implemented we can start the **Bunker** class. The **Bunker** is an object that can be drawn on the screen and can also be hit by bullets, therefore the class will implement both the **Drawable** and **Hittable** interfaces. The class need to remember its location as well as remember all of the bricks in the bunker.

Example: The Bunker class implementing hittable and drawable

```

1 public class Bunker implements Drawable, Hittable {
2     private int x, y, noBricks;
3     private Brick[] bricks;
4     public Bunker(int x, int y) {
5         this.x = x;
6         this.y = y;
7         bricks = new Brick[46];
8         addBricks();
9     }
10    private void addBricks() {
11        // Create the individual bricks and put them in the array
12    }
13    public int getNoBricks() {
14        return noBricks;
15    }
16    public Rectangle[] getShape() {
17        Rectangle[] shapes = new Rectangle[noBricks];
18        for (int i = 0; i < noBricks; i++) {
19            shapes[i] = bricks[i].getHitBox();
20        }
21        return shapes;
22    }
23    public boolean isHit(Bullet r) {
24        // check here to see if it has hit any of the bricks
25        // if the brick is hit, remove it from the array and reduce
26        // noBricks
27        return false;
28    }
29    public boolean isDestroyed() {
30        return noBricks == 0;
31    }
32    public int getPoints() {
33        return 0;
34    }
35 }

```

The bunker class remembers each of the individual bricks in an array. The drawable interface is implemented by asking each of the individual bricks for their shape and returning all of these shapes stored in an array.

6.6 Updated Interfaces

In the past years Oracle has been changing some of the details about how interfaces work, this section is to discuss some of the more recent functionality. Interfaces in Java 8 onwards now can have static and default methods. These changes make interfaces more capable and makes them more useful.

Static Methods

Static methods can be added to interfaces that allow some utility code to be executed. Because interfaces cannot have instance variables or non-final static variable, the methods are still somewhat limited but still useful.

Static methods work just like a static method in a class. We can define the method to perform some kind of processing or output some information or something. The following example defines a static method to compare two shapes based on their area.

Example: Static method in the Shape interface

```

1 public interface Shape {
2     void move(int x, int y);
3     double getArea();
4     double getOutline();
5     static Shape getLargest(Shape s1, Shape s2) {
6         if (s1.getArea() > s2.getArea()) {
7             return s1;
8         } else {
9             return s2;
10        }
11    }
12 }
```

We can see that the code here is very simple, it only does some calculations based on the parameters and returns the result. This method can then be used like a static method in a class.

Example: Using static method from Interface

```

1 public static void main(String[] args) {
2     Shape[] shapes = new Shape[3];
3     shapes[0] = new Circle(5,5,5);
4     shapes[1] = new Square(5,5,5);
5     shapes[2] = new Rectangle(5,5,5,10);
6
7     Shape temp = Shape.getLargest(shapes[0], shapes[1]);
8     Shape largest = Shape.getLargest(temp, shapes[2]);
9     System.out.println("Largest shape has area of " + largest.getArea());
10 }
```

Default Methods

Default methods allow for some methods to be defined in an interface, but used in the classes that implement the interface. Again, these are not the same as method in a class, because interfaces cannot have instance variables. But they can allow a class to use a method that is not defined in the class.

Example: Default method in the Shape interface

```

1 public interface Shape {
2     void move(int x, int y);
3     double getArea();
4     double getOutline();
5     static Shape getLargest(Shape s1, Shape s2) {...}
6     default void printArea() {
7         System.out.println("My area is " + getArea());
8     }
9 }
```

Here the keyword **default** is used to preface the method and it prints a message to the screen. This method will be based on the area calculated by the **getArea** method in the class that implements the interface.

Example: Using a default method from an interface

```

1 public static void main(String[] args) {
2     Circle c = new Circle(5, 5, 5);
3     Square s = new Square(5, 5, 5);
4     Rectangle r = new Rectangle(5, 5, 5, 10);
5     c.printArea();
6     s.printArea();
7     r.printArea();
8 }

```

In this example, none of the classes **Circle**, **Square** or **Rectangle** have the definition of the method **printArea**. This is only in the interface. Yet the method is called on each of the objects and the messages are printed to the screen.

6.7 Practical Exercises

1. Define an interface named **Vehicle**
2. Define a method named **getName** in the interface **Vehicle**
3. Define a method named **getTopSpeed** in the interface **Vehicle**
4. Define a method named **getMaxPassengers** in the interface **Vehicle**
5. Define a class named **Car** which implements the interface **Vehicle**. All methods required by the interface should be implemented. The values returned by these methods should be stored as instance variables in the class, and originally passed to in using a constructor.
6. Define a method named **getWeight** in the class **Car**. This method will return the value for the weight of the car (in tonnes), which should be stored as an instance variable and passed in using the constructor.
7. Define a class named **Ship** which implements the interface **Vehicle**. All methods required by the interface should be implemented. The values returned by these methods should be stored as instance variables in the class, and originally passed to in using a constructor.
8. Define a method named **getDisplacement** in the class **Ship**. This method will return the amount of water (in tonnes) displaced by the ship when it is floating, which should be stored as an instance variable and passed in using the constructor.
9. Define a class named **VehicleTest**, this class should define a main method containing the following statements;
 - Declare and construct an array of **Vehicle** objects named **vehicles** for storing 5 objects
 - Construct a ship object with the name "Titanic", maximum passengers of 3327, top speed of 39 km/h and a displacement of 52310 tonnes. This object should be stored in the array.
 - Construct a car object with the name "Toyota Avensis", maximum passengers of 5, top speed of 200 km/h and a weight of 2.02 tonnes. This object should be stored in the array.

- Construct a car object with the name "Lamborghini Gallardo", maximum passengers of 2, top speed of 325 km/h and a weight of 1.485 tonnes. This object should be stored in the array.
 - Construct a ship object with the name "Nimitz", maximum passengers of 5680, top speed of 58 km/h and a displacement of 100020 tonnes. This object should be stored in the array.
 - Construct a car object with the name "Mazda 3", maximum passengers of 5, top speed of 190 km/h and a weight of 1.3 tonnes. This object should be stored in the array.
10. Define a class method named **fastestVehicle** in the class **VehicleTest**. This method should take an array of vehicle objects as a parameter and return the name of the fastest vehicle.
 11. Add statements to the main method of the **VehicleTest** class to call the **fastestVehicle** method (passing the array **vehicles** as the parameter) and store the result in a String named **fastest** and then printed to the screen.
 12. Add to the main method of the **VehicleTest** class to complete the following steps
 - Loop over each element in the **vehicles** array and, using the instanceof keyword print one of the following messages
 - For a ship object "This ship displaces XX tonnes of water"
 - For a car object "This car weighs XX tonnes"
 13. Complete the **contains** method for the **Rectangle** class given in the chapter.
 14. Complete the **contains** method for the **Circle** class given in the chapter. The static method **sqrt** in the **Math** class can be used to calculate the square root of a number.

6.8 Game Completion Tasks

1. Implement the **Hittable** interface in the **Brick** class. This should use the same technique as the ship class. The width of the brick is `width * Bunker.BRICK_SIZE` and the height of the brick is simply `Bunker.BRICK_SIZE`. A brick is automatically destroyed if it is hit and the player gets no points for destroying it.
2. Implement the method **getHitBox** in the **Brick** class. This method should take no parameters and return a rectangle object that is the shape of this brick. This should be a newly created object, not the object that we use to check if a bullet has hit us.
3. Implement the **Hittable** interface for the **Bunker** class. The bunker will only return that it has been destroyed when all bricks have been destroyed. There are no points available for destroying the bunker. When the **isHit** method is called, the code should loop through the array of bricks and check if the bullet hits each one of them. It is possible for a bullet to hit multiple bricks. When all bricks are checked, the bricks that have been hit should be removed from the array.

Chapter 7



Inheritance

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the concept of inheritance in programming
- ☐ be able to extend classes and alter inherited functionality
- ☐ understand the concept of polymorphism, and the use of the `instanceof` keyword with inheritance
- ☐ understand the concept of abstract classes and method

7.1 What is Inheritance?

We saw in the previous chapter that exploiting the common code between classes can lead to benefits from code reuse. Often separate classes will share much common functionality, this is another area we can exploit code reuse.

Lets look at a real world example with bicycles. Mountain bikes, road bikes and tandem bike all share certain characteristics of bicycles. For example, when modelling them we would possibly model their current speed, pedal rate or what gear they are in. But these bicycles also have additional features that make them different from each other. For example, tandem bikes allow two riders, road bikes have drop handlebars and mountain bike often have larger tyres and suspension systems.

In the real world we often associate objects to each other by means of a hierarchy. For example, my bicycle could be described as a mountain bike but it shares the above common features with other types of bicycle. As such it could also be simply described as a bicycle, or even more generally as a vehicle. All of these descriptions are true, it performs all of the functions of a bike, so a mountain bike “is a” bike, it also performs all of the functions of a vehicle (getting me from place to place), so a mountain bike “is a” vehicle.

Figure 7.1 shows a representation of a hierarchy of different types of vehicles. In this example, we classify vehicles as either cars, bicycles or aeroplanes. These different classifications all share the same common traits of vehicles (the ability to get you from one location to another) but are different in how this is achieved. When we look at the hierarchy one level down it is the same. For example, there are different classes of aeroplanes, fighter jets, passenger jets and micro lites. All three of these classes of aeroplanes are vehicles that can fly, but they are different in other ways.

This idea can be applied to object oriented programming in the same way. **Classes** can inherit shared functionality and behaviour from an other class.

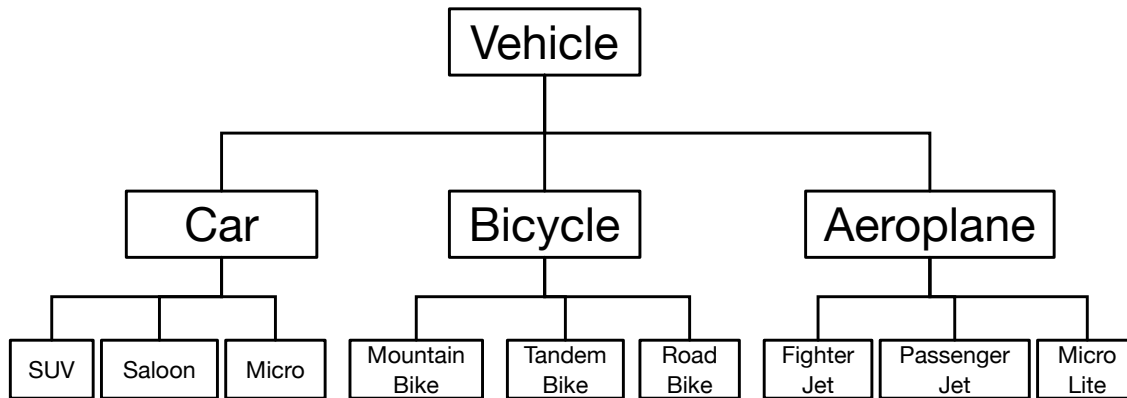


Figure 7.1: Representation of the hierarchy of vehicles including bicycles

Properties of inheritance in Object-Oriented Programming

Hierarchies in object-oriented programming have a number of properties. Firstly, the higher you go up the hierarchy, the more general or less specific the class becomes. For example, a bike is a more general idea than a mountain bike and a vehicle is a more general idea than a bike. Conversely, the lower down a hierarchy you go the more specific the class is.

Each class can only be connected to one more general class above it in the hierarchy. This is where inherited functionality will come from. Each class can be connected to an unlimited number of more specific classes below it in the hierarchy.

Classes inherit much of the state and behaviours of all the classes directly above them in the hierarchy. This means that if the bike class has the functionality to remember and change pedal rate, then the mountain bike class also has this functionality **automatically**. Additionally, because this rule also applies to the classes or concepts higher in the hierarchy, we also inherit functionality from the classes above the class directly above us. In this example that means that if the vehicle class has functionality for measuring and changing speed, then the bike class will automatically have this functionality too, and because the bike class has this functionality, the mountain bike class also has this functionality.

Creating inheritance

Inheritance in object-oriented programming is created through relationships between classes. This relationship can be described as “is a”. For example, a mountain bike “is a” bike and a bike “is a” vehicle. The classes in these example are called the **subclass** (sometimes called child class) and the **superclass** (sometimes called parent or father class).

The **superclass** is the class directly above in the hierarchy, there can be only one superclass. A subclass is one of the classes below in the hierarchy, there can be many subclasses. This relationship between subclass and superclass can only be defined in the subclass. In our example we have the idea of the vehicle and bicycle classes, and the relationship between them is a bicycle “is a” vehicle does not make sense the other way around i.e. a vehicle “is a” bicycle. So in this case, the vehicle would be the superclass and bicycle the subclass. This is defined using the keyword **extends** in the declaration of the bicycle class: `class Bicycle extends Vehicle`, i.e. `class` subclass `extends` superclass.

The consequence of this is that the bicycle class now has direct access to use all instance methods and variables of the vehicle class that are declared as public, default or protected. Now any code that we **add** to this class will only be related to this being a bicycle and not a vehicle. This reduced the amount of code in individual classes and makes them easier to understand and modify. Because we are **adding** code, we say that the bicycle class **extends** the functionality of the vehicle class.

Lets look at a real example of inheritance in use. The class A contains a single method named `sayHi`, this just prints a message to the screen. The class B is defined as a subclass of A, this

means that it contains all of the functionality of A but also adds another method called `sayBye`.

Example: A and B class definitions

```
1 public class A {
2     public void sayHi() {
3         System.out.println("Hi!");
4     }
5 }
```

```
1 public class B extends A {
2     public void sayBye() {
3         System.out.println("bye");
4     }
5 }
```

Now we have two classes that we can test. We can create an object based on each class and call their methods. Using an object based on the class A, we should be able to call the `sayHi` method, but because B is a subclass of A we should also be able to call this method using an object based on B. However, an object based on the class A inherits no functionality from the class B, as such it cannot use the method `sayBye`.

Example: Testing the use of inherited methods

```
1 public static void main(String[] args) {
2     A a = new A();
3     B b = new B();
4
5     a.sayHi();
6     b.sayHi();
7
8     b.sayBye();
9 }
```

The above example shows the `sayHi` method being called by objects based on both classes A and B, even though it is only defined as part of the class A. As state the method `sayBye` only exists in the class B and can only be called using an object based on that class.



Access levels

Instance variables and methods declared in a superclass can be used in a superclass if they are declared as either default (if it is in the same package), protected or public. However, if the access modifier is default (if it is not in the same package) or private we will not be able to access or change the variable.

This means that our subclasses can only change a private instance variable through the use of one of the classes public methods. If this does not allow the required level of control over the variable, we must change the access level of the instance variable to another access level in the superclass. However, if we wish to keep to the ideal of encapsulation, instance variables should only ever be made protected if it is absolutely necessary.

Inheritance example

To highlight the feature we have just studied, let's examine an example of inheritance. We will develop some classes for a banking system. The first is a simple bank account class, that should have the ability to add money, withdraw money and tell us the balance (how much money we have).

Example: The BankAccount class

```
1 public class BankAccount {
2     private double balance; // how much money
3     public void deposit(double amount){
4         balance = balance + amount;
5     }
6     public void withdraw(double amount){
7         balance = balance - amount;
8     }
9     public double getBalance() {
10        return balance;
11    }
12 }
```

The above example, will work for very basic representations of bank account. However, in reality there are many different types of bank accounts. Typically, most people will have a current account for their normal banking. These account provide no interest on deposited money and usually charge the user for carrying out transactions. Additionally, some customers will have a savings account, these can not be used as easily to deposit or withdraw money (you might have to visit the bank) but the bank will pay you interest on your savings.

The basic functionality of both types of accounts is the same as is provide by the bank account class. But each of the different types of account will add a little bit of functionality.

Current account

This class will add the functionality of deducting a small fee from the balance of the account.

Example: Extending the bank account class to allow transaction fees

```
1 public class CurrentAccount extends BankAccount {
2     public static final double TRANSACTION_FEE = 0.1;
3     public void deductFee() {
4         withdraw(TRANSACTION_FEE);
5     }
6 }
```

Saving account

This class extends the functionality of the bank account by adding the functionality to calculate interest and add it to the balance of the account.

Example: Extending the bank account class to allow adding interest

```
1 public class SavingAccount extends BankAccount {
2     public static final double INTEREST_RATE = 0.04;
3     public void addInterest() {
4         double interest = getBalance() * INTEREST_RATE;
5         deposit(interest);
6     }
7 }
```


We now have three different types of bank account that can be used, however we were only required to write methods such as deposit or withdraw once. This is an excellent way to reduce work when programming and make our code easier to understand and modify.

Inheritance from Object

We have just studied how to declare the relationship between subclass and superclass, but what happens when there is no superclass defined? In this case a superclass is automatically defined. Every class has one superclass. The only exception to this rule is the class `Object` included in the Java system library.

The class `Object` is the root of every inheritance hierarchy. Every class in Java automatically inherits functionality from the object class. If we do not specify a superclass when we are defining our class, then its superclass is `Object`. Every class inherits many methods from the object class, here is a list of the most commonly used;

- `boolean equals(Object obj)`
- `int hashCode()`
- `String toString()`

These methods are often used by parts of the Java system, as such there are defined in a place where they will be available to use in every class within the system. For example, `equals` is used to compare two objects which is useful for preventing duplicates in a data structure, `hashCode` is used to efficiently place objects in a data structure based on the values of the instance variables and `toString` produces a formatted string for printing to the screen or storing in log files.

Polymorphism and inheritance

We have studied in the previous chapter what polymorphism means for interfaces and have seen examples of it in use. Polymorphism also applies to inheritance. If a class A extends a class B, the objects based on both classes A and B can be stored in a variable of type B. Additionally, if the class B extends another class C, the objects based on the classes A, B or C can be stored in a variable of the type C. This continues all the way up the hierarchy as all objects based on any class can be stored in a variable of type `Object`.

Related to our banking example, an object constructed based on the `CurrentAccount` class, can be stored in a variable of the type `CurrentAccount`, `BankAccount` or `Object`. The variables that we use still limit the methods that we can use, so if the current account object is stored in an object type variable, we will only be able to use methods from the object class.

Additionally, when using `instanceof` to determine the type of a variable, it will return `true` for any of the classes above it in the inheritance hierarchy. For instance if we have an object, named `b`, based on the `CheckingAccount` class. The result of the expression `(b instanceof CheckingAccount)` is `true`. The result of the expression `(b instanceof BankAccount)` is `true` and the result of the expression `(b instanceof Object)` is also `true`.

The `instanceof` operator will return `true` any time the object matches any of the following

- The class of the object itself
- The class of the objects superclass
- The class of the superclass's superclass, etc.
- Any interface implemented by the class of the object
- Any interface implemented by any of the objects superclass's

7.2 Changing inherited functionality

When we are creating more specialised subclasses, we may want to change some of the behaviour that we have inherited from our superclass. This process is called **overriding** a method. To override a method that has been declared in the super class, we define the method again but change the functionality.

There are some requirements when overriding a method;

The **name** of the method must be the same. If the name is different we are defining a new method

The **return type** of the method must be the same. A method can only return one type of data, if it was different Java would not know what type of data will be returned. This will cause a compiler error.

The **access level** of the method must be the same or greater. If the access level was reduced, the subclass would not be able to use the same methods as the superclass. This will cause a compiler error.

The **parameters** of the method must be the same. If they are different we are defining a new method with the same name. This is called method **overloading**.

After we have defined the method with the correct details, we can alter the functionality of the method. In our example we implemented a current account, it is very common for a current account to charge a transaction fee every time money is withdrawn or deposited. We can change the functionality inherited from the bank account class and add a call to deduct fee every time that withdraw or deposit is called. (Here we are assuming that balance is not private)

Example: Current account update for transaction fees

```

1 public class CurrentAccount extends BankAccount {
2     public static final double TRANSACTION_FEE = 0.1;
3     public void deductFee() {
4         withdraw(TRANSACTION_FEE);
5     }
6     public void deposit(double amount){
7         balance = balance + amount;
8         deductFee();
9     }
10    public void withdraw(double amount){
11        balance = balance - amount;
12        deductFee();
13    }
14 }

```

However we now have a bit of a problem, we originally wrote `deductFee` to work by calling the `withdraw` method. Now every time `deposit` or `withdraw` is called, it will call `deductFee`, which will call `withdraw`, which will call `deductFee`... Our code will enter an infinite loop. What we would really want to do is to specify that we are using the `withdraw` method that is defined in the superclass. This can be achieved using the keyword **super**.

The super keyword

In a subclass, `super` refers to the direct superclass. This basically means whatever class we extend in our class declaration. There are three situations where the `super` keyword can be used;

1. To access the instance variable of the superclass (as long as the access level is correct) when we have used an instance variable with the same name in this class

2. To call a method of the superclass that have been overridden in this class
3. Inside a constructor to pass parameters to the constructor of the direct superclass

Superclass instance variables

There is no reason that you should ever have to use the `super` keyword to access or change the value of an instance variable in the superclass. It is bad programming practice to use the same name for instance variable in the subclass. This can cause much confusion when editing program (it is hard to know which is being used) and additionally the behaviour of methods may not be what you expect, depending on if a method uses the variable in the superclass or subclass. **DO NOT DO THIS!!!**

Calling superclass methods

Using `super` to call a method of the superclass is only required when that method has been overridden. If the method has not been overridden, just calling the method normally will work.

If the method has been overridden, the syntax of `super` is very similar to that of `this`.

Syntax: using `super` to call a method in the superclass

```
1 super.methodName(parameters);
```

This will call the method in the superclass and pass it any parameters that are required. The unchanged functionality of the method will then be executed. Using the `super` keyword we can change the current account class to the following, and also improve encapsulation by changing the access level of balance back to private.

Example: Current account updated to use the withdraw method from the superclass

```
1 public class CurrentAccount extends BankAccount {
2     public static final double TRANSACTION_FEE = 0.1;
3     public void deductFee() {
4         super.withdraw(TRANSACTION_FEE);
5     }
6     public void deposit(double amount){
7         super.withdraw(amount);
8         deductFee();
9     }
10    public void withdraw(double amount){
11        super.withdraw(amount);
12        deductFee();
13    }
14 }
```

Superclass constructor

As we studied in chapter 3, all objects must be constructed. Classes can be defined that do require parameters in order to be constructed. When the direct superclass of a class requires parameters in its constructor, they must be supplied by the subclass. If the subclass does not require any parameters the constructor of the superclass is called automatically for us.

Lets examine our example again, what if we added a constructor to the bank account class that required the initial balance to be passed as a parameter? The result of adding the constructor

(shown below) is that the subclasses will fail to compile unless they supply this parameter to the superclass.

Example: Constructor added to the bank account class

```
1 public BankAccount(double initial){  
2     balance = initial;  
3 }
```

Because the saving account (and the current account) extends bank account, this means that it takes what the bank account is and then adds more to it. Therefore if the bank account class requires an initial balance, then the savings account needs this too. We can pass any parameters to the constructor of the bank account (we could have the the initial balance be 0), but generally this requirement is usually passed to the creator of a saving account.

To pass parameters to the constructor of the super class all we need is the keyword super followed by the parameters in brackets. However, it is important to remember that the statement must be the first one in the constructor of the subclass.

Syntax: using super to pass constructor parameters to the superclass

```
1 public ClassName(parameters) {  
2     super(parameters);  
3     ...  
4 }
```

The parameters that are supplied must match one of the constructors in the superclass. In this example we only need to pass a single double to the bank account class, this parameter is now required by the saving account class. It must be passed to the saving account class before it is constructed.

Example: Constructor passing parameters to bank account class

```
1 public SavingAccount(double initial) {  
2     super(initial);  
3 }
```

Alternatively, if we do not want to require any parameters in the subclass, we can decide what values to pass to the constructor of the superclass. In this example, we use a constructor that requires no parameters in the saving account class, but we pass the value 0 to the bank account class as a parameter of the superclass constructor.

Example: Constructor passing default parameter to bank account class

```
1 public SavingAccount() {  
2     super(0);  
3 }
```

Method overloading

In Java it is possible for a class to contain many methods with the same name. However, each of these methods must have different parameters. When the method is used, Java chooses the correct

version of the method based on the parameters that are supplied. An excellent example of this is a class called `PrintStream`, this is the class that `System.out` belongs to.

The print stream class has many methods with the names `print` and `println`. Each method requires a different type of parameter, one will accept an `int`, one will accept a `String` and another will accept any object. Inside each method is the calculations required to print that type of parameter to the screen. This makes it much easier to use this class, as we don't really have to think very much about what we must pass as a parameter. This is why the values of variables can be printed without using any format specifier (`%d...`).

Examples

```
System.out.println(123);  
System.out.println("Hi");  
System.out.println(b);
```

7.3 Changing the Functionality Inherited from Object

Every class inherits a number of methods from the object class. These methods are often very useful, but are generally not specifically designed for the classes we have defined. A very common practice is to overload these methods so they work specifically for the instance variable and methods we have defined in our classes. The reason for this is because all objects inherit these functions, we can call them on any object.

The most useful of the methods inherited are;

- **boolean** `equals(Object obj)`
- **int** `hashCode()`
- `String toString()`

Each of the methods is already implemented, but the implementations are not very useful. As an example, let's examine the `equals` method. The `equals` method indicates if two classes are equal to each other. This means that for every instance value contained in the class, both objects should have the same value. This can be very useful in a large number of programs. The only problem is that the object class does not contain any of the instance variables of the classes that we define, therefore the implementation that is available can only compare two objects by comparing their memory addresses.

This is not very useful for comparing two separately constructed objects that may contain the same data. The big problem is that every class is different, this means that in order to effectively use the `equals` method for a class it needs to be overridden. Let's take the example of the `Bullet` class. The `Bullet` class contains many instance variables, but the ones that matter are the variables that represent location, direction and damage. In order for two bullets to be the same, they need to have the same location and direction and damage.

The name, return type, access level and parameters are all already defined in the object class, we must copy these. The method will take a single object as a parameter, which it will compare against this item object. The return type of the method is `boolean`, where the value should be `true` if both objects are the same and `false` if they are not the same.

Example: Equals method overridden for the bullet class

```

1 public boolean equals(Object o) {
2     if(o instanceof Bullet) {
3         Bullet b = (Bullet)o;
4         if(this.x == b.x && this.y == b.y && this.damage == b.damage &&
5            this.up == b.up) {
6             return true;
7         } else {
8             return false;
9         }
10    } else {
11        return false;
12    }
13 }

```

The method itself is not very complicated code, first we have to check if the object is actually a bullet object, if it is not we just return false. Once we know it is a bullet object we type cast it and compare each of the instance variables against the instance variables of the parameter. If all of the variables match we return the answer true, if they do not the answer false is returned.

Another very useful method is the `toString` method, this is supposed to return a string containing all of important details about an object so that it can be easily printed. Again, the basic implementation in the object class does not know about the instance variables in our classes. The basic version of the method only returns a string containing the name of the class and its current memory address. This is not very useful.

Again lets take the example of the bullet class from our game. Each bullet has a location and direction and power, it would be useful if we could easily display this information on the screen. As before the signature of the method is already designed so we just need to copy it. The method takes no parameters and return a String.

Example: ToString method overridden for bullet class

```

1 public String toString() {
2     String s = "Bullet located at (" + x + ", " + y + ") the bullet is
3         travelling ";
4     if (up) {
5         s += "up";
6     } else {
7         s += "down";
8     }
9     s += " and has power " + damage;
10    return s;
11 }

```

Here we use concatenation to create a string containing all of the information about this object formatted well for printing. This can be particularly useful for outputting information to the screen. This class originally had a method called `printBullet` which served a similar purpose, the important distinction is that when this method is called, nothing is printed to the screen. A string is returned as a result of the method and if we wish to print it we must do so ourselves.

Example: Making use of the toString method in testing

```

1 Bullet b = g.enemyShips[i].fire();
2 if(b != null) {
3     System.out.println(pB.toString());
4 }

```

This greatly simplifies the code that we wrote earlier. We no longer have to print each instance variable one at a time, we just print the entire string describing that object.

7.4 Inheritance in Interfaces

Just as there is inheritance between class, there is also inheritance between interfaces. Inheritance in interfaces is much simpler than in classes, because interfaces cannot contain implemented method. As such when one interface inherits from another, it is simply adding further requirements for any class that implements the subinterface.

Example: A and B interface definitions

<pre> 1 public interface A { 2 void methodA(); 3 } </pre>	<pre> 1 public interface B extends A { 2 void methodB(); 3 } </pre>
---	---

In this example, any class that implements interface A is required to implement methodA. However, any class that implements interface B is required to implement both methodA and methodB.

7.5 Abstract Classes

The term abstract in English means something that is an idea or concept and does not exist in the real world. For example, think of the concept of a phone, this is an idea that exists on thought. However, based on this idea many devices have been constructed which are all different implementations of the abstract phone concept.

Abstract when applied to object-oriented programming means something that is not implemented. The abstract keyword can be applied to classes and methods. In fact we have already seen examples of abstract method when we were studying interfaces.

An abstract class is similar to an interface in some ways. For example, an object cannot be created based on an abstract class. Attempting to create the object will result in a compiler error, just as attempting to create an object based on an interface. Additionally, an abstract class is of no use unless it is implemented by a subclass just as an interface is of no use unless it is implemented.

The big difference, between abstract classes and interfaces is that an abstract class can contain instance variables and completed (non-abstract) methods. However, an abstract class can also contain abstract methods, these are the same as the methods listed in an interface. An abstract method does not require an implementation and any method without an implementation must be declared as abstract. A method is declared abstract by placing the keyword abstract before the return type of the method.

Syntax: Declaring an abstract method

```

1 public abstract void methodName(parameters);

```

Additionally, any class can be declared as abstract, even if it contains no abstract methods. However, if a class contains a single abstract method, then the class must be declared as abstract.

Syntax: Declaring an abstract class

```
1 public abstract class ClassName {  
2     ...  
3 }
```

An abstract class is similar to the idea of an interface in that it imposes a requirement on its subclasses. Any class that extends an abstract class must either provide an implemented version of the abstract methods or also be declared as an abstract class. As a general programming convention, abstract classes are usually named with the word `Abstract` to the beginning of the name of the class. This makes it much easier to identify abstract classes that you may be working with.

Abstract class or interface

When designing a system, it can sometimes be difficult to decide if it is better to use an abstract class or an interface. Generally, if the subclasses will share not only the same methods, but also the same code and instance variables for those methods, it is better to use an abstract class. However, as each class can only have one superclass, in situations where we want to be associated with several different types, using interfaces can be more useful as we can implement many interfaces.

Abstract class usage example

This example is going to introduce the idea of using abstract classes in a graphics environment. The topic of drawing user interfaces will be covered in section ?? in much greater detail. This example is just to show a situation where they are useful.

Lets assume that we are drawing some two dimensional shapes on the screen. All of the shapes will have some common functionality and code, so we can declare an abstract class to represent this common code, and add an abstract method to show where the different code will be implemented. The `Abstract2DDrawing` class below contains instance variables, the required constructor and a move method which will be common to all drawings.

Example: The abstract drawing class

```
1 public abstract class Abstract2DDrawing {  
2     protected int x, y;  
3     public Abstract2DDrawing(int x1, int y1){  
4         x = x1;  
5         y = y1;  
6     }  
7     public void move(int x1, int y1){  
8         x += x1;  
9         y += y1;  
10    }  
11    public abstract void draw(Graphics g);  
12 }
```

Additionally, the class declares an abstract method named `draw`. This method requires a single parameter of the type `Graphics` and will return no value. The parameter will be passed from a system library when the method is called (this will be covered later) and the idea of the method is that this is where the instructions on how to draw the shape will be executed.

When a subclass of the abstract drawing class is defined, there are two requirements. First, the required parameters must be passed to the superclass constructor. Secondly, the `draw` method must be implemented. In this example, we are creating a `Building` class that extends the abstract drawing class. This class will draw the shape of a building on our screen (just a simple rectangle).



Figure 7.2: Image displayed by the example

The actual methods for drawing on the screen are contained in the Graphics class, but in this example we are only using a single method `fillRect`. This will fill a rectangle defined by the 4 parameters with whatever color the system is drawing in.

Example: The building class

```
1 public class Building extends Abstract2DDrawing{
2     private int w, h;
3     public Building(int x1, int y1, int w1, int h1) {
4         super(x1, y1);
5         w = w1;
6         h = h1;
7     }
8     public void draw(Graphics g) {
9         g.fillRect(x, y, w, h);
10    }
11 }
```

Lastly, in another section of code we can actually use the building class to draw some images to a window on the screen. The code for setting this window up and getting access to a graphics object are complicated, so we will only show the objects being created and the parameters passed.

Example: Using the building class

```
1 Building b = new Building(70, 50, 55, 350);
2 b.draw(g);
3 Building c = new Building(160, 50, 55, 300);
4 c.draw(g);
```

Here two objects are constructed based on the same building class. Because different parameters have been passed to the two objects, they are created in different locations on the screen.

7.6 Final Classes and Methods

In chapter 5, we saw the use of the keyword `final` to define variables that could not be changed. The keyword `final` is also used in inheritance. It can be used to describe both classes and methods.

Final classes

A final class is one that cannot be extended. This is used when we create a class for others to use, but we do not want them to be able to change the functionality. This means that they cannot create a subclass and alter the functionality, they can only use the class as we have designed it.

A typical example of a final class would be a class representing a bank account. Once we have written the code that defines how money is deposited or interest is applied, we do not want the user of the class to be able to change how we have implemented this. Without the class being declared final, they could simply extend the class and override a method designed to apply a fee to the account, they could then replace this with code that adds money to the account instead.

Syntax: Declaring a final class

```
1 public final class BankAccount{  
2     ...  
3 }
```

Once the final keyword is used in this way, we define code exactly the same as before, however we know that this code can never be overridden.

Final methods

If we want to allow the programmers using our code to create subclasses, but we still want to prevent them from changing some of the method, we can define these methods as final. This means that when the class is extended, the final methods cannot be overridden. A method can be declared final by placing the keyword final before the return type of the method.

Example: A final method declared for password checking.

```
1 public final boolean checkPassword(String p){  
2     ...  
3 }
```

This example show the declaration of a method that should not be overridden in a subclass. If this was allowed to be overridden, the programmer could simply replace it with one that always returns true. This makes our objects more secure.

Calling methods from the constructor

In Java, it is perfectly acceptable to call other methods from any code within a class, this is also true for the constructor of a class. However, any method that you call from a constructor should be declared as final. This is because when the class is extended, the functionality of this classes might be overridden. We will not know what effect this might have on the constructor. Declaring the methods as final will prevent this problem.

7.7 Immutable Objects

An object is considered **immutable** (unchangeable) if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a good strategy for creating simple,

reliable code. Immutable objects are particularly useful in multi-threaded applications, which we will look at in chapter 14. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

Lets look at an example of an immutable object that we have been using already. The `String` class is immutable. Once a string is created, it cannot be modified. This is why many of the methods in the string class that we have used return a `String`. Think of examples such as `replace`, this method finds words or characters within a string object and replaces them. However, because string objects are immutable the actual object is never changed, instead a copy of the string is created with the changes made.

So what makes an object immutable?

1. All instance variables (and class variables) are declared as `final` (and usually `private`)
2. Setter methods are not provided, or any other method that changes the value of an instance variable.
3. Subclasses are not allowed to override any of the methods (declared as `final`), usually this is done by declaring the class as `final`.
4. Instance variables that refer to objects cannot be changed.

The first point is easy, our variables are declared as `final`. This means that the compiler will not allow the value to be changed once it has been assigned a value. Initial values must be provided either in the declaration or in the constructor.

Point 2 might be considered unnecessary, however consider the example of an array as an instance variable. If the array is declared as `final`, this means that the reference cannot be changed, but the values stored in the array may be modified. This means that for an object to be immutable, no method must be allowed to change the value of an array or other mutable object in the class.

Point 3 means that when extended, the subclass cannot change how any of the methods operate, this means that the code we have implemented cannot be altered. However, unless the entire class is declared as `final`, it is possible to add instance variables that are changed by new methods. This means that a subclass of an immutable object can be mutable.

Point 4 refers to instance variables inside the object. Consider the example of a immutable student object that contains an array of `Grade` objects. If these grade object are mutable, then we must prevent programmers from changing these objects. This means that if there is a `getGrade` method, instead of returning a grade object we store, we should return a new `Grade` object containing the same information. If we simply return the grade object, the user may change the value of the grade or modify the object in some other way and our code will still refer to the same object with a new value.

Example: Using immutable string objects

```
1 String hel = "Hello world";
2 String gdb = hel.replace("Hello", "Goodbye");
3 System.out.println(hel);
4 System.out.println(gdb);
```

Here in this example, we see that we first create a `String` named `hel` and assign it a value. When the `replace` method is called, the returned value is stored in another string variable named `gdb`. When both strings are printed, we see that the value of `hel` is not changed by the `replace` method, and the new string was created and stored in the variable `gdb`.

Manipulating strings can be very common, if we are creating many new objects in this process it can be wasteful. There is a mutable version of a string called the `StringBuilder`. This object allows the sequence of characters store inside to be altered.

Example: Using the mutable string builder object

```
1 StringBuilder sb = new StringBuilder("hello world");
2 sb.replace(0, 5, "GoodBye");
3 System.out.println(sb);
```

In this example, the `replace` method is used and directly changes the string builder object, instead of creating a new copy. This can be useful in cases where we are performing a lot of modifications to a string. Consider an example where we are creating a string showing the contents of an array. Every time we add a new piece of information, a new `String` is created.

Example: Building a string with the contents of an array

```
1 int[] numbers = new int[100000];
2 for (int i = 0; i < numbers.length; i++) {
3     numberString = numberString + numbers[i] + " ";
4 }
5 numberString = numberString + "]" ;
```

In this example, first we create a new `String` with the contents "[", then in each step in the loop we perform the following operations

- create a new string object concatenating the original value with the next number
- create a new string object concatenating the new string with the value " "

Finally, we create one more copy adding the final character "]". For the array in the example, we have copied all of the contents of the string 200,001 times before we have completed the task. If instead we replace the string object with a string builder object, this would be much more efficient.

Example: Building a string using a string builder with the contents of an array

```
1 int[] numbers = new int[100000];
2 StringBuilder numberBuilder = new StringBuilder("[ ");
3 for (int i = 0; i < numbers.length; i++) {
4     numberBuilder.append(numbers[i]);
5     numberBuilder.append(" ");
6 }
7 numberBuilder.append("]") ;
```

In this example, no extra objects are created and therefore we never need to copy the contents of the string. Comparing the runtime of these two pieces of code for an array of 100000 random integers gives very different results. On my computer, the string version of the code took over 27 seconds to complete and the String builder took only 8 milliseconds. That is 3438 times faster!

7.8 Inheritance Example

This section applies some of the material covered in this chapter to the game example that is shown through the book. In the last chapter, we identified common code spread across the different classes. This common code was then captured as an interface which the classes implemented. This will allow different types of objects be moved together or drawn on the screen together.

After studying inheritance, we can look at our code and see where there are some classes with code in common. So far we have been developing the ship class so that it is useful for both enemy ships and the players ship. While there are a lot of actions common between these two, they are

not exactly the same. Instead, we will split these into two classes, **Player** and **Enemy**. However to save time we will have both of these classes inherit common functionality from the **Ship** superclass.

The primary differences are between the two classes are;

- When firing, bullets will appear from different locations (above player, below enemy)
- The number of points returned as a result of the ship being destroyed are different
- Enemy ships are of different types, represented by an enum
- Player and enemy ships are moved in different ways

This means that the methods **fire**, **getPoints**, and **move** should be implemented in the subclasses **Player** and **Enemy**, but not in the superclass **Ship**. For this reason we will make those methods and the ship class abstract and implement the methods in the subclasses.

Example: Reformatted Ship class

```

1 public abstract class Ship implements Hittable, Movable {
2     protected int x, y;
3     private int health, cooldown;
4     public static final int SHIP_WIDTH = 10;
5     private Rectangle box;
6
7     public Ship(int x1, int y1, int c, int h) { ... }
8
9     public abstract Bullet fire();
10
11    public abstract void move();
12
13    public abstract int getPoints();
14
15    public boolean isHit(Bullet b) { ... }
16    public void takeDamage(Bullet bullet) { ... }
17    public boolean isDestroyed() { ... }
18    public int getCooldown() { ... }
19 }

```

This example shows the **Ship** class after having been changed to serve as a super class to **Player** and **Enemy**. The important points, the class is abstract so it can contain abstract methods like **fire** and **move**. Some of the variables are defined as protected so that they can be used in the subclass. Other methods such as **isHit** and **takeDamage** are fully implemented and as such do not need to be defined in the **Player** or **Enemy** classes.

Example: Partially completed Enemy class

```

1 public class Enemy extends Ship {
2     private AlienType type;
3
4     public Enemy(int x1, int y1, int c, int h, AlienType a) {
5         super(x1, y1, c, h);
6         type = a;
7     }
8
9     public Bullet fire() {
10        Bullet b = null;
11        if (getCooldown() == 0) {
12            // TODO calculate correct location for bullet
13            b = new Bullet(x, y, true, 5);
14        }
15        return b;
16    }
17
18    public int getPoints() {
19        return type.getScore();
20    }
21
22    public void move() {
23        // TODO calculate how to move ship
24    }
25 }

```

This example shows how the Enemy class extends the Ship class and completes some of the required methods. Here an extra parameter is required to determine which type of ship it is. This parameter is then used to determine how many points the enemy ship is worth when destroyed.

7.9 Practical Exercises

1. Define a class named **Vehicle**. This class will be used to define all of the methods that are common to all types of vehicles. The class should have a constructor, which takes the name, top speed and maximum number of passengers as parameters. These values should be stored in private instance variables within the class. In addition the class should have the following methods;
 - **getName** - returns the name of the vehicle
 - **getTopSpeed** - returns the top speed of the vehicle
 - **getMaxPassengers** - returns the maximum number of passengers the vehicle can carry
2. Define a class named **Car**. This class should extend the **Vehicle** class from the previous question. The class should have a constructor, which takes the name, top speed, maximum number of passengers and weight (double) as parameters. The first three parameters should be passed to the constructor of the superclass and the value of the last parameter should be stored in an instance variable. The class should define a single method, **getWeight** which returns the weight of the car as a double value in tones.
3. Define a class named **Ship**. This class should extend the **Vehicle** class from the first question. The class should have a constructor, which takes the name, top speed, maximum number of passengers and displacement (double) as parameters. The first three parameters should be passed to the constructor of the superclass and the value of the last parameter should be

stored in an instance variable. The class should define a single method, `getDisplacement` which returns the amount of water that the ship displaces as a double value in tones.

4. Create a class called `VehicleTest`. This class should have a main method. This class should contain the following statements;
 - Declare an array of `Vehicle` objects named `vehicles`
 - Construct an array of `Vehicle` of size 5 and store it in the variable `vehicles`
 - Construct and insert the following five vehicles into the array
 - Car: Name: “Lamborghini Gallardo”, Passengers: 2, Top Speed: 325 km/h, weight 1.485 tones
 - Ship: Name: “Titanic”, Passengers: 3327, tTp Speed: 39 km/h, Displacement: 52310 tones
 - Car: Name: “Toyota Avensis”, Passengers: 5, Top Speed: 200 km/h, Weight: 2.02 tones
 - Ship: Name: “Nimitz”, Passangers: 5680, Speed: 58.3, Displacement: 100020 tones
 - Car: Name: “Mazda 3”, Passengers: 5, Top Speed: 190 Km/h, Weight: 1.3 tones
 - Add a class method named `heaviestVehicle` to the class. This method should take as a parameter an array of `Vehicle` objects and return the name of the heaviest one as a `String`. (Displacement is the same as weight)
 - Declare a `String` named `heaviest`
 - Call the method `heaviestVehicle`, passing the array `vehicles` as a parameter. The result should be stored in the variable `heaviest`.
 - Print the value of the variable `heaviest`.
 - Write a loop to iterate over all of the elements in the array in reverse
 - Inside this loop you should use the `instanceof` keyword to check if the object in the array belongs to the `Car`, `Vehicle` or `Ship` class
 - For each if statement there should be a print statement containing “Object X is a Vehicle”, “Object X is a Car” or “Object X is a Ship” as appropriate. X should be replaced by the name of the vehicle
5. Create a class named `Person`, this class will provide functionality that is common to all of people. The class should have a constructor, which takes the name and age as parameters. These values should be stored in private instance variables within the class. In addition the class should have the following methods;
 - `getName` - returns the name of the person
 - `getAge` - returns the age of the person
6. Write a class named `Employee`, this class should extend the class `Person`. In order to extend this class `Employee` must contain a constructor that passes the required parameters to the superclass. Additionally, the class should contain instance variables for the employees salary and employer, which are also passed as a parameters to the constructor. The following methods should also be added;
 - `getSalary` - This should return the salary of the employee as a double
 - `getEmployer` - This method should return the name of the company that employs this person
7. Write a class named `PersonTest`, this class should contain a main method with the following statements;
 - Declare and construct an array of `Person` objects named `people`, this array should have the size 3.

- Construct and insert the following three objects
 - A Person named John who is age 24
 - An Employee named Paul who is 29 and works for a company named Microsoft and earns a salary of 40000
 - An Employee named Jones who is 21 and works for a company named Google and earns a salary of 30000.
 - Add a class method named `oldestPerson` to the class. This method should take as a parameter an array of Person objects and return the name of the oldest person as a String.
 - Declare a String named `oldest`.
 - Call the method `oldestPerson`, passing the array `people` as a parameter. The result should be stored in the variable `oldest` and then printed to the screen.
 - Add a loop to the main method to print out each element of the array by calling the `toString` method of the object.
8. Override the method `toString` from `Object` in the `Person` class. This method should return a String containing a description of the Person, including their name and age. Test the code by running the main method in the `PersonTest` class.
 9. Override the method `toString` from `Object` in the `Employee` class. This method should reuse the `toString` method from the `Person` class and add details of their employer and salary. Test the code by running the main method in the `PersonTest` class.

7.10 Game Completion Tasks

1. Add the method `update` to the `Ship` class. This method will be called once for every step in the game. The only action that should be performed is to reduce cooldown by one until it reaches 0.
2. Modify the `fire` method in the `Ship` and `Enemy` classes. After the weapon has fired, the `cooldown` should be set to 10.
3. Add an array of `Movable` objects to the `Game` class, as well as an integer for counting how many objects are in the array.
4. Add an array of `Hittable` objects to the `Game` class, as well as an integer for counting how many objects are in the array.
5. Add an array of `Bullet` objects to the `Game` class, as well as an integer for counting how many objects are in the array.
6. In the constructor of the `Game` class, every object created that implements hittable should be added to the hittable array and every object that implements movable should be added to the movable array.
7. Add the method `step` to the `Game` class. This method should take no parameters and return no value.
8. The `step` method will represent all of the calculations for updating parts of the game. In the `step` method the following things should happen:
 - The player should be updated (call the `update` method)
 - Each of the Enemy ships should be updated
 - The `move` method should be called for every object in the movable array

- The **fire** method should be called for the player object. This assumes that key checking will be added later. The returned bullet should be saved and checked in case it is null. If the bullet is not null, it should be added to the bullet array, the movable array and the hittable array.
- This process should be repeated for each of the enemy ships
- For every bullet in the array, we will check if it has hit every object in the hittable array. When a hittable object has been hit its **takeDamage** method should be called and if destroyed, the number of points it returns should be added to the players score.

Chapter 8

Nested Classes

It is possible to declare multiple classes (interfaces or enums too) in the same file. These can be defined one after another in the source code. However, we know that if a class is public, then the name of the class must match the name of the file. This means that any other classes that we declare in this way cannot be declared as public.

Example: The contents of the file FirstClass.java

```
1 public class FirstClass{
2     ...
3 }
4 class SecondClass{
5     ...
6 }
```

This shows a situation where we have declared two classes in the same file. Because **FirstClass** is declared as public, the name of the file must be FirstClass.java. Any other classes can only be declared as **private**, **protected** or **default**. This is why **SecondClass** is declared as default. **SecondClass** can be used as normal from anywhere inside the same package, just like anything else declared as default.

This can sometimes be useful for placing a lot of code together, but it makes understanding the code more difficult. Normally, if I want to find code for a class, I only need to go to the correct file in the correct package, however in this case I will have to search multiple files until I find the one containing **SecondClass**. This is not a recommended way of structuring your code.

A **nested** class is when one class is defined **inside** another class. It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "**helper classes**" makes their package more streamlined. It increases encapsulation: Consider two top-level classes, A and B, where B needs access to instance variables of A that would otherwise be declared private. By hiding class B within class A, A's instance variables can be declared private and B can access them. In addition, B itself can be hidden from the outside world. It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called static nested classes. Non-static nested classes are called **inner** classes.

8.1 Inner Classes

An inner class because it is not static, must belong to an instance of the outer class. This means that if we want to create an instance of an inner class, we first need to have a reference to an object of the outer class.

An inner class is used to define something that will only be used in a single place. Consider the Node classes in most linked data structures, they are used to represent the relative position of the information inside the data structure, but no other classes need to know about the actual class. When we use a link based stack, we do not need any information about the Node class because we only ever interact with the object we insert and remove, never a Node object.

Example: Link based stack implementation using inner node class

```
1 public class LinkStack implements Stack {  
2  
3     private class Node {  
4         private Object element;  
5         private Node next;  
6  
7         public Node(Object e) {  
8             element = e;  
9         }  
10    }  
11  
12    private Node top;  
13    private int size;
```

In this example, we have declared a private inner class named `Node`. Because the class is private, it can only be used from within the same file. This means that the `LinkStack` class can know what a `Node` object is but to any other class it does not exist. This is excellent because now our node class cannot be confused with similar classes from another link based data structure.

Additionally, we know that the stack interface stores and returns objects. To any other class using a link based stack object, they do not need to know how it is represented inside the class.

Accessing data from the outer class

Because an inner class is non-static it can access the data of the class it is declared inside, even if the data is private. To give an example of this, we are going to create an `Iterator` class for our link based stack implementation.

Iterators

An iterator is an object that can be used to access all of the elements in a data structure one by one. Iterators are defined by the interface `Iterator` in the package `java.util`. When we define an iterator, we need to implement two methods;

- **hasNext** - returns a boolean value to tell us if there are any more items we have not accessed
- **next** - returns the next item in the data structure that we have not already accessed

If we have an iterable data structure containing three items, we can ask for an iterator to access these items. The first time we call **next**, it will give us the first item, the second time we call **next** it will give us the second item and so on.

If we are designing an iterator for a link based implementation of the stack, we will need to know which `Node` we should return the data from. Whenever we are asked for an item, we return the item and change the location where the next item will be taken from. This means for the link based stack we start at the `Node top` and then the next node and the next node. This means we need to remember what the current node we are looking at is.

Example: Iterator implemented as inner class

```
1 public class LinkStack implements Stack, Iterable {
2     private Node top;
3     private int size;
4     private class StackIterator implements Iterator{
5         private Node current;
6         private StackIterator() {
7             current = top;
8         }
9         public boolean hasNext() {
10             return current != null;
11         }
12         public Object next() {
13             Object o = current.element;
14             current = current.next;
15             return o;
16         }
17     }
18     ...
}
```

The definition of the methods `next` and `hasNext` both make use of the instance variable `current`. The most important line in this example, is line 7 where the reference to the node on top of the stack is copied into the variable `current`. What is important is that because this is an inner class, the reference does not need to be passed as a parameter, the stack iterator already has access to `top`.

Public inner classes

Inner classes can be declared with any visibility, e.g. `public`, `protected`, default or `private`. Normally inner classes will be declared as `private` if they will not be used by other classes. Public inner classes can be constructed by any class to a reference to an outer class. For example, if I have a reference to a `LinkStack` object and the `StackIterator` class was public, then I could create an instance of the `StackIterator` class.

Example: Creating an instance of an inner class

```
1 LinkStack stack = new LinkStack();
2 LinkStack.StackIterator it = stack.new StackIterator();
```

Note how to create the stack iterator, we need to use a reference to a link stack object, e.g. `stack.new StackIterator()`; rather than simply `new StackIterator()` if it had not been an inner class. This is not a very common operation.

8.2 Anonymous inner classes

In the previous example, we created a public inner class to allow users to create an iterator for our link based stack. This is not a very good way to define the class, it requires a lot of knowledge for the users of our class. Instead we should create a method that they can call to get an iterator for the data structure. The interface `Iterable` in the package `java.util`, defines a data structure that provides an iterator. It requires a single method named `iterator`, which returns an iterator object.

If this is the only way that the users of our data structure can access our iterator, then we do not need to declare it as public, in fact we do not even need to give the class a name. We can instead define an anonymous inner class directly in the iterator method.

An anonymous inner class is generally created as an implementation of an interface or abstract class. However, we know that we cannot create an object based on these types. So an anonymous inner class provides the implementation of the class or interface when it is being created.

Example: Iterator defined as anonymous inner class

```
1 public Iterator iterator() {  
2     Iterator it = new Iterator() {  
3         Node current = top;  
4         public boolean hasNext() { ... }  
5         public Object next() { ... }  
6     };  
7     return it;  
8 }
```

Here we can see that instead of a semicolon after we call the constructor for the `Iterator` interface, we define the class inside a set of brackets. The semicolon is in fact placed after the class has been defined on line 7. The object is then returned by the method so the all the user is required to do to access an iterator is to call this method.

8.3 Static Nested Classes

Static nested classes are slightly different from inner classes. A static nested class cannot access the non-static instance variables of the outer class. Static nested classes can be constructed without an instance of the outer class, think of them like static methods or variables.

An example of a static nested class is `Double`, which is nested in the abstract class `Line2D` in the package `java.awt.geom`. The `Double` class is actually a subclass of the `Line2D` class, it provides an implementation of the abstract class where all of the information is represented as doubles. There is another nested class called `Float`, which does the same but represented as floats. These classes are useful for performing some calculations related to lines and line segments, which can be useful in the development of 2D applications or games.

To declare an instance of a static nested class we use the name of the outer class and the nested class. For example,

```
OuterClass.StaticNestClass nestObj = new OuterClass.StaticNestClass(); or  
Line2D.Double l = new Line2D.Double(0,0,10,10);
```

This object can then be used as normal, the same as any other object that we construct.

Further Types

In Java, there are also related topics such as Lambda expressions and local inner classes. However, because this book is focusing on the key concepts in OOP, not just in Java, these topics are left to you to read about further.

8.4 Game Example

In the game we are creating there is one example of a class that is used only in a single place, the `Brick` class. We only use the brick class inside the `Bunker` class, there is no reason this class needs to be accessible by any others. We can safely make this an inner class inside the bunker class.

Example: The bunker class with the brick class as a private inner class

```
1 public class Bunker implements Drawable, Hittable {
2     private int x, y, noBricks;
3     private Brick[] bricks;
4
5     public Bunker(int x, int y) { ... }
6
7     private class Brick implements Hittable {
8         ...
9         public Brick(int x1, int y1) { ... }
10        public Rectangle getShape() { ... }
11        public boolean isHit(Bullet r) { ... }
12        public boolean isDestroyed() { ... }
13        public int getPoints() { ... }
14    }
15    ...
16 }
```


Chapter 9

Text Input and String Processing

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the concept of input and output in programming
- ☐ be able to use the Scanner to read input from the user
- ☐ be able to give the user many attempts to enter the correct input
- ☐ be able to find and extract information from a string

In this chapter and the next we will look at different ways of interacting with our programs. The most basic form of interaction is achieved through text input and output. We have already used the methods `print` and `println` to output information and messages to the screen, this will be complemented with using an object called a scanner to read text input from the user.

9.1 Text Input

Reading text input from the user is a little different in Java than it is in C. Instead of just calling a function to read values, we have to use an object to perform these steps for us. There are many different ways to do this provided by different classes in the Java libraries. We will be using a class called **Scanner** from the package `java.util`. The scanner class can be used to create many scanner objects that are useful for breaking up text and converting it into useful formats for us. The Scanner is designed to be reusable in many different situations, this means that we have to specify **how** it is to be used when we create it. Additionally we will also have to import the class any time we want to use it.

As with all objects, before we can use it, it must be constructed. There are 10 different constructors for the scanner class, these can be used to read information from different sources. There are two that are of particular interest to us, but for now we will focus on just one that can be used to read information from the command prompt.

The constructor we will use is described below. This is an excerpt taken from the API documentation for the scanner class. The constructor requires a single parameter, that is of the type **InputStream**. But we do not know what an input stream is or how to create it. We will learn more about what an input stream is and how to use them later when we study file IO. For now we will always use the same parameter that is created for us but the JVM. In the same way we use the **PrintStream** created for us called `System.out`, we will use an input stream created for us called `System.in`.

API: Scanner - Constructor

public Scanner(InputStream source)

Constructs a new Scanner that produces values scanned from the specified input stream. Bytes from the stream are converted into characters using the underlying platform's default charset.

Parameters: source - An input stream to be scanned

This means to create a scanner we just need to define a variable to store the object and pass it the correct parameter.

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

This defines a variable of the type scanner named input (we can use any name) and stores the scanner object created to read information from `System.in`.

Using a scanner

Reading information using a scanner is different than using `scanf`. When using a scanner we can only read information one piece at a time and in the order that it was entered. The information that we read can vary from a single character up to an entire line of text. The scanner class contains many methods that can be used to read information and each method reads a specific type of data.

Reading order

A scanner will always read a group of characters together. These groups are known as **tokens**. Tokens are separated by whitespace characters such as new line, space or tab characters. The characters used to separate the tokens are called delimiters and can be changed, but we will not need to do this in our examples. Because the scanner reads tokens in order from first to last, all of the methods for reading information are named **nextXXX**. This refers to the next token to be read, for example the method `nextInt` will try and read the next token as an int and `nextDouble` tries to read the next token as a double.

Each of the methods will read a token and often try to interpret it in some way, this may cause an error if we are not careful about what we type. For example if we try to read a double using the method `nextDouble`, but the user types the string "abc", then the scanner will attempt to convert "abc" into a double and fail because it is not. This will cause our programs to stop working unless we are very careful.

Here are some of the methods we can use to read information, we will look at some of these in more detail later in this section.

Method	Return type	Explanation
<code>nextInt</code>	int	Reads the next token as an int
<code>nextDouble</code>	double	Reads the next token as a double
<code>next</code>	String	Reads the next token as a String
<code>nextLine</code>	String	Reads all of the tokens remaining on this line as a String

Lets say for example we have a number of lines of data containing the following information about a number of students; an integer representing their grade in an assignment, some characters representing their letter grade in an exam, a real number representing their grade in a programming exam finally their name. All of this information is contained on a single line, what operations and in what order are required to read this information into our program in a way that is usable.

First we need to read the assignment grade token, to do this we call the `nextInt` method. Next we need to read the letter grade token, this is done by calling the `next` method. We also need to read in the programming exam token, this is done by calling the `nextDouble` method. Finally we need to read in the students name, as a name will most likely be multiple tokens we will call the `nextLine` method to read in all of the tokens remaining on the line.

API: Scanner - nextInt**public int** nextInt()

Scans the next token of the input as an int.

Returns: the int scanned from the input

Lets look at an example of using the nextInt method to read an integer from the user.

Example: Reading an int using nextInt

```
1 import java.util.Scanner;
2
3 public class ScannerExamples {
4     public static void main(String[] args) {
5         Scanner in = new Scanner(System.in);
6
7         System.out.println("Enter an int: ");
8
9         int x = in.nextInt();
10
11        System.out.println("You entered " + x);
12    }
13 }
```

Here we can see that there are a number of steps;

1. On line 1 the scanner class is imported so we can use it
2. On line 5 the scanner named in is constructed
3. On line 7 a message is printed telling the user what to do
4. On line 9 we declare a variable named x and use it to store the result of calling the method nextInt on the object in
5. On line 11 we print out the number

The code above can be easily changed to read other values from the user. The scanner itself only needs to be created a single time, but it can be used many times to read information from the user. For example, the code to read the student information described above would be:

Example: Code to read the student information

```
1 Scanner in = new Scanner(System.in);
2 System.out.println("Enter the students information");
3 int assignmentGrade = in.nextInt();
4 String examGrade = in.next();
5 double pExamResult = in.nextDouble();
6 String name = in.nextLine();
7 System.out.println(name + " got results " + examGrade + ", " +
    pExamResult + " and " + pExamResult);
```

API: Scanner - nextDouble**public double** nextDouble()

Scans the next token of the input as a double.

Returns: the double scanned from the input

API: Scanner - next**public** String next()

Finds and returns the next complete token from this scanner. This method may block while waiting for input to scan.

Returns: the next token

API: Scanner - nextLine**public** String nextLine()

Advances this scanner past the current line and returns the input that was skipped. This method returns the rest of the current line, excluding any line separator at the end. The position is set to the beginning of the next line.

Returns: the line that was skipped

User Prompts

In the API excerpt for the next method, it notes that the method may **block** while waiting for input to scan. This means that the scanner will wait until the user has typed something. When this is happening, it can often be confused for the program freezing or entering an infinite loop. It is always important that when asking the user for input that there is a message to tell them what input is expected.

Usually, before we call a method on a scanner object, we would first print a message to the user telling they are expected to enter. These messages should be relatively specific, so that the user has a good idea of the type of data that is to be entered. This will help prevent problems where the scanner is expecting one type of information and the user enters another.

9.2 Input Checking

When we write programs for user input, that input is expected in a specific order. If we are expecting an integer, but anything else is entered, this will cause a mismatch error and probably crash our program. This can only happen if we are trying to read a specific type of data, such as an int, long or double. There can never be a mismatch error when we are trying to read a string. This is because, the rules allow any type of character to be stored in a String.

Example: Output when incorrect type of data is entered

```
1 Enter an int:
2 abc
3 Exception in thread "main" java.util.InputMismatchException
4   at java.util.Scanner.throwFor(Unknown Source)
5   at java.util.Scanner.next(Unknown Source)
6   at java.util.Scanner.nextInt(Unknown Source)
7   at java.util.Scanner.nextInt(Unknown Source)
8   at ScannerExamples.main(ScannerExamples.java:9)
```

This is an example of the output you will see when a mismatch error happens. The meaning of this output will be discussed in much more detail in chapter 12. For now we only need to be able to recognise the problem when we see it.

There are two possible solutions to these types of problems, we can write our programs in a more robust way so that they can recover when this happens or we can check the type of the next token before we read it. The first solutions we will look at when we reach chapter 12.

What comes next?

These problems with user input can be solved if we have our program check user input before we process it. The scanner allows us to do this using a number of methods that check the type of the next token. We can call these methods before we read a piece of information. For example, if we are going to read a double, we first ask the scanner if the next token is a double, if it is everything is okay and we read the double. If the next token is not a double, then we can read the token as a string and ask the user to enter the information again.

The methods that we use are all basically the same, if we are going to call the `nextInt` method then first we check by calling the `hasNextInt` method. These methods all take no parameters and return a boolean value as a result. If the result is true, then the type is correct and if it is false then the type is not correct.

API: Scanner - hasNextInt

public boolean hasNextInt()

Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the `nextInt()` method. The scanner does not advance past any input.

Returns: true if and only if the scanner's next token is a valid int value

As the methods all return a boolean value, we can easily use them within an if statement and decide what to do. It is often useful to tell the user that they did not enter the correct type and also tell them what they entered.

Example: Checking that the input is an int

```

1 Scanner in = new Scanner(System.in);
2 System.out.println("Enter an int: ");
3
4 if (in.hasNextInt()) {
5     int x = in.nextInt();
6     System.out.println("You entered " + x);
7 } else {
8     String m = in.next();
9     System.out.println("Not an int, you entered <" + m + ">");
10 }

```

More attempts

This type of check is useful, but there is a big problem with it. If the user was supposed to enter an integer but did not then we still need to know the value of the integer. Ideally we should give the user another chance to enter the value and possibly many chances until they get it correct. If we are going to give the user another attempt, we need to remove the token that is not correct, this is done by calling the method `next` (line 8 in the example).

Because the user may make many mistakes when entering input, we will want to give them many attempts to get it correct. The easiest way to do this is using a loop, but how do we know when the loop should execute. The easiest solution is to use a loop that executes when the user is incorrect, this way if the user is correct the loop will never execute and we can simply read the value.

For example, if we ask the user to enter an int we would add a loop that removes a token if it is **not** an integer. If we are using a scanner named `scan` the code `scan.hasNextInt()` will return true if the token is an integer, but we want a loop that will execute when the token is not an integer. We can apply the boolean operator not to the result of the code and it will reverse the value. This means we use `!scan.hasNextInt()` instead (! is the not operator).

What this means is that when we want to read an integer value, the loop will execute as long as the token is not an integer value. Inside the loop then we need to remove the token from the scanner by calling the `next` method. It is often helpful to also print a message showing the user what they entered. The only way for the loop to end is if the user enters an integer value.

Once the loop is completed, we can read the value we want. Because of the position of the loop, just before, we know that the token we are about to read must be an integer.

Example: Multiple attempts to enter an integer

```
1 public static void main(String[] args) {  
2     Scanner in = new Scanner(System.in);  
3  
4     System.out.println("Enter an int: ");  
5     while (!in.hasNextInt()) {  
6         String temp = in.next();  
7         System.out.println("You entered <" + temp + "> try again");  
8     }  
9     int x = in.nextInt();  
10    System.out.println("You entered " + x);  
11 }
```

Verifying Input

Sometimes it is necessary to make sure that values are within the correct range as well as the correct type. For example if we were reading an assignment grade from the user, we would want to make sure that it was between 0 and 100. Again we would want the user to have multiple attempts to enter the value correctly. This is more complicated than simply checking that the type is correct because we must also check that the value is correct. This will require two loops.

The code we have written already is correct, but it needs to be inside another loop that makes sure that the value is within the correct range. The easiest way to do this is have all of that code repeated as long as the value is not in the correct range.

Example: Validating the value of the input

```
1 System.out.println("Enter an integer grade: ");  
2 int grade = -1;  
3 while (grade < 0 || grade > 100) {  
4     while (!in.hasNextInt()) {  
5         String temp = in.next();  
6         System.out.println("You entered <" + temp + "> try again");  
7     }  
8     grade = in.nextInt();  
9     System.out.println("You entered " + grade);  
10    if (grade < 0 || grade > 100) {  
11        System.out.println("The value must be between 0 and 100, try again");  
12    }  
13 }
```

Just as in the first example, we create a loop with a condition that will cause the loop to be executed as long as the value is not within the correct range. In this example because we want a value between 0 and 100, we check to see if the number is less than 0 or greater than 100. We start by giving the variable a value that is not permitted, this means that we will enter the loop. The correct type is verified first and then the value is read. If the value is not within the correct range,

the loop will be executed again, so we need to add an if statement to tell the user what the correct range is and to enter the value again.

9.3 Dialogs

When developing graphical applications, generally the user is not expected to type information into the console. In fact most graphical applications will not show a console to the user. When creating these types of applications, we may want to display small pieces of information to the user or read small pieces of information from the user. We can use dialog boxes to do this.

There are a number of class methods in the class `JOptionPane` in the package `java.swing` that can be used to easily create dialogs. We will very briefly study some of these methods. There are different types, but we will only look at the message dialog and the input dialog.

To use either of these dialogs, it is important that we remember to import the `JOptionPane` class.

Message Dialog

A message dialog is simply, a small box containing a message for the user. The box will stay on the screen until the user clicks the OK button.

API: JOptionPane - showMessageDialog

public static void showMessageDialog(Component parentComponent, Object message)
Brings up an information-message dialog titled "Message".

Parameters:

parentComponent - determines the Frame in which the dialog is displayed; if null, or if the parentComponent has no Frame, a default Frame is used
message - the Object to display

We can see that the method requires two parameters, the first is the frame that the dialog box should be displayed in and the second is the message to be displayed. As we have not been using any frames, we can just pass the value `null` for the first parameter. For the second parameter, we pass the message that we want to be displayed as a `String` (even though the type says object).

The `showMessageDialog` method is a class method, this means that we do not need to create an instance of the `JOptionPane` class. The method can simply be called directly. For example, if we wanted to show the message "Hi there", we would use:

```
JOptionPane.showMessageDialog(null, "Hi there");
```

This would then result in the following window being shown.

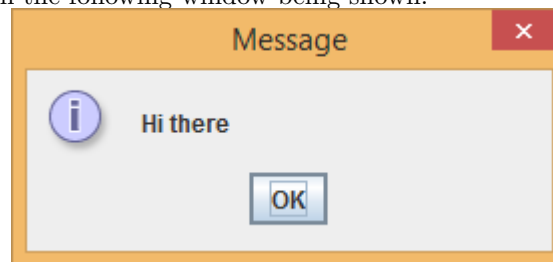


Figure 9.1: A Message Dialog Example

It is important to remember that the execution of the program will wait until the user has clicked OK. No more code will be executed until this has been done.

Input Dialog

An input dialog is similar to a message dialog, except that the user is expected to enter some information. An input dialog contains a message to the user, a box for them to enter some text and two buttons OK and Cancel.

API: JOptionPane - showInputDialog

public static String showInputDialog(Component parentComponent, Object message)
Shows a question-message dialog requesting input from the user parented to parentComponent. The dialog is displayed on top of the Component's frame, and is usually positioned below the Component.

Parameters:

parentComponent - the parent Component for the dialog
message - the Object to display

We can see that the method requires two parameters again, these are pretty much exactly the same as before. The difference here is the return type of the method. This method returns a String as a result. If the user clicks OK, the content of the text box is returned as a String, and if the user clicks cancel then null is returned. Again, because this is a class method, we can call it directly without creating a JOptionPane object. For example, if we wanted to ask the user their name, we would use:

```
String name = JOptionPane.showInputDialog(null, "What is your name?");
```

This would then result in the following window being shown.

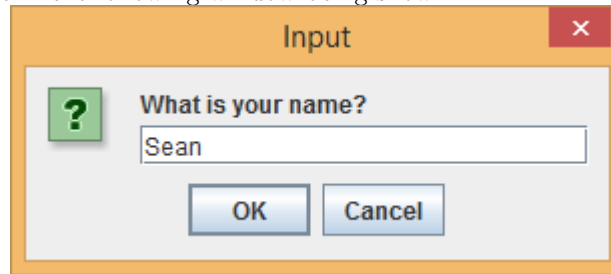


Figure 9.2: An Input Dialog Example

Just like the message dialog, the input dialog causes the code to stop executing until the OK or Cancel button has been clicked. Then the code will continue to execute normally.

The input dialog can only be used to read a string from the user. So if we wanted to read the users age, we would need to convert the string into an integer after it was read from the input dialog. This can be done using methods such as `parseInt` from the `Integer` class.

Example: Converting a String to an integer

```
1 String ageString = JOptionPane.showInputDialog(null, "What is your age?");
2 int age = Integer.parseInt(ageString);
3 System.out.println(age);
```

9.4 String Processing

Using a scanner to read each piece of information one at a time is easy and convenient, but what happens if we are given a string containing the information and need to extract it. In this case

there are a lot of methods that we can use to process a string to get the information that we want. Some useful methods in the string class are;

- `public int indexOf(String s)`
- `public String[] split (String s)`
- `public String substring(...)`
- `public char charAt(int index)`
- `public boolean equalsIgnoreCase(String s)`

Searching in a String

API: String - indexOf

public int `indexOf(String s)`

Returns the index within this string of the first occurrence of the specified substring. The returned index is the smallest value `k` for which: `this.startsWith(str, k)` is true. If no such value of `k` exists, then `-1` is returned.

Parameters:

`s` - the substring to search for.

Returns:

the index of the first occurrence of the specified substring, or `-1` if there is no such occurrence.

There are many overloaded variations of this method, but all of them can be used to search a string for a particular value. We can search for a character or a string and we can search from the beginning or from a particular index.

Example: Example usage of indexOf

```
1 String str = "Some long and overly complicated string";
2 int i = str.indexOf("om");
3 int j = str.indexOf("om", 5);
4 int k = str.indexOf('l');
5 int l = str.indexOf('l', k+1);
6 System.out.println(i + " " + j + " " + k + " " + l);
```

Breaking a String into parts

API: String - split

public String[] `split (String s)`

Splits this string around matches of the given regular expression.

The string `"boo:and:foo"`, for example, yields the following results with these expressions:

Regular Expression	Result
<code>:</code>	{ <code>"boo"</code> , <code>"and"</code> , <code>"foo"</code> }
<code>o</code>	{ <code>"b"</code> , <code>"a"</code> , <code>":and:f"</code> }

Parameters:

`s` - the delimiting regular expression

Returns:

the array of strings computed by splitting this string around matches of the given regular expression

While the API talks about regular expressions, we can ignore this and simply choose the character or string that we will use to break up a string. This will return an array to us that we can then more easily process.

Example: Example of the usage of `split` to break up a string

```

1 String str = "123 45 12 5 3";
2 String[] splits = str.split(" ");
3 int sum = 0;
4 for (int i = 0; i < splits.length; i++) {
5     sum += Integer.parseInt(splits[i]);
6 }
7 System.out.println(sum);

```

In this example we are using the space character break up the string, that way we are left with an array containing strings each with an integer value represented. These strings can then be easily parsed and used in a calculation.

Getting a section of a String

API: String - `substring`

public String `substring(int beginIndex, int endIndex)`

Returns a string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex - beginIndex`. Examples:

"hamburger".`substring(4, 8)` returns "urge"

"smiles".`substring(1, 5)` returns "mile"

This can be used to pull the parts of a string that we want and ignore the parts that we do not want. This is particularly useful when combined with `indexOf`.

Example: Using `substring` to get a piece of a string

```

1 String str = "text t (123) more text";
2 int s = str.indexOf('(');
3 int e = str.indexOf(')');
4 System.out.println(str.substring(s+1, e));

```

Here this example will find all of the text that is between the first set of brackets.

Getting a single character

API: String - `charAt`

public char `charAt(int index)`

Returns the char value at the specified index. An index ranges from 0 to `length() - 1`. The first char value of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

Parameters:

`index` - the index of the char value. **Returns:**

the char value at the specified index of this string. The first char value is at index 0.

This will simply give you the requested character and does not need an example.

Comparing Strings

As we learned when studying inheritance, all objects have a method called `equals` which can be used to determine if two objects are the same. There are actually two implementations of this for the `String` class, `equals` and `equalsIgnoreCase`. The first only returns true if the strings are exactly the same, the second returns true after comparing the letters being represented and ignoring whether they were upper case or lower case.

API: `String - equalsIgnoreCase`

public boolean `equalsIgnoreCase(String s)`

Compares this `String` to another `String`, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length and corresponding characters in the two strings are equal ignoring case.

Parameters:

`s` - The `String` to compare this `String` against

Returns:

true if the argument is not null and it represents an equivalent `String` ignoring case; false otherwise

Example: Comparing two Strings

```
1 String str = "hello";
2 String not = "HELLO";
3 System.out.println(str.equalsIgnoreCase(not));
```

9.5 Using a Scanner to Process A String

If we have been given a `String` from another method and we want to get information from it, we can use a scanner to do this. This uses another constructor of the scanner class, this one takes a string as a parameter.

API: `Scanner - Constructor`

public `Scanner(String source)`

Constructs a new `Scanner` that produces values scanned from the specified `String`.

Parameters: `source` - An input stream to be scanned

We can then use the scanner to read the individual tokens in the string one at a time and in the correct type.

Example: Using a Scanner to get information from a String

```
1 String line = "100 45 90 40.80 80.40";
2 Scanner s = new Scanner(line);
3 int i = s.nextInt();
4 int j = s.nextInt();
5 int k = s.nextInt();
6 double l = s.nextDouble();
7 double m = s.nextDouble();
8 System.out.println(i+j+k+l+m);
```

9.6 Practical Exercises

1. Define a class named **ReadInt**, this class should contain a main method. In the main method you should complete the following operations:
 - Define and construct a scanner to read input from the user
 - Read a single integer from the user and store it a variable
 - Print the value of the variable multiplied by 2
2. Define a class named **ReadDouble**, this class should contain a main method. In the main method you should complete the following operations:
 - Define and construct a scanner to read input from the user
 - Read a single double from the user and store it a variable
 - Print the value of the variable multiplied by 1.5
3. Define a class named **ReadWords**, this class should contain a main method. In the main method you should complete the following operations:
 - Define and construct a scanner to read input from the user
 - Read 3 words from the user and store them in variables (or in an array)
 - Print the words that were entered in reverse order (on a single line separated by spaces)
4. Define a class named **Calculator**, this class should contain a main method. In the main method you should complete the following operations:
 - Print the following message to the screen **"Enter the first number"**
 - Read an int from the user using a scanner and store it in a variable
 - Print the following message to the screen **"Enter the second number"**
 - Read an int from the user using the same scanner and store it in a variable
 - Print the following messages
 - **"The sum of X and Y is A"**
 - **"X minus Y is B"**
 - **"X divided by Y is C"**
 - **"X multiplied by Y is D"**
 - **"X modulus Y is E"**
5. Define a class named **TryAgain**, this class should contain a main method. In the main method you should complete the following operations:
 - Define and construct a scanner to read input from the user
 - Print the message **"Please enter an integer"**
 - Read a single integer from the user
 - If the user does not enter an integer print the message **"you entered XXX, please enter an integer"**
 - Give the user another attempt to enter an integer
 - Print a message to the user saying **"congratulations, you entered Y"**
6. Define a class named **TryMany**, this class should contain a main method. In the main method you should complete the following operations:
 - Define and construct a scanner to read input from the user
 - Print the message **"Please enter an integer"**

- Read a single integer from the user
 - As long as the user has not entered an integer you should print the message "you entered XXX, please enter an integer"
 - Give the user another attempt to enter an integer
 - Print a message to the user saying "congratulations, you entered Y"
7. Define a class named **Student**. The class should have private instance variables to represent the following information, which is passed in using a constructor
- The first name of the student, as a variable called **firstName**
 - The family name of the student, as a variable called **familyName**
 - The student number of the student as a variable called **studentNumber**
 - The age of the student as a variable called **age**
 - The name of the degree they study (IOT or SE) as a variable called **degreeName**
8. Override the **toString** method in the **Student** class. This method should print the students information as a single line of text in the following format:
"Name: XXXX YYYY, Student number: ZZZZ, Age: WW, Degree: UUU"
9. Define a class named **TestStudentInput**. This class should have a main method containing the following statements:
- Define and construct a scanner to read input from the user
 - Define and construct an array of Student objects of size 3
 - Read the first name of a student from the user and store it in a variable
 - Read the family name of a student from the user and store it in a variable
 - Read the number of a student from the user and store it in a variable
 - Read the age of a student from the user and store it in a variable
 - Read the name of the degree that the student is studying and store it in a variable
 - Construct a student object based on the values read from the user and store it in the array
 - Repeat the process two more times
 - In reverse order, print the student number and degree of each student on a single line

Chapter 10

Graphical Input and Output

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ be able to draw shapes on the screen
- ☐ understand the concept of callbacks and how these are used with interfaces
- ☐ to be able to use callbacks to interact with the user

In this chapter we will study the basics of drawing different shapes on the screen. Following this we will study the use of callbacks to allow the user to interact in different ways using the keyboard and the mouse.

10.1 Drawing on the Screen

In chapter 7 we saw some examples of code that draws on the screen, but this code was not explained in detail. We will look at this process as a nice example of object-oriented programming principles.

When to Draw?

The first thing we must understand about writing code to draw on the screen, and all other linked activities, is that it is difficult to know when our code should be executed. For example, the code will need to be executed when the image is first drawn on the screen, whenever the window is moved, if another window is moved in front of it, when the window is resized and in many other situations. Knowing when any of these situations has happened is very complicated, but fortunately the classes in the library will do all of this for us.

But we still have to write the code, how will these classes know what code to execute when the screen is drawn? We let the classes know by implementing specific methods within our classes, the JVM will then call these methods at the correct times automatically. This idea is called inversion of control and we will look at it in more detail next year. The main idea is that we create a class that extends a class in the library for drawing, in this class we will have to implement the correct method that tells Java what should be drawn. Then we only have to tell Java what objects it should draw on the screen.

Creating a Window

The first thing we need is somewhere to draw our images, this is done in a window. The steps to create a window and draw on the screen are as follows:

1. Create a window (called a JFrame in Java)

2. Set the size of the window
3. Tell it to shut when the X button is pressed
4. Create a painted component to be displayed
5. Add our painted component to the JFrame
6. Set the JFrame to be visible on the screen

Example: Creating a window on the screen that will display our drawing

```
1 public class Window {  
2     public static void main(String[] args) {  
3         JFrame window = new JFrame();  
4         window.setSize(300, 400);  
5         window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
6         Picture pic = new Picture();  
7         window.add(pic);  
8         window.setVisible(true);  
9     }  
10 }
```

This code will create a window with a width of 300 and height of 400 and will draw shapes that are defined by the class `Picture`. But what does the `Picture` class draw? This is something that we still have to define. In order to define the `Picture` class, we need to know what type of object can be added to the `JFrame`.

API: `JFrame` - `add` (Inherited from `Container`)

public `Component` `add`(`Component` `comp`)
Appends the specified component to the end of this container.
Parameters: `comp` - the component to be added

The `add` method shows that a component can be added to a `JFrame`. This means that we need to extend the `Component` class or one of its subclasses, if we want our picture class to be added to the `JFrame`. The abstract `JComponent` class is the class that most graphical components extend and is a subclass of `Component`, we will extend this. Because most of the work of drawing on the screen is taken care of by the JVM, we need to know what method to implement to draw our picture. The method `paintComponent` is called automatically when the contents of a `JComponent` should be drawn, this is where our drawing code should go.

API: JComponent - paintComponent**protected void** paintComponent(Graphics g)

Calls the UI delegate's paint method, if the UI delegate is non-null. We pass the delegate a copy of the Graphics object to protect the rest of the paint code from irrevocable changes (for example, Graphics.translate). If you override this in a subclass you should not make permanent changes to the passed in Graphics. For example, you should not alter the clip Rectangle or modify the transform. If you need to do these operations you may find it easier to create a new Graphics from the passed in Graphics and manipulate it. Further, if you do not invoke super's implementation you must the opaque property, that is if this component is opaque, you must completely fill in the background in a non-opaque color. If you do not honor the opaque property you will likely see visual artifacts.

The passed in Graphics object might have a transform other than the identify transform installed on it. In this case, you might get unexpected results if you cumulatively apply another transform.

Parameters: g - the Graphics object to protect

This API excerpt contains a lot of complicated information, most of which you will not understand until you study computer graphics. All we really need to know about it is the signature of the method. **protected void** paintComponent(Graphics g). Now we can define our picture class and be ready for drawing.

Example: The Picture class ready for drawing shapes

```

1 public class Picture extends JComponent {
2     protected void paintComponent(Graphics g) {
3         // Shape drawing code goes here
4     }
5 }
```

In this example, the code we add to the `paintComponent` method will be called whenever the JVM needs to draw the an object based on the Picture class. In the main method of the Window class we have told the JVM that an object of the Picture class should be drawn by adding one to the JFrame object we created.

Screen Geometry

Before we can begin drawing shapes on the screen, we need to understand a little bit more about the how shapes are represented on the screen. The most important point is to understand the coordinate system that is used in computer graphics.

When computers were first being used with screens, these screens worked by firing a stream of electrons at a phosphorescent screen. When the electrons struck the screen that part would light up in the correct colour. The electrons were fired by an electron gun, which moved through the screen one line at a time from left to right. The first line was at the top of the screen and the last was on the bottom. This entire process would happen many times every second.

Because the electron gun started at the top of the screen and moved down, the coordinate system that was used has the origin at the top left of the screen and the value of y increases as the we move down. Figure 10.1 shows a comparison of this coordinate system with the traditional coordinate system used in normal calculations.

Screen coordinates are counted in pixels, where each pixel is a single dot on the screen. As we can not draw anything smaller, this means that all of the numbers we use will be integers.

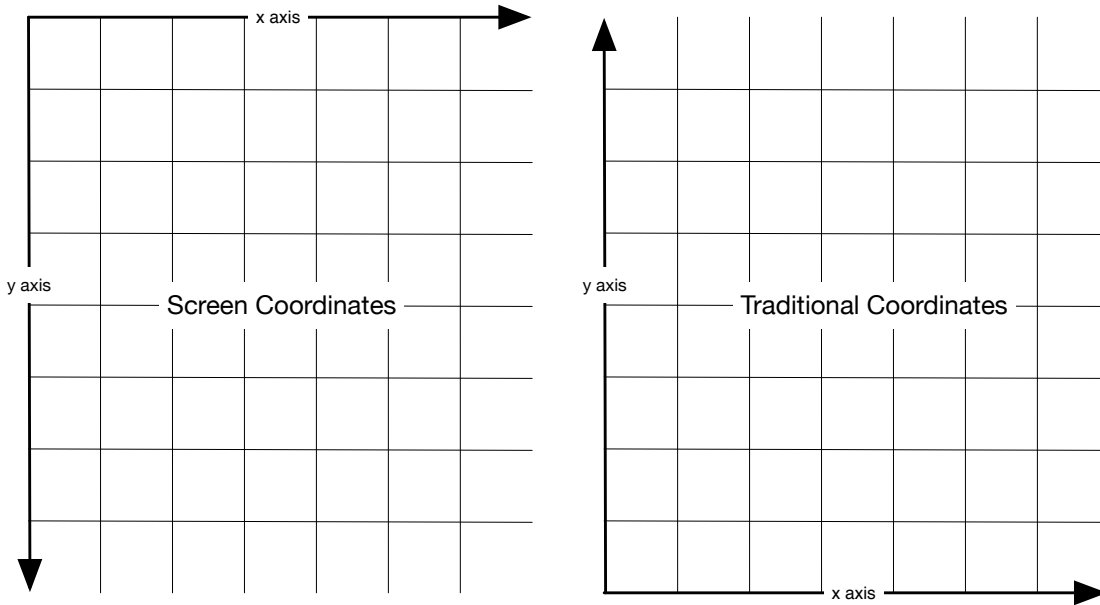


Figure 10.1: Screen coordinates compared with traditional coordinates

Drawing

Now that we have the window created and our picture is going to be drawn in the window, we can add some code to actually draw something. We noted earlier that the drawing code is going to be within the method `paintComponent`, now we must look at how we actually draw. Drawing is done using the parameter that is passed to this method, **Graphics** `g`. The graphics class contains many methods that can be used to draw various shapes on the screen, some of these methods are easy to use and others are more complicated. For complete details see the Graphics class in the API.

Here is a list of some of the easiest methods to use when drawing on the screen.

- **void** `drawLine(int x1, int y1, int x2, int y2)`
- **void** `drawOval(int x, int y, int width, int height)`
- **void** `drawPolygon(Polygon p)`
- **void** `drawRect(int x, int y, int width, int height)`
- **void** `fillOval(int x, int y, int width, int height)`
- **void** `fillRect(int x, int y, int width, int height)`
- **void** `fillPolygon(Polygon p)`

Additionally, the colour that these shapes are drawn in can be changed by calling the method `setColor`, which takes a color object as a parameter. Typically, we will use colors that are already represented as constants in the `Color` class, such as `Color.black`, `Color.green`, `Color.blue`. However, we can also create our own colours by creating a new color object by passing in values for red, green and blue to the constructor.

We are going to approach this topic in an object-oriented way, our picture will be made up of a number of smaller drawings and each of these will be represented by a single class. Each of these classes will extend the `Abstract2DDrawing` class (Shown in chapter 7). Inside the draw method of each class we will add the code to draw each smaller drawing.

We will start with a very basic drawing of a building shape on a skyline, this can be represented by a simple rectangle. Our basic building will be 200 pixels high and 50 pixels wide. The abstract 2D drawing class contains a constructor which sets the x and y location of the drawing. As we create multiple objects they can be placed in different locations, this means the when we are drawing the shapes, they need to be drawn using **relative coordinates**.

If we want to place a building at position (50, 50), then the value of x and y are set to 50. When we draw the rectangle, we use the values of x and y to call the method. The only code we need in our draw method is `g.fillRect(x, y, 50, 200);`.

Example: Completed Building class

```
1 public class Building extends Abstract2DDrawing {  
2     public Building(int x, int y) {  
3         super(x, y);  
4     }  
5     public void draw(Graphics g) {  
6         g.fillRect(x, y, 50, 200);  
7     }  
8 }
```

Now that we have completed this class, we can finally add the required code to the `paintComponent` method in the `picture` class to add our buildings. The nice thing about the strategy that we have used is that based on the same class, we can add the same drawing multiple times by creating multiple objects.

Example: Create two building objects

```
1 protected void paintComponent(Graphics g) {  
2     Building b = new Building(50, 150);  
3     b.draw(g);  
4     Building b2 = new Building(150, 150);  
5     b2.draw(g);  
6 }
```

This code creates one buildings at the point (50, 150) and a second at the point (150,150). This buildings are then drawn on the screen by calling the draw method on each object. This gives is the picture shown in the window below.

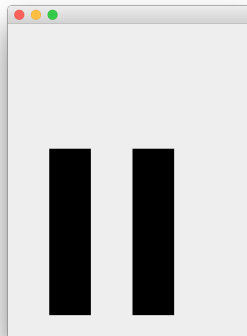


Figure 10.2: Two building objects drawn on the screen

For simple examples like this, using a class to represent a building is more complicated than simply drawing the shapes in the `paintComponent` method. However, as the individual pieces get more complicated, the use of this simplifies the process. We will show this by making our picture more complicated, adding windows to the image will make these shapes look more like buildings.

Windows can be added by drawing smaller rectangles in a different colour in a rows on top of the rectangle we have already drawn. As our building is 50 pixels wide we can easily fit 3 10 by 10 windows on each floor and as it is 200 pixels high we can easily add 12 floors of windows.

Drawing a single floor of windows can be done using a loop that executes 3 times, we just need to use the index of the loop to calculate the position of the window. There should be a gap of 5 between each window, so we can calculate the x coordinate of the window by multiplying $i * 15$ (width of window + gap) and adding 5 (gap at beginning). However, because all of these buildings can be moved, the coordinates of the window must be relative to the position of the building itself, we must also add x.

The same process is used to calculate the y coordinate of each window, so we end up with nested loops containing something like this: `g.fillRect(5 + x + j * 15, 5 + y + i * 15, 10, 10);`. But before we draw the window, we must first change the colour. Additionally, we must also make sure the colour is correct before we draw the main rectangle.

Example: Updated draw method from the building class

```

1 public void draw(Graphics g) {
2     g.setColor(Color.black);
3     g.fillRect(x, y, 50, 200);
4     g.setColor(Color.YELLOW);
5     for (int i = 0; i < 12; i++) {
6         for (int j = 0; j < 3; j++) {
7             g.fillRect(5 + x + j * 15, 5 + y + i * 15, 10, 10);
8         }
9     }
10 }

```

This gives us the following picture.

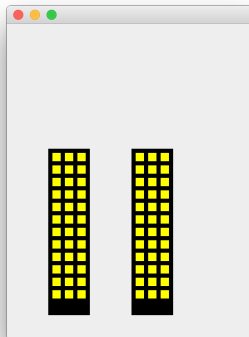


Figure 10.3: Two building objects drawn on the screen with windows

The benefits of using classes for each of these drawings is that we can also extend the class to make a slightly different picture. For example, if we wanted to add a sloped roof to some of the buildings but not all. To do this we create a new class that extends the `Building` class called `RoofBuilding`. This will be a slightly bigger building because we are adding a roof to the top, so in the constructor we move the building down by adding 20 (the height of the roof) to the value of `y` when we pass it to the super constructor.

To draw the roof we have to override the draw method of the building class. However, because we have already written the code to draw the building, we do not want to write it again. Instead we first call the draw method of the superclass. Then we can add the code to draw our roof.

Example: Roof Building class

```
1 public class RoofBuilding extends Building{
2     public RoofBuilding(int x, int y) {
3         super(x, y+20);
4     }
5     public void draw(Graphics g) {
6         super.draw(g);
7         g.setColor(Color.black);
8         Polygon p = new Polygon();
9         p.addPoint(x, y);
10        p.addPoint(x, y-20);
11        p.addPoint(x+50, y);
12        g.fillPolygon(p);
13    }
14 }
```

If we change the code in our picture class to create one building object and one roof building, we end up with a picture that looks like this:

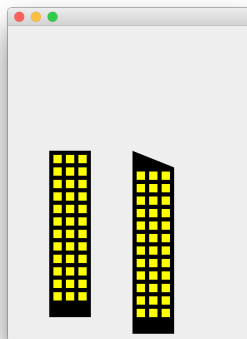


Figure 10.4: Two building objects drawn on the screen, one with a roof

10.2 Using the mouse to interact

In the same way as it is difficult to know when the drawing code should be executed, it is also difficult to know when the mouse has been used. Fortunately there are parts of the JVM that we can use to be told whenever different things happen to with the mouse. This works based on the concept of **callback** methods. A callback method is a method that we implement knowing that it will be called whenever a particular event happens, such as the mouse being clicked.

The functionality of callbacks is implemented using interfaces in Java. This is a nice idea because the interface will tell us what methods will be called when something happens. Classes that implement these interfaces are often called **listeners**, because they listen for when an even happens.

When interacting using the mouse, we need to implement the **MouseListener** interface from the package `java.awt.event`. The mouse listener interface requires that the following methods are implemented:

- **void** mouseClicked(MouseEvent e)
This method is called whenever the mouse is pressed down and released
- **void** mousePressed(MouseEvent e)
This method is called whenever the mouse is pressed down
- **void** mouseReleased(MouseEvent e)
This method is called whenever the mouse is released
- **void** mouseEntered(MouseEvent e)
This method is called whenever the mouse moves inside the window
- **void** mouseExited(MouseEvent e)
This method is called whenever the mouse moves outside the window

Mostly, we will only add code to one or two of the methods and the rest will be left empty, this is because the interface provides us with many options of when our code could be called. In our example, we will implement some code that will be executed whenever the mouse is clicked, so we will add our code to the first method. What we will make this code do is turn the lights on and off in our building drawing, however this will require us to add and change some code in the Picture and Building classes first.

First we have to change the code of the Building class so that the lights can be turned on and off, this can be done by changing the colour that the windows are drawn in. So we need to add an instance variable that holds the value as to if the lights are on or off. Because this variable has only two possible values, it should be implemented as a boolean variable. So we add **private boolean** lights = **true**; to the Building class (and its subclasses automatically).

Secondly, we need to choose the colour of the windows based on the value of the variable **lights**. The most sensible and understandable is to have the lights be turned on whenever the value of lights is **true**. So we add an if-else statement that sets the colour before we draw the windows to either yellow for lights on or blue for lights off.

Example: Selecting the colour of the windows

```
1 if (lights) {  
2     g.setColor(Color.YELLOW);  
3 } else {  
4     g.setColor(Color.BLUE);  
5 }
```

Lastly, we need to add some way for this to be changed from outside of the class. So we add a method to the class that passes in a boolean value and changes the value of lights to match.

Example: Allowing the lights to be turned on or off from the outside

```
1 public void setLights(boolean b){  
2     lights = b;  
3 }
```

Next we need to have some way in the Picture class for turning the lights on and off in the buildings. However, in the code we designed earlier the Building objects are declared whenever the paint component method is called and therefore they only exist inside this method. We need to instead make these objects instance variables of the class if we want to be able to change them from other methods.

Like in the building class, we also need to add a method to allow the lights to be turned on and off. The method is basically the same signature but in the code we just call the method on each of the building objects. We end up with our picture class looking like this:

Example: Update picture class for lights

```

1 public class Picture extends JComponent {
2     Building b = new Building(50, 150);
3     RoofBuilding b2 = new RoofBuilding(150, 150);
4
5     protected void paintComponent(Graphics g) {
6         b.draw(g);
7         b2.draw(g);
8     }
9
10    public void setLights(boolean l){
11        b.setLights(l);
12        b2.setLights(l);
13    }
14 }

```

Now we are finally ready to implement our mouse interaction, we will do this in a class called **LightListener**. For the light listener to be able to call the method of our picture object, it must have a reference to it. So we add a constructor that requires a picture object and an instance variable of a picture object. We also need to know if the lights were on or off already, so we need an instance variable to remember what the value of lights was, we will call this **lightOn** and set the initial value to true because the lights will be on when the picture is first drawn.

In the **mouseClicked** method, we first add the code to change the value of the variable **lightOn**, the easiest way to do this is to use the not operator to set it to the opposite of its previous value. Second we add the code to call the set light method in the picture object to the new value of **lightOn**. This is enough that when the picture is drawn again the lights will no be changed. However, the JVM does not know that the picture has changed and needs to be drawn again. Lastly we call a method on the picture object called **repaint**, this tells the system that the picture object needs to be drawn again and the system will call the required methods.

Example: The light listener fully implemented

```

1 public class LightListener implements MouseListener {
2     private Picture picture;
3     private boolean lightOn = true;
4     public LightListener(Picture p){
5         picture = p;
6     }
7
8     public void mouseClicked(MouseEvent e) {
9         lightOn = !lightOn;
10        picture.setLights(lightOn);
11        picture.repaint();
12    }
13    public void mousePressed(MouseEvent e) {}
14    public void mouseReleased(MouseEvent e) {}
15    public void mouseEntered(MouseEvent e) {}
16    public void mouseExited(MouseEvent e) {}
17 }

```

So far we have changed the code to allow the lights to be turned on and off and we have defined a class that knows what to do when the mouse is clicked. This leaves two important tasks still to be completed. First we need to actually create an object based on the **LightListener** class.

Secondly we need to tell the window that we want to be told whenever one of these events actually happens.

Example: Window code updated for the light listener

```

1 public class Window {
2     public static void main(String[] args) {
3         JFrame window = new JFrame();
4         window.setSize(300, 400);
5         window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6         Picture pic = new Picture();
7         window.add(pic);
8         LightListener ll = new LightListener(pic);
9         window.addMouseListener(ll);
10        window.setVisible(true);
11    }
12 }

```

On line 8 of the above example, we create an object (11) based on the light listener class. More importantly on line 9 we tell the `JFrame` that we are interested in hearing about mouse events by adding the light listener object as a mouse listener. This now means that whenever the system detects that a mouse click has happened within our window, the `mouseClicked` method in the light listener object will be called and the lights will be changed.

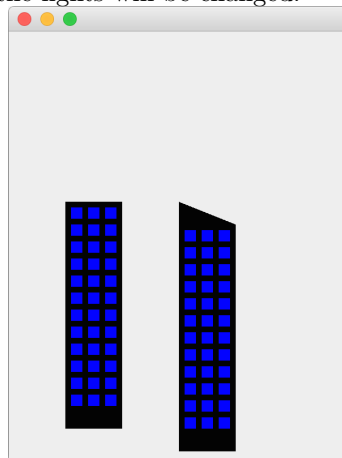


Figure 10.5: Two building objects drawn on the screen, with the lights turned off

It is also possible to use the motion of the mouse to control our applications. This is done using by implementing the `MouseMotionListener` interface. This interface has only two method, `void mouseDragged(MouseEvent e)` and `void mouseMoved(MouseEvent e)`. Once created a mouse motion listener class can be added as a listener using the method `void addMouseMotionListener(MouseMotionListener m)`.

If we are required to implement listeners for both the mouse clicks as well as motion of the mouse, we can simplify the process by using the `MouseInputAdapter` class. This is an abstract class that implements both mouse listener interfaces and provides an empty implementation for each method. We simply need to extend the class and override the methods that we want to use. The object created based on this class needs to be added both as a mouse listener and as a mouse motion listener.

10.3 Interacting using the keyboard

Interacting using the keyboard, is based on the same technique as the mouse. The main difference is that we use a different interface. This time we use the `KeyListener` interface, this interface contains the following methods;

- **void** `keyTyped(KeyEvent e)`
This method is called whenever a character key (not function keys) is pressed and released
- **void** `keyPressed(KeyEvent e)`
This method is called whenever any key is pressed down
- **void** `keyReleased(KeyEvent e)`
This method is called whenever any key is released

Just as in the mouse listener interface, a single parameter is passed to the method when it is called. In the mouse example, we ignored this parameter as we did not need to know any details about the individual clicks. However, when using the keyboard to interact, we will want to know which of the keys has been pressed and this information is contained in the `KeyEvent` parameter.

Key events

Every key event object contains a number that represents the key that was pressed. We can access this number by calling the method `getKeyCode`. This method requires no parameters and returns an int value.

API: KeyEvent - getKeyCode

```
public int getKeyCode()
```

Returns the integer `keyCode` associated with the key in this event.

Returns: the integer code for an actual key on the keyboard.

For each key on the keyboard, there is a constant in the `KeyEvent` class that contains the value for that key. These constants have named like `VK_LEFT` or `VK_A` and can be looked up in the documentation of the `KeyEvent` class. This allows us to write code that does a different action depending on what key was pressed. For example, if we wanted to move an object when the left key was pressed we would add an if statement with the condition `e.getKeyCode() == KeyEvent.VK_LEFT`.

For this part of the example, we are going to add a new piece to our drawing. This time we will add a moon to the sky above our buildings. We create a new class called `Moon` that extends the `Abstract2dDrawing` class. This class will just add a circle wherever it is positioned.

Example: Class for drawing the moon

```
1 public class Moon extends Abstract2DDrawing {
2   public Moon(int x, int y) {
3     super(x, y);
4   }
5   public void draw(Graphics g) {
6     g.setColor(Color.GRAY);
7     g.fillOval(x, y, 50, 50);
8   }
9 }
```

After adding this to the picture class as an instance variable and in the paint component method, we need to add a method that changes the position of the moon in the x and y directions. This method requires two parameters, the first to tell us how much to move in the x direction and the

second to tell us how much to move in the second direction. The values in these parameters are then passed to the move method in Moon object.

Example: Code to call the move method of the Moon object

```
1 public void moveMoon(int x, int y){
2     moon.move(x, y);
3 }
```

Now we have added an object that can be moved in the picture and a method that we can call to move it. Now we need to create our key listener class. Just as with the light listener class, we will require an instance variable to store a reference to the picture object and constructor to pass the object. We will implement the `keyPressed` method and have the moon move any time one of the direction keys are pressed.

To do this we will have an if statement for each of the values `VK_LEFT`, `VK_RIGHT`, `VK_UP` and `VK_DOWN`. Within the body of each of these if statements, we will call the `moveMoon` method and pass in the parameters required to have the moon move 10 pixels in the correct direction. Finally, we also need to tell the picture that it needs to be repainted, this could be done individually after the `moveMoon` method was called, but it is easier to just add it once after the if statements.

Example: Completed moon listener class

```
1 public class MoonListener implements KeyListener {
2     private Picture picture;
3     public MoonListener(Picture p){
4         picture = p;
5     }
6     public void keyTyped(KeyEvent e) {}
7     public void keyReleased(KeyEvent e) {}
8     public void keyPressed(KeyEvent e) {
9         if(e.getKeyCode() == KeyEvent.VK_LEFT){
10            picture.moveMoon(-10, 0);
11        } else if(e.getKeyCode() == KeyEvent.VK_RIGHT){
12            picture.moveMoon(10, 0);
13        } else if(e.getKeyCode() == KeyEvent.VK_UP){
14            picture.moveMoon(0, -10);
15        } else if(e.getKeyCode() == KeyEvent.VK_DOWN){
16            picture.moveMoon(0, 10);
17        }
18        picture.repaint();
19    }
20 }
```

Again, we need to complete the final two steps or our code will never be executed. We still must create an object based on the `MoonListener` class and tell the system that we are interested in hearing about key events and that the methods of the `MoonListener` object should be called.

Example: Code to tell the system we are interested in key events

```
1 MoonListener ml = new MoonListener(pic);
2 window.addKeyListener(ml);
```

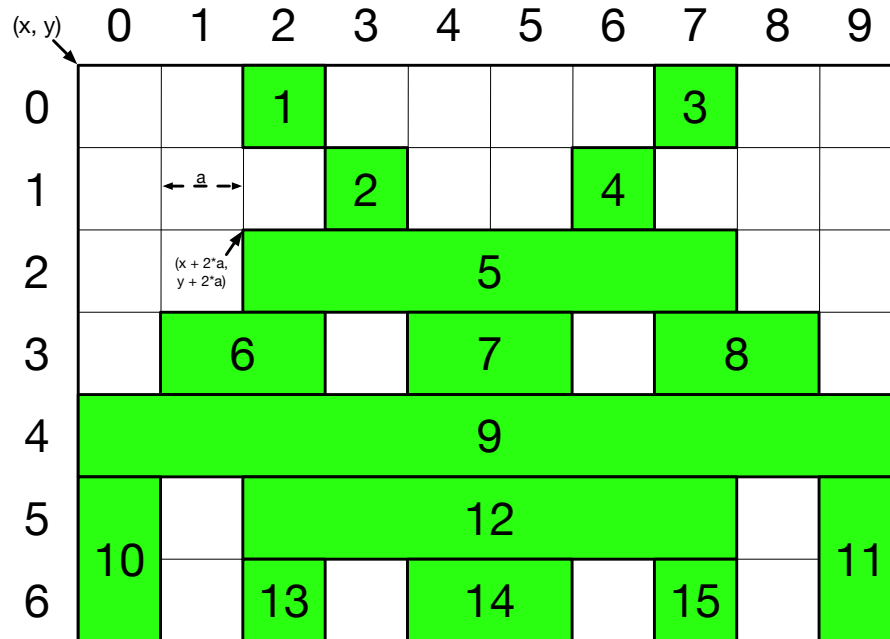


Figure 10.6: Shape of the type A alien ship

10.4 GUI Game Example

In the previous sections, we looked at the basic theory of how to draw shapes on the screen. We will now apply this theory to the game we have developed. We will start by defining the coordinates at which parts of a single ship should be drawn.

These shapes belong to parts of the game that are expected to move across the screen. Therefore we must define the locations drawn relative to the x and y location of the object itself.

Figure 10.6 shows the shape of the type A alien ship. The drawing is marked with lines to make calculating the positions of the individual points easier, where each line is the size of one block (referred to as a). The top left point of the drawing is considered the coordinates of the ship (x, y) the rest of the parts of the ship must be described in terms of these values. For simplicity the individual rectangles are highlighted and numbered to make describing them easier.

Starting with rectangle number 1, we would define the rectangle using the parameters $(x + 2 * a, y, a, a)$ for the coordinates of the top left corner and the width and height. This places that block in the correct position on the screen and is adjusted correctly from the calculated location of the ship. The formula for the x position is actually very simple, x plus the size of the block multiplied by the number of blocks over. This can be repeated in the y location except the number of blocks down in this case is zero.

The four parameters can be shown here $(x + i * a, y + j * a, w * a, h * a)$ where i is the number of the column of the top left of the rectangle, j is the number of the row of the top left of the rectangle and w and h represent the number of blocks wide and high respectively. Using this we can easily define the values for all of the rectangles in the shape, which is shown in the following table.

Number	x Location	y Location	Width	Height	Coefficients
1	$x + 2 * a$	$y + 0 * a$	$a * 1$	$a * 1$	{ 2, 0, 1, 1 }
2	$x + 3 * a$	$y + 1 * a$	$a * 1$	$a * 1$	{ 3, 1, 1, 1 }
3	$x + 7 * a$	$y + 0 * a$	$a * 1$	$a * 1$	{ 7, 0, 1, 1 }
4	$x + 6 * a$	$y + 1 * a$	$a * 1$	$a * 1$	{ 6, 1, 1, 1 }
5	$x + 2 * a$	$y + 2 * a$	$a * 6$	$a * 1$	{ 2, 2, 6, 1 }
6	$x + 1 * a$	$y + 3 * a$	$a * 2$	$a * 1$	{ 1, 3, 2, 1 }
7	$x + 4 * a$	$y + 3 * a$	$a * 2$	$a * 1$	{ 4, 3, 2, 1 }
8	$x + 7 * a$	$y + 3 * a$	$a * 2$	$a * 1$	{ 7, 3, 2, 1 }
9	$x + 0 * a$	$y + 4 * a$	$a * 10$	$a * 1$	{ 0, 4, 10, 1 }
10	$x + 0 * a$	$y + 5 * a$	$a * 1$	$a * 2$	{ 0, 5, 1, 2 }
11	$x + 9 * a$	$y + 5 * a$	$a * 1$	$a * 2$	{ 9, 5, 1, 2 }
12	$x + 2 * a$	$y + 5 * a$	$a * 6$	$a * 1$	{ 2, 5, 6, 1 }
13	$x + 2 * a$	$y + 6 * a$	$a * 1$	$a * 1$	{ 2, 6, 2, 1 }
14	$x + 4 * a$	$y + 6 * a$	$a * 2$	$a * 1$	{ 4, 6, 2, 1 }
15	$x + 7 * a$	$y + 6 * a$	$a * 1$	$a * 1$	{ 7, 6, 1, 1 }

Now that we know the details of the individual rectangles, we need to find some way to encode this information. As the shape of the ships is associated with the type, this would seem like a logical place to put it. However the type does not and cannot know about the x and y location of the ships. This means that we cannot create the rectangles, but we can remember the important information required to create them, the coefficients of a in the formula.

Example: Shape coefficients encoded as 2D array

```

1 public enum AlienType {
2     A(10, 7, 10, new int [][] { {2, 0, 1, 1}, {3, 1, 1, 1}, {7, 0, 1, 1},
3         {6, 1, 1, 1}, {2, 2, 6, 1}, {1, 3, 2, 1}, {4, 3, 2, 1},
4         {7, 3, 2, 1}, {0, 4, 10, 1}, {0, 5, 1, 2}, {9, 5, 1, 2},
5         {2, 5, 6, 1}, {2, 6, 1, 1}, {4, 6, 2, 1}, {7, 6, 1, 1} } ),
6     B(8, 8, 20, new int [][] { ... } ),
7     C(10, 8, 15, new int [][] { ... } );
8     private int width, height, score;
9     private int [][] shapes;
10
11     private AlienType(int w, int h, int s, int [][] ss) {
12         width = w;
13         hieght = h;
14         score = s;
15         shapes = ss;
16     }
17     public int getWidth() { ... }
18     public int getHeight() { ... }
19     public int getScore() { ... }
20     public int [][] getShapeCoefficients() {
21         return shapes;
22     }
23 }

```

This allows us to write a single method that can draw all three of the different types of ships on the screen. Objects to be drawn on the screen were to implement the drawable interface, which requires the implementation of the method `getShape`. Once we have completed this for the Ship and Enemy class we can add the required code to draw the ships on the screen.

Example: The `getShape` method for the `Enemy` class

```

1 public Rectangle[] getShape() {
2     int[][] co = type.getShapeCoefficients();
3     Rectangle[] shapes = new Rectangle[co.length];
4     for(int i = 0; i < co.length; i++) {
5         shapes[i] = new Rectangle(x + co[i][0] * BRICK_SIZE, y + co[i][1] *
6             BRICK_SIZE, co[i][2] * BRICK_SIZE, co[i][3] * BRICK_SIZE);
7     }
8     return shapes;
9 }

```

This method will use the coefficients as well as a constant containing the number of pixels in each block to create an array of rectangle objects to represent the enemy ship on the screen. Next we need to add a method to the `Game` class that allows returns the drawable objects to be shown on the screen.

Example: Returns the drawable objects.

```

1 public Drawable[] getDrawable() {
2     Drawable[] drawable = new Drawable[4];
3     drawable[0] = player;
4     drawable[1] = enemyShips[0];
5     drawable[2] = enemyShips[1];
6     drawable[3] = enemyShips[2];
7     return drawable;
8 }

```

As we make our game more complicated, this will be changed to allow more enemies to be returned as well as adding the bunkers and bullets that have been fired too. We need something to be drawn on the screen, so we create a class called `GameScreen` to represent the game while it is being played.

Example: The `JComponent` we will use to draw the game

```

1 public class GameScreen extends JComponent {
2     private static final long serialVersionUID = -8282302849760730222L;
3     private Game game;
4     public GameScreen(Game g) {
5         game = g;
6     }
7     protected void paintComponent(Graphics g) {
8         g.setColor(Color.black);
9         g.fillRect(0, 0, Game.SCREEN_WIDTH, Game.SCREEN_HEIGHT);
10        g.setColor(Color.green);
11        for (Drawable d : game.getDrawable()) {
12            for (Rectangle s : d.getShape()) {
13                g.fillRect(s.x, s.y, s.width, s.height);
14            }
15        }
16    }
17 }

```

Finally, we need a window to display our game screen in, so we add a window class.

Example: The window class for the game

```
1 public class Window {  
2     public static void main(String[] args) {  
3         JFrame window = new JFrame();  
4         window.setSize(Game.SCREEN_WIDTH, Game.SCREEN_HEIGHT);  
5         window.setResizable(false);  
6         window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
7         window.setTitle("Space Invaders");  
8         window.setLocationRelativeTo(null);  
9         GameScreen gs = new GameScreen(new Game());  
10        window.add(gs);  
11        window.setVisible(true);  
12    }  
13 }
```

Many of these tasks will be left to be implemented by the student because they are repetitive and not very interesting.

Game Control

Just like before we are going to create a keyboard listener to control the player in the game, but this time we will do things a little bit differently. The key listener is not going to be causing any code to be executed, instead it will simply remember when important keys have been pressed. Other parts of the game will use the key listener to check if the keys have been pressed and then make changes to the game.

Firstly, we will write code to remember if the keys have been pressed. In fact from the most important keys, we will remember if they are being pressed right now. This will allow us to hold down keys to repeat an action in the game like moving or firing.

Example: The player key listener

```
1 public class PlayerListener implements KeyListener {
2     private static boolean left;
3     private static boolean right;
4     private static boolean fire;
5
6     public void keyPressed(KeyEvent e) {
7         if (e.getKeyCode() == KeyEvent.VK_LEFT) {
8             left = true;
9         } else if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
10            right = true;
11        } else if (e.getKeyCode() == KeyEvent.VK_SPACE) {
12            fire = true;
13        }
14    }
15
16    public void keyReleased(KeyEvent e) {
17        if (e.getKeyCode() == KeyEvent.VK_LEFT) {
18            left = false;
19        } else if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
20            right = false;
21        } else if (e.getKeyCode() == KeyEvent.VK_SPACE) {
22            fire = false;
23        }
24    }
25
26    public static boolean isLeft() {
27        return left;
28    }
29
30    public static boolean isRight() {
31        return right;
32    }
33
34    public static boolean isFire() {
35        return fire;
36    }
37
38    public void keyTyped(KeyEvent e) {}
39 }
```

The important points to note here are that the boolean variables are all static/class variables, as are the methods for checking their values. The way this works is that when a key is pressed down the value is set to true, while the game continues this can be checked by the ship and used to move the ship or fire the weapon.

Example: Using the key listener in the player class

```

1 public void move() {
2     if (PlayerListener.isLeft()) {
3         move(-2);
4     } else if (PlayerListener.isRight()) {
5         move(2);
6     }
7 }

```

This code gives an example of how the key listener might be used to move the ship. Here it is assumed that there is a private method called `move`, which takes a single `int` parameter denoting how much to move and in which direction. This method then checks if the new position is allowed in the game (still inside the screen). The same strategy can be used for determining when the ship should fire.

NOTE, as of yet we have not created a part of the game to manage moving the objects and playing the game so the code will still simply show a static image. This will be done later.

10.5 Practical Exercises

1. Write a class named **Face**. This class should extend the abstract class **Abstract2DDrawing**. Using the methods of the **Graphics** class, draw a face with a height of 150 and a width of 100. The top left corner of the face should be at position `x, y`. These values should be passed as parameters to the constructor. The face should have:

- Eyes
- A nose
- A mouth
- Pupils

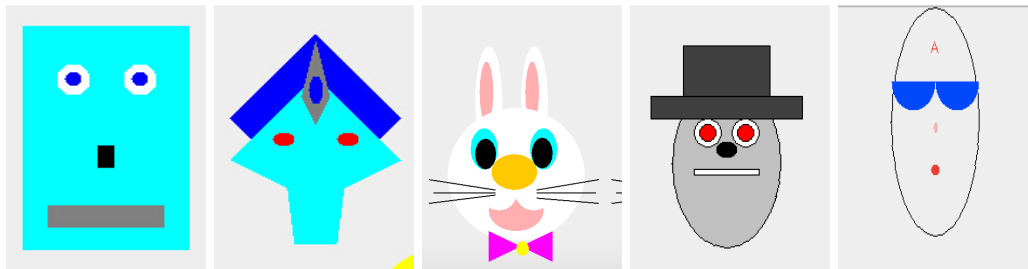


Figure 10.7: Examples of faces drawn by previous students

Test your drawing by creating two face objects in different locations on the screen using the **Picture** class. Remember all coordinates in your code will need to be relative to `x` and `y`.

2. Write a class named **HatFace**. This class should extend the class **Face**. In this class you should override the `draw` method from the face class so that after drawing the face (by calling the method in the superclass) you then draw a hat on the top of the face. This is similar to the **RoofBuilding** example in the chapter.
3. Write two classes that extend the **Abstract2DDrawing** class. Each class should draw a different object. Each object should have at least 5 shapes in it. Examples of the type of objects would be: Car, Truck, Ship, sailing boat, house, flower, tree, train, rocket, person, animal etc.

4. Draw a picture using the shapes that you have created in the previous question. There must be at least one of every object you have created and at least 6 objects in the picture. **Try to make it look nice!!!**

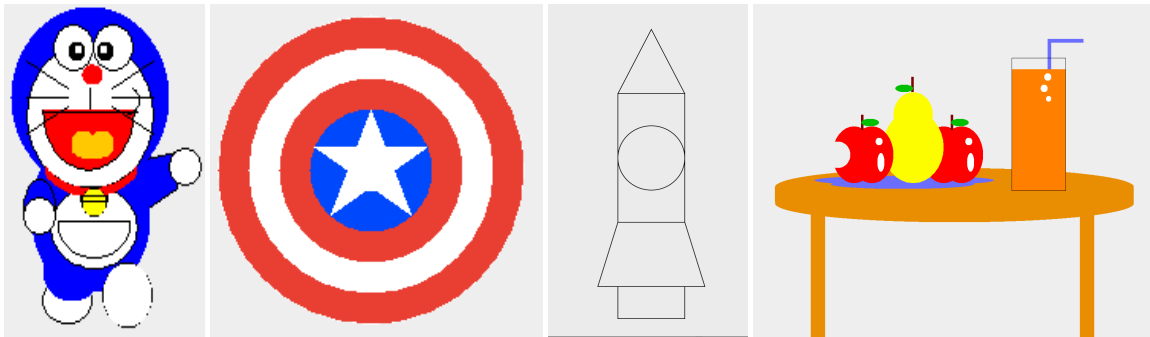


Figure 10.8: Examples of shapes drawn by previous students

5. Define a class named **Maze**. This class should extend the **Abstract2DDrawing** class. Using lines, the class should draw a maze a simple maze using lines that is height and width 300. Figure 10.9 shows a basic design of a maze I generated. This maze required 41 separate lines to complete. Your maze should be of your own design and have between 25 and 50 lines in it.
6. Define a class named **Player**, this class should extend the **Abstract2DDrawing** class. This class should fill a circle of height and width 10 which is centred at the location (x, y).
7. Define a method named **setXY**, this method should take two integers as parameters and use these to set the value of x and y in the player object.
8. Define a class named **MazePicture**, this class should extend the **JComponent** class. Add instance variables to represent a maze and a player.
9. Override the **paintComponent** method (just like in the **Picture** class), and draw both the maze and the player.
10. Define a method named **movePlayer** in the **MazePicture** class. This method should take two integer parameters and use the move method in the player object to change its location. Remember to call the **repaint** method at the end of this method.

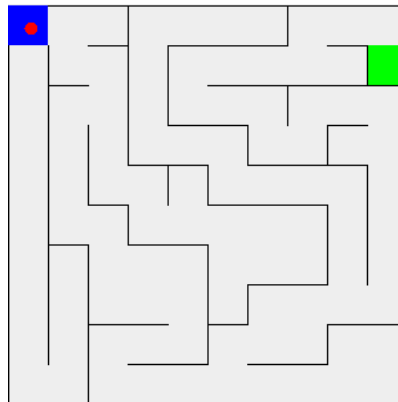


Figure 10.9: A basic maze design

11. Define a class named **PlayerKeyControl**, this class should implement the **KeyListener** interface. This class should have a constructor that takes a **MazePicture** object as a parameter and stores it as an instance variable.
12. Implement the **keyPressed** method so that whenever the user presses the up key the player is moved up by 2 and so on.
13. In the **Window** class construct a **PlayerKeyControl** object and add it to the window as a key listener.
14. Add a method to the **Maze** class called **hitWall**. This method should return true if the player is located within 5 pixels of any of the walls.
This is a difficult task to achieve, in order to simplify the process it is easier to represent each of the lines using the class **Line2D.Double**. This class contains a method named **ptSegDist** which takes two integers as parameters and returns the distance from this line segment. If all lines are stored in an array, checking can be done easily in a loop. Each of the lines can then be drawn using a loop from the array with the following code `g.drawLine((int)lines[i].getX1(), (int)lines[i].getY1(), (int)lines[i].getX2(), (int)lines[i].getY2());` (assuming the array is named **lines**).
15. Change the code in the **movePlayer** method so that after the player has been moved, the **hitWall** method is called using the new x and y location. If the player has hit the wall, the x and y location of the player should be set back to its starting location.
16. Define a method named **complete** in the **Maze** class. This method should take two integers as parameters and return true if the values represent a location within the finishing zone of the maze.
17. Change the code in the **movePlayer** method so that after the player has been moved, the **complete** method is called using the new x and y location. If the player has completed the maze, a message dialog box should appear with the message "Congratulations" and the program should exit (call the code `System.exit(0);`)
18. Define a class named **PlayerMouseControl**, this class should extend the class **MouseInputAdapter**. This will allow the class to be registered as both a mouse listener and a mouse motion listener. This class should have an instance variable that holds a reference to the **MazePicture** object.
19. Define two integer instance variables to hold the last known position of the mouse in the x and y axis in the **PlayerMouseControl** class. Override the **mouseMoved** method. In this method you should calculate how much the mouse has moved by and move the player by that amount.
20. Modify the **movePlayer** and **draw** methods in the **MazePicture** class. In the **draw** method, a boolean variable should be checked before the player is drawn. In the **movePlayer**, the boolean value should be set to false whenever the player hits the wall.
21. Define a method named **resetPlayer** in the **MazePicture** class. This method should move the player back to the starting location and set it to be visible again.
22. Change the **keyPressed** method in the **PlayerKeyControl** class. This method should call the **resetPlayer** method for the maze picture object whenever the "r" button is pressed.
23. Override the **mouseClicked** method in the **PlayerMouseControl** class. This method should call the **resetPlayer** method for the maze picture object whenever the mouse is clicked.

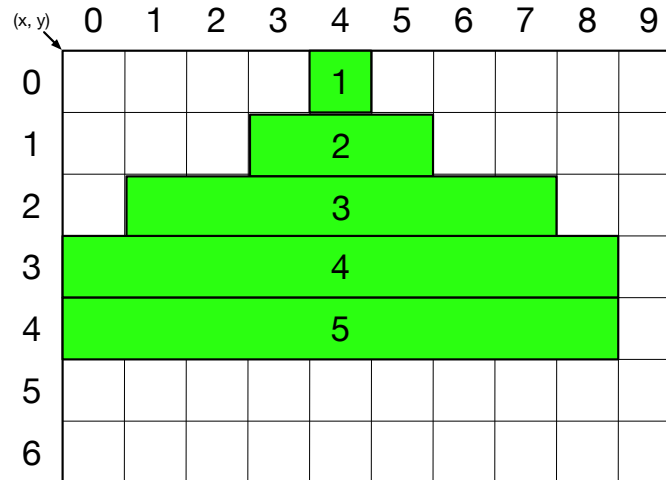


Figure 10.10: The shape of the Players ship

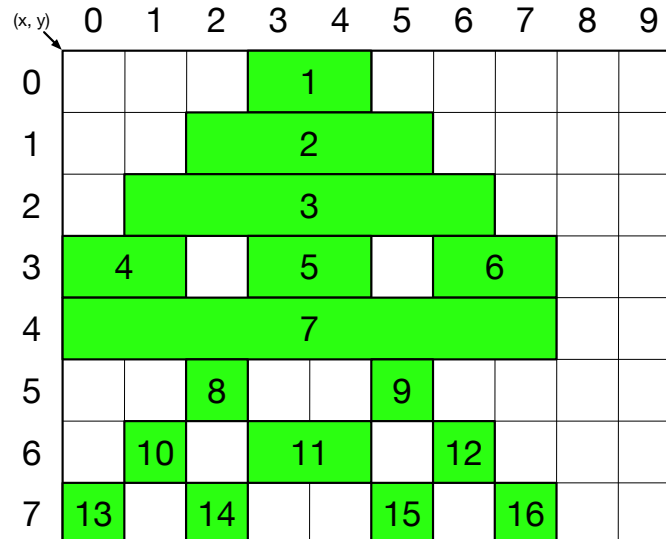


Figure 10.11: The shape of the Type B enemy Ship

10.6 Game Completion Tasks

1. Add an array of `Drawable` objects to the `Game` class, as well as an integer for counting how many objects are in the array.
2. For every object that implements the `Drawable` interface ensure that it is added to the array when it is constructed.
3. Change the `getDrawable` method so it returns a copy of this array.
4. Figure 10.10 shows the shape of the players ship. Complete the `getShape` method in the `Player` class to represent the players ship correctly on the screen.
5. Figure 10.11 shows the shape of the type B enemy ship. Fill in the correct values for the coefficients in the `AlienType` enumerated type.

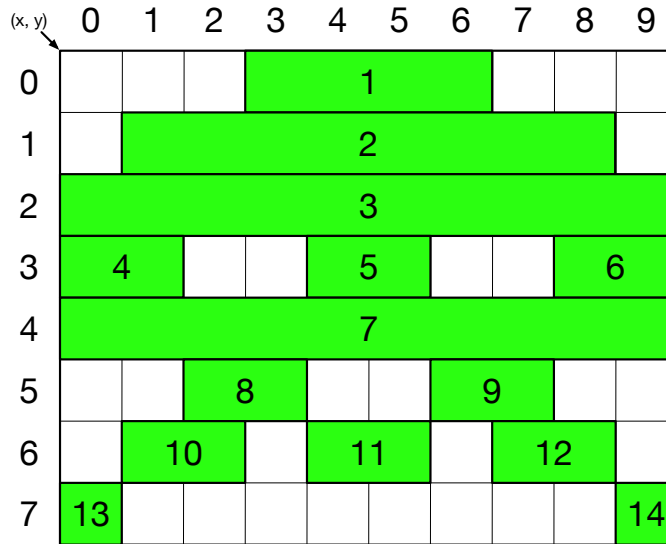


Figure 10.12: The shape of the Type C enemy Ship

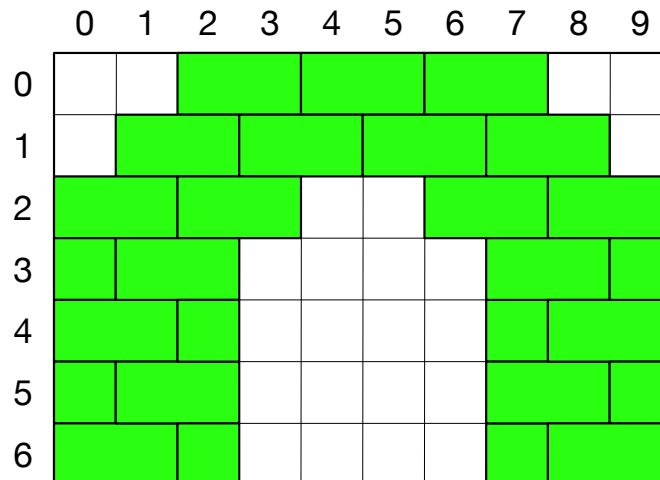


Figure 10.13: The shape of defensive bunkers

6. Figure 10.12 shows the shape of the type C enemy ship. Fill in the correct values for the coefficients in the `AlienType` enumerated type.
7. Complete the `addBricks` method in the `Bunker` class based on the shape described in figure 10.13.
8. Add code to the `PlayerListener` class so that we can also check for the user having pressed N, H or X. The check should look for either uppercase or lowercase and should provide a class method for other parts of the game to check if the key has been pressed. Note, we do not need to know if it is currently being pressed only if it has been pressed (this should be reset after we check). Additionally, as these are characters you should implement this in the `keyTyped` method.

Controlling Layout

This section is some details that are useful for creating the game but are not essential things to learn about object-oriented programming. For the purposes of the course, none of this stuff will be on the exam.

If we want to make our game look better, we will want it to show more than just the game screen. We might want to show a menu, or the previous high scores or some other screen like settings. So far we have only focused on how to get a single drawable component displayed on the screen. In this section we will look at getting multiple different screens prepared and showing the correct screen at the correct time.

To make this work we are going to create a `ScreenController` class to control which of the available screen is shown and when. We are also going to create a menu screen that will be shown when the game first opens. This menu screen will be created the same way as the game screen was created, by extending the `JComponent` class. In the classes `paintComponent` method we will paint the background in black and add the menu options by drawing text in green. Additionally, to make the menu more complex, we will add some unmoving parts from the game.

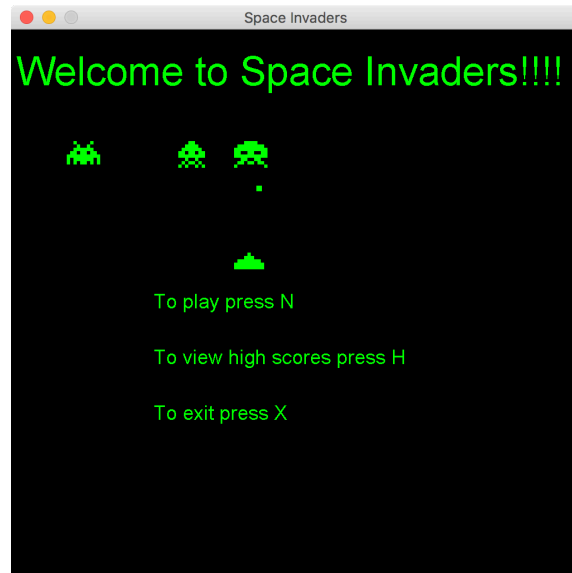
Example: Code for showing the menu on the screen

```

1 public class MenuScreen extends JComponent {
2     private static final long serialVersionUID = 8486434029426170925L;
3     private Drawable[] images;
4
5     public MenuScreen() {
6         images = new Drawable[5];
7         images[0] = new Enemy(50, 100, 5, AlienType.A);
8         images[1] = new Enemy(150, 100, 5, AlienType.B);
9         images[2] = new Enemy(200, 100, 5, AlienType.C);
10        images[3] = new Bullet(220, 140, true, 0);
11        images[4] = new Player(200, 200, 5);
12    }
13
14    public void paintComponent(Graphics g) {
15        g.setColor(Color.black);
16        g.fillRect(0, 0, Game.SCREEN_WIDTH, Game.SCREEN_HEIGHT);
17        g.setColor(Color.green);
18
19        g.setFont(new Font("Arial", Font.PLAIN, 36));
20        g.drawString("Welcome to Space Invaders!!!!", 5, 50);
21
22        g.setFont(new Font("Arial", Font.PLAIN, 18));
23        g.drawString("To play press N", Game.SCREEN_WIDTH / 4, 250);
24        g.drawString("To view high scores press H", Game.SCREEN_WIDTH / 4,
25        300);
26        g.drawString("To exit press X", Game.SCREEN_WIDTH / 4, 350);
27
28        for (int i = 0; i < images.length; i++) {
29            Rectangle[] r = images[i].getShape();
30            for (int j = 0; j < r.length; j++) {
31                g.fillRect(r[j].x, r[j].y, r[j].width, r[j].height);
32            }
33        }
34    }

```

Lines 6 to 11 are responsible for the position of the three alien ships and the player and bullet. These are only drawn on the screen inside the loop on line 30. The rest of the `paintComponent`



method draws the background and the text. Figure ?? shows the result of this code.

Now we have two classes that extend `JComponent` and can be shown on the screen, we need some way to control how and when these are shown. To do this we will use the `CardLayout` layout manager for the `JPanel` class. We are going to do this inside a separate class called the `ScreenController`. In this class we will create a `JPanel` with the card layout, add each of our screens with a different name associated. Use the associated name to choose which screen should be shown.

Example: The screen controller class

```

1 public class ScreenController {
2     private Game game;
3     private JPanel panel;
4     private CardLayout layout;
5
6     public ScreenController(JFrame w) {
7         game = new Game();
8         layout = new CardLayout();
9         panel = new JPanel(layout);
10        panel.add(new GameScreen(game), "Game");
11        panel.add(new MenuScreen(), "Menu");
12        w.getContentPane().add(panel);
13        layout.show(panel, "Menu");
14    }
15
16    public void run() {
17        // run the game and choose the correct screen to show
18    }
19 }

```

In the constructor of the class, we first create a `Game` object, this is what will be used to play the game. Next we create `layout` and `JPanel` objects before adding our game screen and menu screen to the `JPanel`. We then add the `JPanel` to the `JFrame` (we created this in the `Window` class). Finally, we tell the layout manager which of the screens it should show. There is also a `run` method which we will later use to change screens as well as drive the action in our game.

Example: The modified Window class

```
1 public class Window {  
2     public static void main(String[] args) {  
3         JFrame window = new JFrame();  
4         window.setSize(Game.SCREEN_WIDTH, Game.SCREEN_HEIGHT);  
5         window.setResizable(false);  
6         window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
7         window.setTitle("Space Invaders");  
8         window.setLocationRelativeTo(null);  
9         ScreenController controller = new ScreenController(window);  
10        window.addKeyListener(new PlayerListener());  
11        window.setVisible(true);  
12        controller.run();  
13    }  
14 }
```

Here we can see the minor changes that need to be made to the Window class to make this new code work. Later we will have to connect the listener code to the run method so that we can control the execution of the game.

Chapter 11

Testing

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the purpose of testing
- ☐ be able to describe the difference between traditional testing and unit testing
- ☐ be able to create test classes using JUnit
- ☐ be able to design and implement basic tests using JUnit
- ☐ be familiar with the use of tests to find errors in code

11.1 Testing

Testing is an important part of programming and software engineering. Described simply, testing is making sure that the program you have written works correctly. When talking about professional software engineering, testing is evaluation of the software against requirements gathered from users and system specifications. There are two primary methods of testing, **traditional testing** and **unit testing**.

11.2 Traditional Testing

Traditional testing usually involves running the whole program and seeing the result while unit testing tests each component of the system individually. Without testing our programs in some way, it is very difficult to know if they are correct or not. Even when some testing is done, if it is not very thorough there may be mistakes that we have not found.

When testing the entire program, it can be very difficult to test every part of the system. Lets take a simple example of a program that reads some coordinates from the user and tells prints a message telling the user if they are inside a rectangle or not. In this situation, we would need to test the program in both situations (inside the rectangle and outside) to see if the output is correct. In more complicated programs where there are many situations like this, and testing each one of them would be very difficult. Additionally, as we are testing the whole program when we find that something is incorrect, it can be difficult to know where this error actually is.

There are a number of different ways that traditional testing can be done, the most commonly used are;

- Adding print statements
- Using a debugger
- Testing scripts

Print Statements

This is the most common and simple form of testing that is in use. The basic idea is that we print the value of important variables at particular points in execution of the program. Take a very simple example of adding two numbers together, to make sure it was operating correctly we would add a print statement afterwards that prints the value of both variables and the result. We can then verify that the result was correct.

Example: Basic example of added print statements

```
1 int result = x + y;  
2 System.out.println(result + " = " + x + " + " + y);
```

The main problem with testing in this way is that only the original programmer understands meaning and expected output of the print statements. Another programmer might not recognise if the values of `x` or `y` were incorrect. Additionally, this type of testing adds a lot of output and can be difficult to understand in complex applications (particularly if there are multiple threads). Finally, this test code has to be removed when the software is completed, as the user should not see this information. If any further problems are identified, we can have to add new code for testing to find the source of the problem.

Debuggers

A debugger is a special piece of software that can be used to execute a program. Debuggers allow the execution to be paused at important locations and generally allow the user to view the values of important variables. Most will allow the user to go through code one statement at a time if necessary and watch the value of variables as they change.

Debuggers are very powerful when it comes to finding errors in code, but they can be difficult to learn how to use well. Additionally, finding errors using a debugger requires knowledge of an entire system as we need to know how the pieces of the application work together.

Testing scripts

A testing script is automated script that will execute a program and provide it with the required input. The output of the program is then recorded and compared against the output that was expected. This means that in order to use a script to test the correctness of a program, we need to already know exactly what output is expected. If the output is different from the expected output by even a single character it may be viewed as a fail. This is how some of your assignments are graded.

This means that when we change the program we also need to change the expected output. This can be a large amount of work.

Knowing what should happen

All traditional testing methods have the same requirement, we need to know what the output should be before the program can be tested. If we do not know what the result should be, how would we know if it was correct or not. This is possibly the biggest difficulty when it comes to testing.

11.3 Unit Testing

The best alternative to traditional testing is **unit testing**. The idea behind unit testing is that we test each component of the system individually. For example, if we are writing a method that finds the largest value in an array, we would write test for just this method rather than the entire program that it is used in.

We would write tests for all of the components of the system that we write. As a consequence of writing tests this way, we find errors earlier because they can be found as soon as an individual component is completed. If we were testing an entire program, we would have to wait until it was at least partially completed before we could start testing.

Additionally, because we write tests for each component, it is easier to find and fix errors. For example, if the array method we discussed earlier fails its test we know exactly where to look to find the error. If we were testing the whole program, this error might not be visible or cause a problem until later in the execution of the program.

The basic principles of unit testing are that all tests should be describe by all of the following ideas;

- Isolatable
A test on a component should be able to run without the rest of the program being completed
- Repeatable
A test should be able to be repeated
- Automatable
The tests should be able to be run automatically, hourly or daily or every time you compile
- Easy to write

Benefits of unit testing

There are a lot of benefits to using unit testing in the development of large programs. Generally it is expected that you can debug faster, develop faster, design better and reduce future costs.

Faster debugging

Because each unit test covers a smaller amount of code, when a test is failed the error can only be in a small part of the program. This makes it much easier to find the error and solve it, thus making us faster at debugging.

Faster development

Because we spend less time debugging, we spend less time overall in development. Additionally when we add code to the program, we can be sure that it is working correctly.

Better design

When we are unit testing, it causes developers to focus more on what a class should do. Testing might highlight classes that are not very cohesive and cause the developer to redesign them in a better way. This makes our programs easier to maintain and debug in the future.

Reduce future costs

While unit testing takes time at the beginning of a project, we save time over the long term. In software development, time means wages that are paid to developers and anything that can reduce the amount of time it takes to develop a program saves money for the company.

11.4 JUnit

JUnit is a library for Java that makes it easy to write and execute unit tests. JUnit is the most commonly used unit test library for Java. This is because with JUnit, it is easy to use, tests can be written easily and tests can be automated. More information can be found at <http://www.junit.org>.

JUnit is very useful, but it is not perfect. JUnit (and unit testing in general) is not very good for testing graphical applications because user interaction is required. JUnit does not compile good reports for very large projects. JUnit takes some time to set up. It is difficult to test parts of the program that are not implemented in Java.

The main components of JUnit are test classes. These are separate classes used only for testing that are not included in the final program. Test classes can be stored and executed from the same location as the code they are testing, but there are usually placed within a separate folder specifically for test code. Within test classes, there can be many test methods. Each of these test methods can be executed individually or all together as a group.

Assertions

Testing in JUnit is done by adding declarative statements called assertions. These statements will state that some particular property will be true at the time they are executed. If any of the statements within a test are not true, then the entire test has failed. Here is a list of the most commonly used declarative statements;

- **assertEquals**
- **assertTrue**
- **assertFalse**
- **fail**
- **assertSame**

assertEquals

The assertEquals declaration takes two parameters of the same type and fails only if the values are not equal. Typically, we would use this to test if a variable has the value we expect it to have at a give time. For example, `assertEquals(x, 10);` will fail if the value of x is not 10.

assertTrue

The assertTrue declaration takes a single parameter that must be a boolean expression and fails if the result of the expression is not true. This allows a more varied type of check, where we can be sure that some property is true. For example, `assertTrue(x <= 100);` only fails if the value of x is greater than 100 but will allow many different values for x.

assertFalse

The assertFalse declaration is basically the reverse of the assertTrue, this will fail if the result of the expression is not false.

fail

The fail declaration takes no parameters and always causes a test to fail. This might seem like it is not very useful, but it is. Typically, this declaration would be added to a part of the code that should never be executed. For example, if we have a large series of if else statements, we could add the fail declaration so that we know when none of the if statements matched.

assertSame

The assertSame declaration takes two objects as parameters and checks to see if in fact they are two references to the same object in memory. This is similar to assertEquals, but even if two objects are equal (contain the same data) they may be separate in memory.

JUnit Setup

Performing unit tests requires a few steps, they first need to be set up, then they need to be written and finally they need to be executed.

Setting up unit tests

Because we will be doing most of our programming in eclipse, we will look at the steps required to set up unit tests in eclipse.

The following step is not necessary, but is a nice way to keep program code and test code separate. This helps keep the project structure looking clean. Additionally, it allows us to use the same package structure for our tests as we use for our code. For example, both the class `Ship` and its test class can be declared as the same package, but stored in separate locations. In eclipse, we can add a new source folder to the project by right-clicking on our project and selecting **New - Source Folder**. Traditionally, this folder would be named `test` but we can give it any name we want.

The next step is to actually create our test class, traditionally this class will have the same name as the class it is testing with `Test` added to the end. For example, if we are testing the `Ship` class, we would create a class called `ShipTest`. This can be done easily in eclipse by selecting the class we want to test and right-clicking in the package explorer. From the menu select **New - JUnit Test Case**, this will open a window for creating the test class, the only item that needs to be changed is the source folder (only if you are keeping source and tests separate), which should be change from `src` to `test`. The first time this is done for a project, eclipse will ask you if you want to automatically include the JUnit library in the project, click **Yes** or **OK**.

Annotations

After completing the above step there should be a test class created, which already contains a single test. Tests in JUnit can be given any name, but in order for the system to be able to know which methods are tests we need to add a piece of information to the declaration. This piece of information is known as an **annotation**. There are many different uses for annotations and you can study them in your own time. We will be looking at the very basics required to write unit tests.

Annotations are used to put a note on a method, class or variable. Annotations have no effect on the execution of the code in Java, they are most commonly used by other systems to perform some function. Annotations are written using the '@' symbol followed by a name, for example `@Test` or `@Override`.

There are a number of different annotations used in JUnit, but we will only focus on the most commonly used.

- **@Test** - When this annotation is placed before a method, it tells JUnit that the following method is a test that should be executed when the class is being tested.
- **@Before** - When this annotation is placed before a method it tells JUnit that the method is used to perform set up before any of the tests should be executed.

Example: Empty method annotated as a test

```
1 @Test
2 public void testSomething() {
3     // do some testing
4 }
```

Writing unit tests

Before we look at the code we need to decide what exactly we are going to test and what the results should be. Lets start with the `takeDamage` method from our `Ship` class in chapter 5. When we discussed this method we stated that the following things should happen;

- The health of the ship should be reduced by the power of the bullet parameter
- The parameter should not be negative
- The minimum value for health is 0

To properly test this method we need to write a test case for each of these outcomes. Lets start with the first one. The test case should be given a descriptive name such as `testTakeDamageReduction`, which means it is testing the reduction in the take damage method. The first step is that we need to create a ship object to test, because the class is abstract, we will actually need to create a player object for the test. Next we call the `takeDamage` method and add a test to make sure that the value of health is what we expect.

Example: Testing the reduction in health

```
1 @Test
2 public void testTakeDamageReduction() {
3     Ship shp = new Player(50, 400, 0, 20);
4     Bullet b = new Bullet(50,400, false, 5);
5     shp.takeDamage(b);
6     assertEquals(15, shp.getHealth());
7 }
```

We also need to make sure that a negative parameter should not be used to increase the ships health. A single test will cover this, where we pass a negative value to the method, check the return value and then make sure that the health is not changed.

Example: Testing to make sure that a negative parameter is ignored

```
1 @Test
2 public void testTakeDamageNegativeParameter() {
3     Ship shp = new Player(50, 400, 0, 20);
4     Bullet b = new Bullet(50,400, false, -5);
5     shp.takeDamage(b);
6     assertEquals(20, shp.getHealth());
7 }
```

Lastly, we need to check to make sure that the value of health cannot drop below 0. Again this can be covered by a single test. To test this we only need to pass a parameter that is bigger than the ships health and then test to make sure the value of health is 0 after.

Example: Testing to make sure the value of health cannot go below zero

```
1 @Test
2 public void testTakeDamageHealthZero() {
3     Ship shp = new Player(50, 400, 0, 20);
4     Bullet b = new Bullet(50,400, false, 25);
5     shp.takeDamage(b);
6     assertEquals(0, shp.getHealth());
7 }
```

11.5 Testing With IO

Some of the code we want to test will require user input and will output the results to the screen. Testing this type of code using unit tests can be difficult. However, there are some techniques that allow us to replace the user input with text we have written and save the output into a String for checking. These processes must be carried out before we execute the code that we want to test.

Replacing `System.in`

If our code is designed to read some information from the user through `System.in`, we can tell Java to replace that source of text with another. First we have to create our source of text, this is done by typing the text we want and then placing that inside an object that can be used to replace `System.in`. The following code creates a suitable replacement `ByteArrayInputStream in = new ByteArrayInputStream("text we want".getBytes());`. We can then tell Java to use this instead of waiting for the user to type in information with the code `System.setIn(in);`. This tells Java to use the object named `in` that we created in the previous statement instead of `System.in`.

Saving `System.out` to a string

In the same way, we can save anything that the user prints to the stream by replacing `System.out`. This is a little more complicated, and requires three steps. First we create an output stream using the code `ByteArrayOutputStream baos = new ByteArrayOutputStream();`. We then use this to create a `PrintStream` object (the same type as `System.out`) using the code `PrintStream ps = new PrintStream(baos);`. This takes the object created in the first statement to create the `printstream`. Lastly we tell Java to replace `System.out` using the code `System.setOut(ps);`.

Once completed, any time the code uses `System.out.println` or `System.out.print`, the information will be put into the object `baos` instead. After the code we want to test has been executed, we can get the information printed in the form of a String using the code `String out = baos.toString();`. The String `out` now contains all of the information that would have been printed to the screen. We can use this string to check if the result is what we expect.

11.6 Code Coverage

When writing test code for our application, it can be difficult to know how well the code is actually tested. When we have written a large amount of code, it can be difficult to know if we have tested it all. Code coverage is a measurement of the code that has been tested by the unit tests we have written. This is generally measured by tools and expressed as a percentage of the total code. There are a number of coverage criteria, the main ones being:

- Function coverage - Has each function in the application been called by the test code?
- Statement coverage - Has each statement in the application been called by the test code?
- Branch coverage - Has each branch of each control structure (such as in if and case statements) been executed by the test code?
- Condition coverage - Has each Boolean sub-expression evaluated both to true and false?

Example: A simple method

```

1 public int foo (int x, int y) {
2     int z = 0;
3     if ((x>0) && (y>0)) {
4         z = x;
5     }
6     return z;
7 }

```

Lets look at these different types of coverage in terms of the simple method above. Assume that this method is part of a larger application for which we are writing some tests.

- If during the execution of the tests the method 'foo' was called at least once, then function coverage for this function is satisfied.
- If during the execution of the tests every statement was executed, then statement coverage is satisfied. For example if we call the method with the following parameters, `foo(1,1)` every line of code will be executed.
- If during the execution of the tests every possibility for each if statement is executed (true and false), then branch coverage is satisfied. Tests calling `foo(1,1)` and `foo(0,1)` will satisfy this because in the first case the condition of the if statement is true and in the second it is false.
- If during the execution of the tests every possibility for each part of the condition in an if statement is executed (true and false), then condition coverage is satisfied. To achieve this we need to execute the tests calling `foo(1,1)`, `foo(1,0)` and `foo(0,0)`.
 - In the first case both parts of the condition `(x>0)` and `(y>0)` are evaluated as true
 - In the second case, the first part is evaluated as true and the second as false
 - In the third case, the first part is evaluated as false and the second is not evaluated.

Fault injection

In order to fully test our code, we will sometimes need to introduce an error so that our error handling code is executed. It can be difficult to test code that is only supposed to be executed when something has gone wrong. For example we might not test all of the possible things that can go wrong with our code.

11.7 How Much Code Coverage?

Testivus On Test Coverage¹

Early one morning, a programmer asked the great master:

“I am ready to write some unit tests. What code coverage should I aim for?”

The great master replied:

“Dont worry about coverage, just write some good tests. ”

The programmer smiled, bowed, and left. Later that day, a second programmer asked the same question. The great master pointed at a pot of boiling water and said:

“How many grains of rice should put in that pot”

The programmer, looking puzzled, replied:

“How can I possibly tell you? It depends on how many people you need to feed, how hungry they are, what other food you are serving, how much rice you have available, and so on.”

¹Taken from <http://www.artima.com/forums/flat.jsp?forum=106&thread=204677>

“Exactly,” said the great master.

The second programmer smiled, bowed, and left. Toward the end of the day, a third programmer came and asked the same question about code coverage.

“Eighty percent and no less!”

Replied the master in a stern voice, pounding his fist on the table. The third programmer smiled, bowed, and left. After this last reply, a young apprentice approached the great master:

“Great master, today I overheard you answer the same question about code coverage with three different answers. Why?”

The great master stood up from his chair:

“Come get some fresh tea with me and lets talk about it.”

After they filled their cups with smoking hot green tea, the great master began to answer:

“The first programmer is new and just getting started with testing. Right now he has a lot of code and no tests. He has a long way to go; focusing on code coverage at this time would be depressing and quite useless. Hes better off just getting used to writing and running some tests. He can worry about coverage later.”

“The second programmer, on the other hand, is quite experience both at programming and testing. When I replied by asking her how many grains of rice I should put in a pot, I helped her realize that the amount of testing necessary depends on a number of factors, and she knows those factors better than I do its her code after all. There is no single, simple, answer, and shes smart enough to handle the truth and work with that.”

“I see,” said the young apprentice, “but if there is no single simple answer, then why did you answer the third programmer ‘Eighty percent and no less’?”

The great master laughed so hard and loud that his belly, evidence that he drank more than just green tea, flopped up and down.

“The third programmer wants only simple answers even when there are no simple answers and then does not follow them anyway.”

The young apprentice and the grizzled great master finished drinking their tea in contemplative silence.

What does this mean?

This story tries to give you the idea that you should try and not focus on the coverage percentage, or try to find an arbitrary number for it, but instead focus on having as much logic and functionality tested as is humanly possible. Different applications will have different percentages of code that should be tested. It is quite reasonable to have a, say, 50% coverage rate if only because only 50% of the code contains logic that can be tested, and the other 50% happens to be simple objects or things that are handled by a framework.

11.8 Practical Exercises

For these exercises, you will be writing tests to find the errors in the a class called the `StringCalculator`. This class has methods to perform all of the basic operations (addition, subtraction, multiplication, division and modulus), however it is different in a couple of ways. Firstly, the object performs these operations on strings instead of integers and secondly, the user chooses what base the numbers are represented in and lastly, the object only works for positive integer values.

Base

The class allows the user to choose the base that the calculator operates in, for normal maths as we are use to you would choose the base 10 (decimal), for hexadecimal you would choose base 15, for octal base 8 and binary, base 2. The class should function correctly for any base between 2 and 16.

Hexadecimal

If the calculator is using base 16 (hexadecimal) then the acceptable characters are 0 - 9 and a - f (uppercase and lowercase).

Octal

If the calculator is using base 8 (octal) then the acceptable characters are 0 - 7.

Binary

If the calculator is using base 2 (binary) then the acceptable characters are 0 and 1.

Incorrectly formatted numbers

The object should reject any number where the correct digits are not used. For example if the object is constructed to operate for binary, then adding a number containing a 2 should result in a return message of "Incorrect Number Format"

Design Specifications

1. **Constructor**

The constructor takes a single integer parameter between 2 and 16 which determines the base of the numbers being calculated.

2. **convertToInt**

This method takes a single String as a parameter and returns the value that this string represents in the base of the object.

- If the number is negative (any value) then the method should return the value -1
- If the number is in any base above 10, then the method should work for the appropriate uppercase and lowercase characters, i.e. 'a' - 'f' for base 16

3. **convertToString**

This method takes a single String as a parameter and returns the value of this int represented as a string.

- If the number is negative, then the method should return the message "Negative Value"
- If the number is in any base above 10, then the returned string should represent this using digits and uppercase characters in the appropriate range

4. **add** - This method should take two strings as parameters and return a string containing the two values added together.

5. **subtract** - This method should take two strings as parameters and return a string containing the result of the second value subtracted from the first.

6. **multiply** - This method should take two strings as parameters and return a string containing the product of these values.

7. **divide** - This method should take two strings as parameters and return a string containing the number of times that the second number can be subtracted from the first value (integer division).

8. **modulus** - This method should take two strings as parameters and return a string containing the remaining value if first number is divided by the second number.

Tasks

This class contains a number of errors, write unit tests to find these errors and then correct them. There should be at least one unit test for each of the methods specified above, and realistically there will be multiple unit tests for some. Once you have written tests for a method, do not write more tests until you have fixed any errors that you have found. There are at least 7 errors in this code (maybe more).

Coverage

Using a code coverage tool, such as the eclipse plugin available at <http://eclemma.org/>, ensure that you have achieved 100% coverage in your tests. But note that if your tests are not well written, it is possible to have 100% coverage and still not find an error. Download here - `StringCalculator.java`

Example: The StringCalculator class

```

1 package chapter10;
2
3 public class StringCalculator {
4     private int base;
5     public StringCalculator(int b){
6         base = b;
7     }
8
9     public int convertToInt(String number){
10        int value = 0;
11        for (int i = 0; i < number.length(); i++) {
12            value *= base;
13            int digit;
14            if(base > 10){
15                if(Character.isDigit(number.charAt(i))){
16                    digit = number.charAt(i) - '0';
17                } else {
18                    if(Character.isUpperCase(number.charAt(i))){
19                        digit = 10 + number.charAt(i) - 'A';
20                    } else {
21                        digit = 10 + number.charAt(i) - 'a';
22                    }
23                }
24            } else {
25                digit = number.charAt(i);
26            }
27            value += digit;
28        }
29        return 0;
30    }
31
32    public String convertToString(int i){
33        String value = "";
34        if(i == 0){
35            value = "0";
36        }
37        while(i > 0){
38            int v = (i % base);
39            char c;
40            if(v > 9){
41                c = (char) (v + 'a' - 10);

```

```
42     } else {
43         c = (char) (v);
44     }
45     i/=base;
46     value = c + value;
47 }
48
49 return value;
50 }
51
52 public String add(String n1, String n2){
53     int v1 = convertToInt(n1);
54     int v2 = convertToInt(n1);
55     int result = v1 + v2;
56     String r = convertToString(result);
57     return r;
58 }
59
60 public String subtract(String n1, String n2){
61     int v1 = convertToInt(n1);
62     int v2 = convertToInt(n2);
63     int result = v2 - v1;
64     String r = convertToString(result);
65     return r;
66 }
67
68 public String multiply(String n1, String n2){
69     int v1 = convertToInt(n1);
70     int v2 = convertToInt(n2);
71     int result = v1 * v2;
72     String r = convertToString(result);
73     return r;
74 }
75
76 public String divide(String n1, String n2){
77     int v1 = convertToInt(n1);
78     int v2 = convertToInt(n2);
79     int result = v1 * v2;
80     String r = convertToString(result);
81     return r;
82 }
83
84 public String modulus(String n1, String n2){
85     int v1 = convertToInt(n1);
86     int v2 = convertToInt(n2);
87     int result = v1 / v2;
88     String r = convertToString(result);
89     return r;
90 }
91 }
```

Chapter 12

Exceptions

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the difference between syntactic, semantic and resource errors
- ☐ understand the concept of the stack and stack traces
- ☐ understand the terminology of error recovery
- ☐ be able to implement error recovery code for in programs

12.1 Errors

Writing programs is not a simple or easy task. Most, if not all, large programs that are written will contain some errors. By this we mean that is some (possibly small) way the program does not do what it is supposed to do. There are many examples of this in the area of game development, titles such as Assassin's Creed: Unity have been know to launch with issues such as characters missing faces, players falling through the map and poor game performance.

There are three main types of errors:

- Syntactic errors
- Semantic errors
- Input/resource errors

These types of errors generally appear in different ways, at different times and for different reasons. In short, syntactic errors appear at compile time, semantic appear at run time because of a programmer error and resource errors happen when there is a problem with a resource (network or file are most common).

Syntactic errors

Programming languages are **formal** languages, this means that there is an exact definition of what is **acceptable** text for a program in a given language. The actual definition is generally only of interest to language designers and are usually specified in BackusNaur Form (BNF). Almost all programmers simply learn the syntax that is acceptable based on examples and experience with the language, however if interested the syntax can be found in BNF-style at <https://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html>.

These specifications are large and complex, a simple example of declaring a variable could require understanding 30 lines of the BNF that are spread throughout the many pages of the document. Whereas a simple example stating that variables are declared in the form `type name;` is much

easier to understand initially and the programmer can later learn about access modifiers and the specifics of what names can be used.

Syntactic errors are not a serious problem, then can be detected automatically whenever we compile our code. The most difficult part about syntactic errors is often understanding the meaning of the error message that is presented. Language developers attempt to make these easy to understand, however there are so many different types of errors that often the message we read will not explicitly state the problem. This issues are generally overcome as the programmer gains in experience and gains a better understanding of the process of compilation.

Semantic errors

Semantics of a program are about its meaning. Programs will always do exactly what the source code specifies. Unfortunately, this may not be what we wanted the program to do. Technically, semantic errors will include mistakes such as incorrect types i.e. trying to multiply an integer by a string, but these errors are picked up by the compiler and are not considered relevant.

An example of a semantic error would be a method that adds two numbers when it was supposed to subtract them or code that attempts to access an index outside of an array. These are the types of semantic errors that we are interested in. The issue that we are dealing with is that of the **correctness** of the program.

Input/resource errors

Resource errors can be the most difficult to solve. A program might function correctly most of the time, but rely on things outside of the control of the programmer. For example, it might require user input in a specific format or it might need to access a file or access a server on the internet. If some requirement is not met, then the program may crash. The main issue we are dealing with in resource errors is the idea of **robustness** of the program.

Robustness in programming is making sure that no matter what order I type information or if a network resource is not available the program will not crash. The program may not be able to complete the operation it was attempting, but the program should be able to attempt the operation again without needing to be restarted.

Errors, failures and exceptions

A semantic coding error might go unnoticed until it causes some problem, such as accessing an array outside its bounds. In such cases, the Java interpreter will raise an **exception**. An exception is a fatal error that causes the Java interpreter to stop and report the error and crashes our program.

To review the idea of exceptions, we will go through an example program that contains a small bug. The code contains a simple class method that takes an array of integers as a parameter and returns the index that contains the largest value. The main method executes this code and outputs the result.

Example: The findMaxIndex method

```

1 public class ArrayTest {
2     public static int findMaxIndex(int[] a) {
3         int index = -1;
4         int max = 0;
5         for (int i = 0; i < a.length; i++) {
6             if (a[i] > max) {
7                 index = i;
8                 max = a[i];
9             }
10        }
11        return index;
12    }
13
14    public static void main(String[] args) {
15        int[] ex = new int[] { 56, 45, 2, 7, 12, 57, 8 };
16        int mi = findMax(ex);
17        System.out.println("Max is " + ex[mi]);
18    }
19 }

```

This code looks correct, and much of the time it will work well. If we execute the associated test code, we get the following output **Max is 57** (correct). However, if we replace the array with `ex = new int[] { -56, -45, -2, -7, -12 };`, something else happens. Instead we get the following output;

```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
    at ArrayTest.main(ArrayTest.java:17)

```

This is an example of the output we will see when an exception has happened. To properly understand the output generated by exceptions we have to have an idea of how the programming language works when any method can call another method. Every exception is represented by a class (we will see this in more detail later) and the first line of the output will tell us exactly which exception has happened. Some exceptions may include some extra information to help solve the problem. In our example the `ArrayIndexOutOfBoundsException` tells us what index the code was trying to access -1.

12.2 Stack and Stack Traces

Whenever we declare a variable in a method, this must be stored somewhere in memory. However, when we have many methods, there will often be many variables with the same name. This is common in many C-like programming languages. The problem is solved by using a stack to store these variables.

When a method is executed, it is given a position on the stack to store information. In our example, when the main method is executed it is given this space and stores the variables `ex` and `mi` there. Whenever another method is called, a new position is created and placed on the stack. The variables in all other positions can not be accessed until this method is finished. In our example, the method `findMax` gets a position that is placed on top of the stack, and it stores the variables `index`, `max`, `i` and a reference to `ex` with the name `a`. While the code is executing it can access and change any of these variables, but no others.

When this method is complete and has returned its result, it no longer needs these variables and the position that was created for the method is removed. This means that the next position is now moved to the top of the stack and the main method has access to its own variables. This process can go for many level deep, where one method calls another and that calls another and

so on. This information is presented to us whenever an exception occurs. The output from our previous example is called a **stack trace**.

When an exception happens, a stack trace gives us the state of the stack at the exact moment that the exception occurred. This gives us details such as the class and method named as well as the line number for each level of the stack. The items in the stack are printed in the order that they are removed (this is how a stack works). This means that the bottom of every exception will always be the `main` method. The line number tells us exactly what code was being executed at the time. In our example, line 17 was being executed. This allows us to quickly identify where the problem was.

Lets look at this in detail by changing the example above so that it returns the max value instead of the index it is stored in.

Example: The findMax method

```

1 public class ArrayTest {
2     public static int findMax(int[] a) {
3         int index = -1;
4         int max = 0;
5         for (int i = 0; i < a.length; i++) {
6             if (a[i] > max) {
7                 index = i;
8                 max = a[i];
9             }
10        }
11        return a[index];
12    }
13
14    public static void main(String[] args) {
15        int[] ex = new int[] {-56, -45, -2, -7, -12};
16        int max = findMax(ex);
17        System.out.println("Max is " + max);
18    }
19 }

```

If we execute this program we get the following output:

```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
    at ArrayTest.findMax(ArrayTest.java:11)
    at ArrayTest.main(ArrayTest.java:16)

```

This tells us that we tried to access the array at index -1 while we were executing line 11 in the method `findMax`, which was called in the `main` method on line 16. This allows us to see exactly where the problem is in our code and fix it.

12.3 Error Recovery

In this section, we will look at how to design our programs to recover from exceptions. Before we do this we should look at some of the terminology used to describe error handling. When Java has a problem with our code, it **throws** an exception. An exception that is thrown can be **caught** at any point in the stack. Catching an exception is how we can recover from exceptions that happen in our program.

In Java we use a complex statement called a **try-catch** to recover from errors. There are two sections in a try-catch statement;

- The try section, where we attempt the code the might cause a problem by throwing an exception

- The catch section, where we describe what should be done when that exception happens

Syntax: The try-catch statement

```
1 try{
2     // attempt the code
3 } catch(Exception e){
4     // what to do when there is a problem
5 }
```

A try-catch statement could be used to solve the problem in our example, where we try and access the array and if something goes wrong return the value 0.

Example: Using try catch to solve the problem

```
1 try{
2     return a[index];
3 } catch(Exception e){
4     return 0;
5 }
```

The above code means that any exception caused by this code is dealt with immediately and the program will not crash. However, this is not a good solution. This problem is caused by a semantic error and rather than catching the error when it happens, we should instead fix the error. The code is trying to access index -1. This means that the value of index is not set at any point in the loop. By looking at the numbers in the array we can see that they are all negative and the value of max starts at 0. This means that none of the values are bigger than max and the max value is never set. Instead of 0 we should set the value of max to be `Integer.MIN_VALUE`.

Robust programs

A program is robust if it recovers from unexpected errors. By ‘unexpected errors’ we mean resource errors beyond the control of the programmer. Generally, a robust program will catch any exceptions when things go wrong and tell the user if necessary and continue executing if possible.

For example, assume we are writing a program to read student data from a file. If the user enters the name the file incorrectly, the the program will not be able to find the data. Without error recovery this would cause the program to crash. If instead we recover from the error by allowing the user to enter the file name again we can continue executing. This is an example of robust behaviour.

Exception objects

In every catch statement there is a variable that contains information about the exception that was thrown. When we define the code `catch (Exception e)`, we can use the object `e` to find or output information about the exception that happened. When an exception is not caught, the stack trace for that exception is printed and execution is halted. However, when we catch the exception we can still display the stack trace and continue execution.

If we have an exception named `e`, then we can call the method `printStackTrace` to have the stack trace of the exception output to the screen e.g. `e.printStackTrace()`. This will give us all of the information about the program when the exception was thrown. This is particularly useful for debugging the error recovery code we write. However, generally this code should be replaced by some sort of error recovery mechanism when writing code commercially.

12.4 Exceptions

There are many different types of exceptions, each defined as a class. The root of the hierarchy is a class called `Throwable`. However, generally we only deal with two of its known direct subclasses `Error` and `Exception`.

Error

The `Error` class represents serious problems that most applications should not catch. They represent situations that should not happen. For this reason, we will not study these in any detail.

Exception

The `Exception` class represents most problems that a programmer would expect to recover from. All of the exceptions that we deal with will be a subclass of `Exception`, although many not directly. Exceptions can be separated into two categories, **checked** exceptions and **unchecked** exceptions.

Unchecked exceptions

An unchecked exception represents an error that could possibly happen but is generally very unlikely. An exception is unchecked if it is a subclass (directly or indirectly) of the class `RuntimeException`. Typical examples of unchecked exceptions are `IndexOutOfBoundsException`, `NullPointerException` or `InputMismatchException`. Programmers are not required to implement error-recovery code when these exceptions are **possible**.

If we think about the three examples above, it is easy to see why we are not required to implement error-recovery when these exceptions are possible.

- An `IndexOutOfBoundsException` is possible every time we use an array. Imagine how complicated our code would be if every time we wanted to access an array we needed to use a try-catch statement.
- A `NullPointerException` is possible every time we use an object (accessing an instance variable or calling a method).
- An `InputMismatchException` is possible any time we try to read information of a type from a `String` (or the user or a file).

If we were required to add try-catch statements to our code in all of the above situations, we would have code that is very difficult to understand, is longer than it needs to be and takes longer to write.

Checked exceptions

A checked exception represents an error that is likely to happen when the code is executed. An exception is checked if it is a subclass of `Exception`, but not a subclass of `RuntimeException`. Programmers are **required** to implement error-recovery code whenever a checked exception is possible.

Typically, this means code that is performing an action, such as file or network access, that is likely to have a problem. If we want to read a file, there are many problems that can happen such as its location has changed or the format is incorrect. If we want to access information on an internet server, there are many problems that can happen such as a problem with our internet connection or a problem with the server connection.

In most applications, error recovery code is only written for checked exceptions. As this is the case we will focus on these in greater detail later.

12.5 Recovering From Errors

Lets look at an example of implementing some error recovery code. The method `parseInt` in the `Integer` class can be used to convert a string containing a number into an integer value. However, when this method is used it may throw a `NumberFormatException`. This exception is an unchecked exception, so we are not required to implement error recovery, but we can implement it if we want to.

Example: Catching an exception in `Integer.parseInt`

```
1 public static void main(String[] args) {  
2     String s = ""; // string with some value  
3     try {  
4         int x = Integer.parseInt(s);  
5     } catch (Exception e) {  
6         System.out.println("Not a valid int");  
7     }  
8 }
```

We put the code that can cause a problem into the try section of the statement, and if there is a problem with the code our program can continue executing. This means that if the string entered does not contain a valid integer, then the message will be printed out and the code can continue executing. However, the code in the try statement stops at the point where the exception happens. Any code following that statement will not be executed, instead the code in the catch section of the statement is executed and then any code following that. When there is no exception, the code executes normally and the code in the catch section is not executed.

This means that whenever we see a try catch statement, the code can either be executed correctly or an exception will be thrown and the catch statement will be executed.

The code in the example will catch any exception that is a subclass of the `Exception` class because this is the type of exception that we have declared. However, if we wanted to be more specific we could replace this with `catch (NumberFormatException e)`. This makes it clearer what exception is expected to happen and be recovered from.

Multiple catch sections

Different types of errors can happen to the same code. In our example `Integer.parseInt(s)`, there are two possible exceptions, the `NumberFormatException` if the string does not contain a number and a `NullPointerException` if the variable `s` is null. It is likely that we will want to recover from these errors in different ways, in this case we can add multiple catch sections to the same try-catch statement. This allows us to specify what we want to happen for each type of error.

Example: A try-catch statement with multiple catch sections

```
1 try {  
2     int x = Integer.parseInt(s);  
3 } catch (NumberFormatException e){  
4     System.out.println("Not a valid int");  
5 } catch (NullPointerException npe){  
6     System.out.println("s has no value");  
7 }
```

Order

When an exception is thrown in a try section, the JVM will look through the list of exceptions that are caught in order until it finds one that matches. This includes any superclass Exceptions that may be declared. For example, if the first catch statement is for `Exception` and the second is for `NumberFormatException`, then the first will always match because `NumberFormatException` is a subclass of `Exception`.

Finally

Optionally, after the catch sections we can also include a **finally** section. This section will be executed both if an exception is thrown and if it is not. These are usually used to manage resources that are used in the try section of the statement. For example, if we are using a network stream, this should be closed after use. Placing this code in a **finally** section allows the code to be executed and the stream closed both if there is a problem and if there is not.

12.6 Checked Exceptions

A checked exception represents an error that is likely to happen when the code is executed. An exception is checked if it is a subclass of `Exception`, but not a subclass of `RuntimeException`. Programmers are **required** to implement error-recovery code whenever a checked exception is possible.

A typical example of a checked exception is the `IOException` class. This very broad class signifies that something has happened while attempting I/O operations. The `IOException` class has many subclasses, each of these are generally more specific. For example;

- `FileNotFoundException`
- `MalformedURLException`
- `RemoteException`

Whenever it is possible that a checked exception might be thrown by code in a method, the programmer must either catch the exception using a try-catch statement (called **handling**) or pass the exception to the class that called it. Passing the exception on is called **advertising**. This now meant that the other class will now face the same choice.

Lets consider the method `read` in the class `BufferedInputStream`. The declaration of this method is `public int read() throws IOException`. This means that the method may throw an `IOException`. The following code tries to use the `read` method, but it does not catch the error or advertise it, because of this the code will not compile.

Example: Attempting to use the read method

```
1 public int readInt(BufferedInputStream b) {  
2     return b.read();  
3 }
```

If we change the so that the exception is handled, then the code will compile.

Example: Using the read method and handling the exception

```
1 public int readInt(BufferedInputStream b) {  
2     try {  
3         return b.read();  
4     } catch (IOException e) {  
5         return -1;  
6     }  
7 }
```

If instead we decided that, we wanted to advertise the exception and not put any error handling code we could advertise it instead. This is done by adding the keyword **throws** followed by the name of the exception to the end of the method definition.

Example: Using the read method and advertising the exception

```
1 public int readInt(BufferedInputStream b) throws IOException {  
2     return b.read();  
3 }
```

This might seem like it is the easiest solution to the problem. However, now because the **readInt** declares that it may throw an **IOException** and code that uses it must have error-recovery code instead.

Catch or advertise

Choosing whether to catch or advertise is a design decision. If we want our programs to be robust, there must be error-recovery code somewhere. The best practice is to identify the most appropriate place to handle the exception. Wherever we choose, there should be enough information to write a meaningful error message and make a reasonable decision as to if and how to **continue** execution.

In the example above it is impossible to know which is the best choice without knowing what the code will be used for. If we assume that the code was being used to read the grades of students from a file, then returning -1 when a grade cannot be read without any warning would be bad. However, if the class calling the method knew that -1 meant a missing grade and adjusted, it would be OK. On the other side, if the exception is advertised, then the entire file reading operation might be interrupted by an exception instead of just reading a single number.

Continuing execution

The whole idea behind error recovery is that when an error happens that we are able to recover from it and continue executing. This is not always possible, some problems cannot be recovered from. However, when we do come across a problem, what do we do? Lets take an example of a method that is supposed to perform some operation and then return the result. If some error happens while the method is executing we have two options, we can return a special value like -1 or null, or we can throw an exception and allow the problem to be solved somewhere else.

Returning a special value

This is a very common solution to the problem. However, it is very important that this is explained in the documentation of the method. This way any programmer that uses your method will be aware that if something goes wrong the special value is returned and they can check for it.

Throwing an exception

When we encounter a problem and want to throw an exception, we have some choices to make. First, we must decide what type of exception to throw. Second we must decide where the exception should be thrown and lastly we may need to decide where the exception should be caught.

When facing a potential problem in our code, it is often easier to create our own exception to describe the problem. An exception is a class just like any other, we simply need to extend `Exception` and optionally add a small bit of code to explain the exception.

Lets assume that we have been given the task to design and implement a `Stack`. While we were writing the `pop` method, we realised that there could be a problem if the user tried to take an item from an empty stack. Depending on the implementation, this could cause a `NullPointerException` (link based implementation) or an `ArrayIndexOutOfBoundsException` (array based implementation). Instead we can create an exception to throw when this problem happens.

Example: The `StackEmptyException` class

```
1 public class StackEmptyException extends Exception {  
2     public StackEmptyException() {  
3         super("The Stack is Empty");  
4     }  
5 }
```

We can use this new exception in the code of our stack interface to define that this exception might be thrown if the stack is empty and then throw it in the implementation when the problem happens.

Example: Stack interface advertising the exception

```
1 public interface Stack {  
2     public int top() throws StackEmptyException;  
3     public int pop() throws StackEmptyException;  
4     public int push();  
5     public int size();  
6     public boolean isEmpty();  
7 }
```

Now we have declared that the methods `pop` and `top` both might throw the `StackEmptyException`. Next we need to add the code to the methods that will actually perform this action. As an exception is a class like any other, we must construct an object if we want to throw that exception. Therefore we need to use the keyword `new` and pass any parameters to the constructor that are required. This gives us `new StackEmptyException();` to create the object. Once the object is created, we use the keyword `throw` to throw the exception, however this is only done when we are sure that the stack is empty.

Example: Throwing the `StackEmptyException` in the `pop` method

```
1 public int pop() throws StackEmptyException {  
2     if (isEmpty())  
3         throw new StackEmptyException();  
4     int t = values[top - 1];  
5     top--;  
6     return t;  
7 }
```

Variable scope

We are used to the idea that whenever we declare a variable, it is only accessible from certain places. For example an instance variable (if it is public) can be accessed by anyone using an object of that type, a local variable (declared inside a method) can only be accessed inside that method. In Java, there are more levels to this idea. A variable declared inside a section of a try-catch statement, is only accessible inside that section. This means that in our example "Catching an exception in Integer.parseInt" the variable `x` can only be used inside the try section of the code.

Often this is an acceptable way of declaring and using variable, but sometimes we will want the variable to be accessible after the try section has finished executing. To do this, the variable must be **declared** before the try section. This allows the variable to be used later in our code.

Example: Variable declared before try section

```
1 int x;  
2 try {  
3     x = Integer.parseInt(s);  
4 } catch (NumberFormatException e){  
5     System.out.println("Not a valid int");  
6 }  
7 System.out.println("Integer value is "+x);
```

This technique has one problem. Variables declared in a method must be given a value before they can be used. If the try section is successful, the variable `x` is given a value, but if it is not then the variable `x` will have no value. The compiler will enforce that the variable must have a value and will not compile. Therefore we have a choice on how to solve the problem, we can give the variable a value when it is declared (as above) or we can have a value be assigned to the variable in the catch section of the statement.

12.7 Exceptions and Unit Tests

When testing code, the most difficult code to test can be error handling code. JUnit provides a way for us to test that our exceptions are being generated correctly by adding a little more information to the annotation. For example, if we design a test that should cause our code to throw an `IOException`, then instead of putting `@Test`, we put `@Test(expected = IOException.class)`.

This test can only be passed if the code throws an `IOException`. If the code completes successfully, then the test is not passed.

12.8 Practical Exercises

For the first exercises we will be improving the code of the `StringCalculator` class that we were testing in the last chapter.

1. Parameters can also be referred to as **arguments**. Using the `IllegalArgumentException` class in the package `java.lang`, add code to the constructor that will throw an exception when the base is not in the range 2 - 16. The exception should have show message stating the acceptable values for the parameter.
2. Define a class named `InvalidNumberForBaseException` which extends the `Exception` class. This constructor of this class should take two parameters, the first is a character and the second is the expected base. The error message of the class should state "Character X is not allowed in base YY numbers". For example "Character 9 is not allowed in base 8 numbers"
3. Modify the `convertToInt` method so that it advertises the exception. Add the required code to throw this exception when an illegal character is detected.

4. Modify each of the methods that uses the `convertToInt` method (add, subtract, multiply, divide and modulus) so that the `InvalidNumberForBaseException` is caught and the method returns a string containing the message `"Invalid parameter supplied calculation not performed"`
5. Rewrite the tests from the previous week so that they check that the exceptions are thrown in the correct circumstances.

Chapter 13

Streams and File I/O

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ Understand file and how they represent data
- ☐ understand the concept of streams in Java
- ☐ be able to create and use all of the different types of streams
- ☐ understand the concept of object serialisation

13.1 Files

We regularly save and view files every day, but rarely think about how the information is represented inside a file. This is because we usually don't have to, the correct program will open the correct file and display its contents correctly. But what if we are the developers creating this program? Then we must have at least some understanding of the basic file types that can be used.

There are two types of files, text files and data files. Although both of these can be separated into many different subtypes. Text files represent data in a format that is easy for humans to understand, all data is represented as letters numbers and other symbols. Data files are represented in a format that is easier for the computer to understand and are generally more efficient for storing data.

What can be difficult to understand (and is very important) is that all files are data files, text files just use a specific set of values to represent information.

Text Files

Text files are just data files where we use an encoding scheme such as ASCII or Unicode to store our data. These encodings are well known and well understood, so whenever we open a text files we are automatically shown the text and not the data it was encoded to. It is important that we understand **that** this conversion process is happening.

Figure 13.1 show a text file when viewed using a hex editor. Hex editors show the data in a file represented using hexadecimal (binary would be too large and hard to understand). This particular hex editor shows the individual values on the left each byte of data shown by two characters and a representation of the data as text on the right side. In this example, because the data is text, we can understand the right side of the screen.

The important point to remember is that the computer does not see a file containing "hello" it sees the bytes 68 65 6C 6C 6F, these bytes are the numbers 104, 101, 108, 108 & 111 in decimal format. The computer automatically translates these bytes into "hell" for us to understand, however they are just 5 bytes of data.

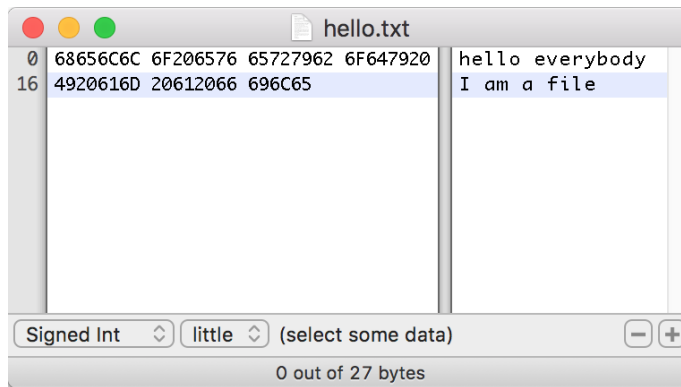


Figure 13.1: A text file when viewed using a hex editor

Data Files

Data files are exactly what they seem, there is no translation or encoding of the data into human readable forms. The data in these files is represented exactly as it is seen inside the computer. Figure 13.2 shows a data file represented in a hex editor. On the left we can see that there is a lot of data in the file, however on the right we can only see text in some places. This is because much of the data being represented is text, this information appears in different places in the file and is readable.

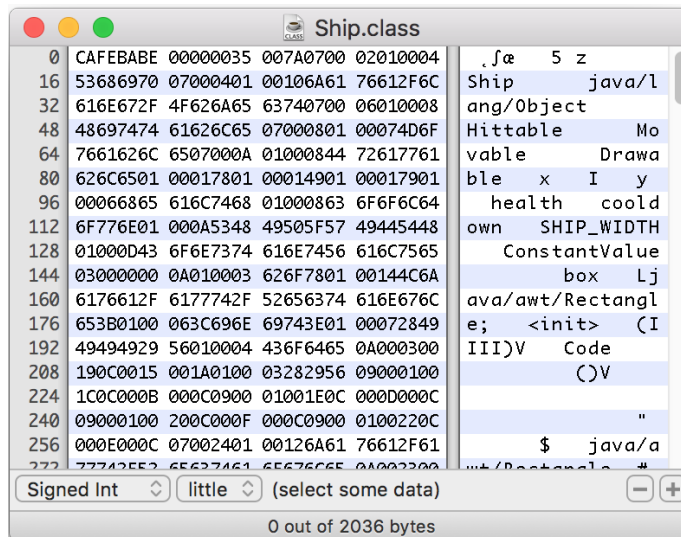


Figure 13.2: A data file when viewed using a hex editor

There is also occasionally random characters represented when the value of a piece of data falls within the range of ASCII values for visible characters. The differences in representation are important to remember when we get to section 13.5.

Text v Data

Neither file format is better than the other in all situations, they are each useful in different ways. Typically, text files are more easily understood by humans but are very inefficient at storing large amounts of information. Figure 13.3 shows the contents of two files in a hex editor. Both files contain the number 123456789, on the left this is encoded as text, and on the right it is encoded as binary data. If we open the first file in a text editor, then we can easily read the number. However

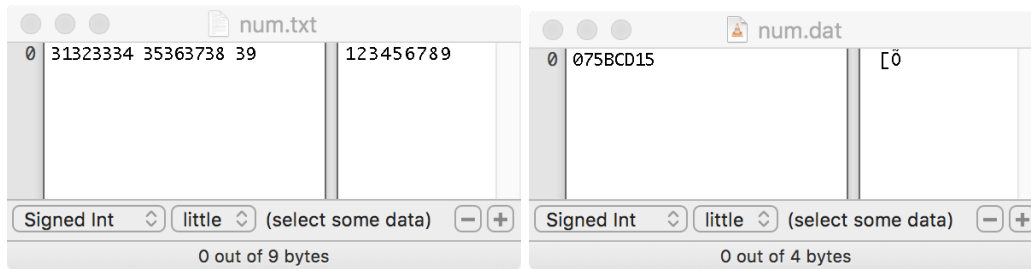


Figure 13.3: Hex editor showing text file and data file storing the same number

if we open the second file we will only see two characters that we do not understand. The file when encoded using text requires 9 bytes to be represented, while the second file requires only 4 bytes to store the number in binary.

13.2 Streams

This chapter focuses on the use of I/O streams in Java by using the examples of file reading. However, most of the examples can be applied to streams that are connected to another source or destination such as a network location.

An I/O stream represents an input **source** and output **destination**. This means that an I/O stream is really just a way to get data from one location to another. There are many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays. Different streams can carry different types of data, simple streams can only carry raw bytes of information while more complex streams can carry primitive data types and objects.

A stream can be viewed simply as a sequence of data. A program can use an input stream to read data from a source and use an output stream to write data to a destination. When we describe it as a sequence of data, what is really meant is that the data will have some order. If the source is a file it will be stored one piece of data after another, if the destination is a file we can only add one piece of data at a time.

There are many different types of streams;

- Byte streams
- Character streams
- Buffered streams
- Data streams
- Object streams

Byte streams

The most basic type of stream is a byte stream. These streams can only write raw data one byte at a time. This can be useful for operations such as copying a large amount of data. However in cases where the program needs to understand the meaning of the data, a byte stream is not very good. Think of a simple example where we wish to read an `int` from a source, an `int` is 4 bytes of information. To correctly read this `int` we would need to read all 4 bytes and then perform the correct operations to calculate the original value based on the 4 bytes.

Given that these streams are not very useful, why are we bothering to learn about them at all? The reason is that this basic functionality is used by other more complex streams. The more complex streams are capable of reading the 4 bytes and automatically converting them into an `int` for us. But internally they operate in this same basic way of reading a single byte at a time and then combining the data into the correct format.

There are many different types of byte streams, all of them extend the classes `InputStream` and `OutputStream` in the package `java.io`. In our examples, we will be using the `FileInputStream` and `FileOutputStream`. Lets look at the API entries for reading and writing in these classes.

API: `FileInputStream` - read

public int read() **throws** `IOException`

Reads a byte of data from this input stream. This method blocks if no input is yet available.

Returns: the next byte of data, or -1 if the end of the file is reached.

Throws: `IOException` - if an I/O error occurs.

API: `FileOutputStream` - write

public void write(int b) **throws** `IOException`

Writes the specified byte to this file output stream. Implements the write method of `OutputStream`.

Parameters: b - the byte to be written.

Throws: `IOException` - if an I/O error occurs.

Here we can see that the methods are generally simple to use, however both methods might throw an `IOException`. `IOException` is a direct subclass of the `Exception` class. This means that it is a checked exception and we must deal with the possible exception when we are using them.

One thing to note is that even though the method only reads a single byte, the actual return type of the method is `int`. This is because the stream reads bytes as values between 0 and 255, rather than between -128 and 127. This allows the special example of returning -1 when the end of the file has been reached.

Lets assume that we have a file on our hard drive and this contains 6 bytes. Lets write a program to read this information into an array and then print it to the screen.

Example: Example of reading 6 bytes from a file into an array and printing them

```

1 public static void main(String[] args) {
2     FileInputStream fis = null;
3     byte[] data = new byte[6];
4
5     try{
6         fis = new FileInputStream("data.dat");
7         for (int i = 0; i < data.length; i++) {
8             data[i] = (byte) fis.read(); // read a single byte
9         }
10    } catch (IOException e){
11        e.printStackTrace();
12    }
13    for (int i = 0; i < data.length; i++) {
14        System.out.print(data[i] + " ");
15    }
16 }
```

The code above is in reality quite simple, however it is complicated by the fact that we must include error-recovery code. All of the work in this code is completed in the four lines from 6 to 9, but the total code required is actually much larger. There are several things that we should note about this code.

- Our variables are declared before the try section of the try-catch statement.

- We use read in a loop to read all six bytes
- The return value of the read method must be typecast before it is stored in the array
- We are not not checking the return value to see if we have reached the end of the file

Now let us look at an example of code that creates a file and writes 6 bytes into it.

Example: Example of writing 6 bytes from an array to a file

```
1 public static void main(String[] args) {
2     byte[] data = new byte[]{12,45,66,19,88,99};
3     FileOutputStream fos = null;
4
5     try{
6         fos = new FileOutputStream("data.dat");
7         for (int i = 0; i < data.length; i++) {
8             fos.write(data[i]); // write a single byte
9         }
10        fos.close();
11    } catch (IOException e){
12        e.printStackTrace();
13    }
14 }
```

Here again, the code is quite simple but complicated by error recovery. The part of the code we should note is a statement on line 10 called `close`.

Closing Streams

Whenever we create a stream in Java, a number of resources are created in memory to help the process of reading and writing. If we do not correctly deal with streams, these will continue to waste memory in our program, causing it to run more slowly or crash when we run out. The JVM limits the amount of memory available to each program.

After we have finished using a stream we should close it. This is done using the method called `close`. This is something that we should always do. There are a number of strategies for choosing where to do this. In the example above, it is done when we are finished writing to the file. However, if there is a problem before we have finished writing, the close method will not be executed and therefore the stream is not closed.

Alternatively, we can use the finally section to close the stream we are using. However, because the close method might throw `IOException`, we must include another try-catch statement. Additionally, because the stream may be null (if there is an exception before or during its creation), we must first check that it is not null before trying to close it.

Example: Closing a stream in the finally section

```

1 try{
2     ...
3 } catch (IOException e){
4     e.printStackTrace();
5 } finally {
6     if(fos != null){
7         try { fos.close(); } catch (Exception e2) {
8             System.out.println("Stream could not be closed");
9         }
10    }
11 }

```

Try with resources

This gives us code that is very messy and difficult to understand. For this reason in Java 7, a new feature was added to simplify the process of using and closing streams. The try with resources statement, is a special version of the try-catch statement that is designed to automatically close streams when we are finished using them.

Syntax: The try with resources statement

```

1 try(declare and construct stream){
2     // use stream
3 } catch (expected exception){
4     // handle expected exception
5 }

```

This allows us to declare and construct our stream and then not need to worry about closing it. It is also possible to not use a catch statement, if the exceptions thrown in your code are advertised instead of handled. If we look at the write bytes example again, we can see that the code is much shorter and the easier to understand.

Example: Writing bytes using the try with resources statement

```

1 public static void main(String[] args) {
2     byte[] data = new byte[] { 12, 45, 66, 19, 88, 99 };
3     try (FileOutputStream fos = new FileOutputStream("data.dat");) {
4         for (int i = 0; i < data.length; i++) {
5             fos.write(data[i]);
6         }
7     } catch (IOException e) {
8         e.printStackTrace();
9     }
10 }

```

13.3 Character Streams

The Java platform stores character values using Unicode conventions, these are similar to ASCII but much more comprehensive. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit

superset of ASCII. This means that if I write a program to read some text from a file, it will automatically adjust to read characters correctly for the user. For example if I write a text file in English and read it using my program, it will correctly identify each of the characters in the file. If you write a text file in Chinese and use the exact same program to read it, each character will be correctly identified.

There are many different types of character streams, all of them extend the classes `Reader` and `Writer` in the package `java.io`. In our examples, we will be using the `FileReader` and `FileWriter`. Lets look at the API entries for reading and writing characters in these classes.

API: `FileReader` - `read`

public int `read()` **throws** `IOException`

Reads a single character.

Returns: The character read, as an integer in the range 0 to 65535 (0x00-0xffff), or -1 if the end of the stream has been reached

Throws: `IOException` - if an I/O error occurs.

API: `FileWriter` - `write`

public void `write(int c)` **throws** `IOException`

Writes a single character. The character to be written is contained in the 16 low-order bits of the given integer value; the 16 high-order bits are ignored. Subclasses that intend to support efficient single-character output should override this method

Parameters: `c` - int specifying a character to be written.

Throws: `IOException` - if an I/O error occurs.

These are very basic methods again, so we will look at an example that reads 30 characters from a file, converts each character to upper case and stores them in an array.

Example: Reading 30 characters one at a time from a file

```

1 public static void main(String[] args) {
2     char[] characters = new char[30];
3     try (FileReader fr = new FileReader("chars.txt")) {
4         for (int i = 0; i < characters.length; i++) {
5             char c = (char) fr.read(); //read character
6             characters[i] = Character.toUpperCase(c);
7         }
8     } catch (IOException e) {
9         e.printStackTrace();
10    }
11    System.out.print(characters);
12 }
```

The code for reading information from the file is exactly the same in this example as in the byte stream example. The only difference between the two is that if there are characters stored as more than a single byte in the file, then they will be read as one by the character stream and read a separate bytes in the byte stream. If we also look at the example for writing characters to a file we see it is also very similar.

Example: Writing characters from a string to a file

```

1 public static void main(String[] args) {
2     String soundOfSilence = "Hello darkness, my old friend\nI've come to
      talk with you again";
3
4     try(FileWriter fw = new FileWriter("lChars.txt")){
5         for (int i = 0; i < soundOfSilence.length(); i++) {
6             char c = soundOfSilence.charAt(i); // get character
7             fw.write(c); // write character
8         }
9     } catch (IOException e){
10         e.printStackTrace();
11     }
12 }

```

13.4 Buffered Streams

Reading and writing characters is useful, but characters are small pieces of information. Generally, we are used to working with larger pieces of information such as strings. When reading and writing strings, generally work one line at a time. So we will look at some classes that allow us to read and write data using strings.

The classes that studying are what is known as **buffered**. These streams are designed to be much more efficient than the streams that we have been using so far. When using normal I/O streams, every time we call the read or write method the functionality is performed by the operating system. This often causes the operating to have to access a file or a network resource, which is a very slow operation. When we use a buffered stream, data is read from or written to an area of memory called a **buffer** each time we use the read or write methods.

When the buffer is full (for writing) or empty (for reading), all of the data is written or new data is read. Because this is done in a single operation for larger amounts of data, the process happens much faster. Lets take the example where we wrote 6 bytes to a file. If this code was being executed on a machine with a magnetic hard drive, then every time we write a byte the operating system would have to start spinning the drive, move the head to the correct location and then write the data. Once this was completed the process would be repeated. However, if we use a buffered stream nothing is written to the disk until the buffer is full. Because of the way hard drives work (separated into sectors of between 512 and 4096 bytes), it takes the same amount of time to write one byte as to write 100. The reverse is also true for reading.

Any stream can be converted into a buffered stream. The idea is that we take a stream that will write data to a source and we place it inside a buffered stream. When the buffer needs to be filled or emptied, it then calls the read or write method of the inner stream. This means that in order to create a buffered stream, first we have to a stream to pass as a parameter. This process is referred to as **stream wrapping**.

There are different types of buffered streams for different types of data. For example to create a buffered byte stream we would use the `BufferedInputStream` or `BufferedOutputStream` and for character streams we would use the `BufferedReader` and `BufferedWriter`.

Constructing a buffered stream

If we wished to create a buffered character stream, we first need to create the stream to pass as a parameter. So if we are reading a file "example.txt", first we would create `FileReader fr = new FileReader("example.txt");`. Once this is created, we can use it to create a `BufferedReader` object. `BufferedReader br = new BufferedReader(fr);`. Note that the parameter to the buffered stream is the file reader that we constructed in the first statement.

These statements can be combined into a single statement by passing the result of the constructor of the first into the constructor of the second.

E.g. `BufferedReader br = new BufferedReader(new FileReader("chars.txt"))`

Flushing streams

When writing data using a buffered stream, the data is only written when the buffer is full. But what if the buffer is not full and we want the data to be written now? Java provides a method that we can use at any time to write any data that is currently in the buffer into the destination. This method is called `flush`. Whenever the flush method is executed, all data in the buffer will be written even if it is only a single byte.

Reading and writing with buffered streams

The buffered streams provide the methods `read` and `write` which work exactly the same way as in the character and byte streams that we have seen already. However, they also add methods that are much easier to use when reading or writing data in larger amounts. Lets look at the methods of the `BufferedReader` and `BufferedWriter`.

API: BufferedReader - readLine

public String readLine() **throws** IOException

Reads a line of text. A line is considered to be terminated by any one of a line feed (`\n`), a carriage return (`\r`), or a carriage return followed immediately by a linefeed.

Returns: A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws: `IOException` - if an I/O error occurs.

API: BufferedWriter - write

public void write(String str) **throws** IOException

Writes a string.

Parameters: str - String to be written

Throws: `IOException` - if an I/O error occurs.

Lets look at an example where we read 7 lines from a file and store them in an array to be printed later.

Example: Reading strings into an array

```

1 public static void main(String[] args) {
2     String[] lyrics = new String[7];
3     try(BufferedReader br = new BufferedReader(new FileReader("chars.txt"
4     ))) {
5         for (int i = 0; i < lyrics.length; i++) {
6             lyrics[i] = br.readLine();
7         }
8     } catch(IOException e) {
9         e.printStackTrace();
10    }
11    for (int i = 0; i < lyrics.length; i++) {
12        System.out.println(lyrics[i]);
13    }
14 }
```

And another example where we write every string in an array into a file, each as a separate line.

Example: Writing an array of Strings as lines in a file

```

1 public static void main(String[] args) {
2     String[] words = new String[] { "Hi", "my", "name", "is", "Sean" };
3
4     try (BufferedWriter bw = new BufferedWriter(new FileWriter("words.txt"
5         ))) {
6         for (int i = 0; i < words.length; i++) {
7             bw.write(words[i] + "\n");
8         }
9     } catch (IOException e) {
10        e.printStackTrace();
11    }
12 }

```

Processing Strings

We learned some basic string processing in chapter 9. This can be easily applied to strings that have been read from files. Consider an example of a program that reads lyrics of a song from one file and replaces some of the words before writing it into another file. In this example we will use the `replace` method in the string class to achieve our aim.

The biggest problem that we face is that we do not know how many lines that there are in the file. We could check and read that much information, but every time we want to use the program on a different file, we would need to find the number of lines. Instead, it is easier to check every time if we have reached the end of the file. In the API documentation of the `readLine` method, it states that null is returned when the end of the stream is reached. So we simply try to read a line and loop as long as the variable is not null.

Example: Reads and writes lines from one file to another making small changes

```

1 public static void main(String[] args) {
2     try (BufferedReader br = new BufferedReader(new FileReader("chars.txt"
3         ));
4         BufferedWriter bw = new BufferedWriter(new FileWriter("words.txt"
5         ))) {
6         String line = br.readLine();
7         while (line != null) {
8             bw.write(line.replace("silence", "music") + "\n");
9             line = br.readLine();
10        }
11    } catch (IOException e) {
12        e.printStackTrace();
13    }
14 }

```

It should be noted, that because the `readLine` method does not include any new line characters in the strings it returns when writing these strings to the file the must be added.

13.5 Data Streams

Data is represented differently in the computer than the way it is presented to us. For example when we print the number 123456789 we see this in the console as 9 characters, however in the computer internally it is represented as 00000111 01011011 11001101 00010101. This is convenient because

the second is very difficult to understand for humans. However, if we were to write this number into a file using a character stream the information will require 9 bytes to be represented, while the second version takes on 4 bytes to be represented inside the computer. If we have a file that is storing several million numbers, this difference will make the file much larger than it needs to be.

Data streams allow us to store primitive data type values and strings in the same format as they are stored in memory in the computer. This means that the files are in a format the computer can easily understand, but to a human will often not make any sense. This makes the operating with files more efficient, because we are no longer required to convert between the human version and the computer version.

For writing and reading data we will use the `DataInputStream` and `DataOutputStream`. These streams are constructed in the same way as the buffered streams. In fact they are usually constructed **using** buffered streams. This means that if we want our program to be efficient we need to create a total of three streams to read or write data.

First we need to construct a `FileInputStream`, then using that we construct a `BufferedInputStream` and finally using that we construct a `DataInputStream`. It is acceptable to not use the buffered input stream and instead simply pass the file input stream to the data input stream, however this will result in a program that is less efficient. The process is similar for the data output stream.

Example: Constructing a data input stream

```

1 FileInputStream fis = new FileInputStream("grades.dat");
2 BufferedInputStream bis = new BufferedInputStream(fis);
3 DataInputStream dis = new DataInputStream(bis);
4
5 OR
6
7 DataInputStream dis = new DataInputStream(new BufferedInputStream(new
    FileInputStream("grades.dat")));

```

Different data

Because we can store different types of data and each will be stored in the same format as in memory, we need a method for each of these types of data. For example, if I want to write an integer into a file I cannot use the same method that writes a double because these types are stored in different ways. For example to write data we have the methods `writeInt` for ints, `writeChar` for chars, `writeDouble` for doubles and the same for other primitive type. To write a string we use the method `writeUTF`. For example to read data we have the methods `readInt` for ints, `readChar` for chars, `readDouble` for doubles and the same for other primitive type. To read a string we use the method `readUTF`.

Lets look at two examples of the reading methods `readInt` and `readUTF` in the API.

API: DataInputStream - readInt

public int readInt() **throws** IOException

Reads four input bytes and returns an int value. Let a-d be the first through fourth bytes read. The value returned is:

`((a & 0xff) << 24) | ((b & 0xff) << 16) | ((c & 0xff) << 8) | (d & 0xff))`

Returns: the next four bytes of this input stream, interpreted as an int.

Throws: `EOFException` - if this input stream reaches the end before reading four bytes.

`IOException` - the stream has been closed and the contained input stream does not support reading after close, or another I/O error occurs.

API: DataInputStream - readUTF

public final String readUTF() **throws** IOException

Reads in a string that has been encoded using a modified UTF-8 format. The general contract of readUTF is that it reads a representation of a Unicode character string encoded in modified UTF-8 format; this string of characters is then returned as a String.

Returns: a Unicode string.

Throws: EOFException - if this input stream reaches the end before reading four bytes.

IOException - the stream has been closed and the contained input stream does not support reading after close, or another I/O error occurs.

UTFDataFormatException - if the bytes do not represent a valid modified UTF-8 encoding of a string.

Assume that we have a file that contains the names, student numbers (as ints) and three grades for a number of students. Lets write a program to read the information about the first student stored in the file and then print the information to the screen. Just like with the scanner and user input, the order that we write our statements must match the order that the data appears in the file.

Example: Reading student information using the DataInputStream

```

1 public static void main(String[] args) {
2     String name = "";
3     int studentNumber = -1;
4     double[] grades = new double[3];
5     try{
6         DataInputStream dis = new DataInputStream(new BufferedInputStream(
7             new FileInputStream("grades.dat")));
8         name = dis.readUTF();
9         studentNumber = dis.readInt();
10        for (int i = 0; i < grades.length; i++) {
11            grades[i] = dis.readDouble();
12        }
13    } catch (IOException e){
14        e.printStackTrace();
15    }
16    System.out.print("Student " + name + " Number: " + studentNumber +
17        " got grades ");
18    for (int i = 0; i < grades.length; i++) {
19        System.out.print(grades[i] + " ");
20    }
21 }

```

Note here that first we read the name of the student using the `readUTF` method, then the student number and then the three grades. Because each of these pieces of information is stored in a different way, mixing up the order would result in data that does not make any sense and might crash the program.

Now lets look at an example of using the `DataOutputStream` to write the same data into a file for a single student.

API: DataOutputStream - writeInt

public final void writeInt(**int** v) **throws** IOException

Writes an int to the underlying output stream as four bytes, high byte first.

Parameters: v - an int to be written.

Throws: IOException - if an I/O error occurs.

API: DataOutputStream - writeUTF

public final void writeUTF(String str) **throws** IOException

Writes a string to the underlying output stream using modified UTF-8 encoding in a machine-independent manner. First, two bytes are written to the output stream as if by the `writeShort` method giving the number of bytes to follow. This value is the number of bytes actually written out, not the length of the string. Following the length, each character of the string is output, in sequence, using the modified UTF-8 encoding for the character.

Parameters: str - a string to be written.

Throws: IOException - if an I/O error occurs.

Example: Writing student information using the DataOutputStream

```

1 public static void main(String[] args) {
2     String name = "Sean Russell";
3     int sn = 12345678;
4     double[] grades = new double[] { 100, 95, 77.7 };
5
6     try (DataOutputStream dos = new DataOutputStream(new
7         BufferedOutputStream(new FileOutputStream("grades.dat")))) {
8         dos.writeUTF(name);
9         dos.writeInt(sn);
10        for (int i = 0; i < grades.length; i++) {
11            dos.writeDouble(grades[i]);
12        }
13    } catch (IOException e) {
14        e.printStackTrace();
15    }
16 }
```

13.6 Object Streams

Writing the large quantities of complex data can be a difficult task. Generally in object-oriented programming we are not dealing with many small pieces of data, we are dealing with objects that contain many pieces of related data. In general, instead of writing different types of data in a specific it is easier to simply write the entire object containing this data instead.

Object streams allow us to write and read the entire contents of an object as a single piece of data. This can greatly simplify code we are writing that deals with complicated objects. Before we can write an object in this way, it must implement the interface `Serializable`. The serializable interface does not require that any extra methods are implemented and is already implemented by many standard classes such as `String`.

Where this can be most difficult is when we have an object that contains a reference to another object as an instance variable. In this case both objects must implement the `Serializable` interface as well as any objects that they contain a reference to. For example, if we have a class called `Grade` and another class called `Student` that contains an instance variable of the type `grade`, then both must be implement the interface before we can write the `Student` object. Additionally, if the `Grade`

class contains an instance variable of the type `Assignment`, this must also be serializable before we can write the `Student` object.

The classes that we use to write objects are `ObjectInputStream` and `ObjectOutputStream`. These classes also contain all of the methods associated with data streams. That means that we can use a single stream to write primitive data types and objects.

These streams are constructed in the same way as the data streams, requiring three streams to be created efficiently. First we need to construct a `FileInputStream`, then using that we construct a `BufferedInputStream` and finally using that we construct a `ObjectInputStream`. The process is similar for the object output stream.

Example: Constructing an `ObjectInputStream`

```

1 FileInputStream fis = new FileInputStream("grades.dat");
2 BufferedInputStream bis = new BufferedInputStream(fis);
3 ObjectInputStream ois = new ObjectInputStream(bis);
4
5 OR
6
7 ObjectInputStream ois = new ObjectInputStream(new BufferedInputStream(
    new FileInputStream("grades.dat")));

```

The methods that we can use are `readObject` and `writeObject`. Let's have a look at the API for the `readObject` method.

API: `ObjectInputStream` - `readObject`

public final `Object` `readObject()` **throws** `IOException`, `ClassNotFoundException`
 Read an object from the `ObjectInputStream`. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are read. Objects referenced by this object are read transitively so that a complete equivalent graph of objects is reconstructed by `readObject`.
Returns: the object read from the stream
Throws: `ClassNotFoundException` - Class of a serialized object cannot be found.
`InvalidClassException` - Something is wrong with a class used by serialization.
`StreamCorruptedException` - Control information in the stream is inconsistent.
`OptionalDataException` - Primitive data was found in the stream instead of objects.
`IOException` - Any of the usual Input/Output related exceptions.

As we can see there are a lot of exceptions that can happen when we are reading an object from a stream. The most common problem is the `ClassNotFoundException`. If we read an object and we do not have that class in our code we will get a `ClassNotFoundException`. This can happen very easily if we are only controlling one side of the read/write process. Let's have a look at the API for the `writeObject` method.

API: ObjectOutputStream - writeObject

public final void writeObject(Object obj) **throws** IOException

Write the specified object to the ObjectOutputStream. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are written. Objects referenced by this object are written transitively so that a complete equivalent graph of objects can be reconstructed by an ObjectInputStream. Exceptions are thrown for problems with the OutputStream and for classes that should not be serialized.

Parameters: obj - the object to be written

Throws: **InvalidClassException** - Something is wrong with a class used by serialization.

NotSerializableException - Some object to be serialized does not implement the `java.io.Serializable` interface.

IOException - Any exception thrown by the underlying OutputStream.

To test these methods, let's create a simple class and write it into a file. The **Student** class, only holds three pieces of information, the name, student number and age of a student. We will use this to create some student objects and store them in a file.

Example: The Student class

```

1 import java.io.Serializable;
2
3 public class Student implements Serializable {
4     String name;
5     int age;
6     int studentNumber;
7     public Student(String n, int a, int sn){
8         name = n;
9         age = a;
10        studentNumber = sn;
11    }
12 }

```

When reading objects from a file, we must be careful that we are reading the correct object. The return type of the method is **Object** this means that after we have read the object from the file, we must typecast it into the correct form. As we saw earlier, before performing a typecast, we should always check the type of the object we are about to cast.

Additionally, there should be two separate catch statements. This is because generally we would want to perform different actions depending on if there was an **IOException** or a **ClassNotFoundException**. An IO exception might simply mean that the name of the file was specified incorrectly or simply that we have reached the end of the file. If the class was not found, then we have a bigger problem that likely can only be solved by including that class and recompiling the whole program.

Example: Reading three objects from a file

```

1 public static void main(String[] args) {
2     Student[] s = new Student[3];
3     try (ObjectInputStream ois = new ObjectInputStream(new
4         BufferedInputStream(new FileInputStream("students.dat")))) {
5         for (int i = 0; i < s.length; i++) {
6             Object o = ois.readObject();
7             if (o instanceof Student) {
8                 s[i] = (Student) o;
9             }
10        }
11    } catch (IOException e) {
12        e.printStackTrace();
13    } catch (ClassNotFoundException e) {
14        e.printStackTrace();
15    }
16    for (int i = 0; i < s.length; i++) {
17        System.out.println("Name: " + s[i].name + " Number: " + s[i].
18            studentNumber + " Age: " + s[i].age);
19    }
20 }

```

There are less problems likely to happen when writing an object. Because we must have the class to create the object, it is not possible to generate a class not found exception. However, there is the possibility of a `NotSerializableException`. This can happen if the object that we are trying to write or any of the objects that is references are not declared as `Serializable`.

Example: Writing three students to a file

```

1 public static void main(String[] args) {
2     Student[] students = new Student[]{new Student("Sean", 30, 12345678),
3     new Student("Paul", 60, 99345678),
4     new Student("James", 32, 11345678)};
5     try (ObjectOutputStream ois = new ObjectOutputStream(
6         new BufferedOutputStream(new FileOutputStream("students.dat")))) {
7         for (int i = 0; i < students.length; i++) {
8             ois.writeObject(students[i]);
9         }
10    } catch (IOException e) {
11        e.printStackTrace();
12    }
13 }

```

13.7 Game Example

One typical use that an arcade game might make use of files is a score board. The names of the 10 highest players are remembered until they are beaten. We will write a `ScoreBoard` class to represent these high scores.

The basic functionality required for this class is;

- the ability to load scores from a file
- the ability to save scores to a file

- the ability to sort the scores
- the ability to return the scores for displaying
- the ability to add a new score to the list

Example: The ScoreBoard and Score classes

```
1 public class ScoreBoard {
2     protected Score[] scores;
3
4     public class Score {
5         private String name;
6         private int value;
7
8         public Score(String n, int s) {
9             this.name = n;
10            this.value = s;
11        }
12        public String getName() {
13            return name;
14        }
15        public int getScore() {
16            return value;
17        }
18    }
19    public ScoreBoard() {
20        scores = new Score[10];
21        for (int i = 0; i < scores.length; i++) {
22            scores[i] = new Score("Player 1", 0);
23        }
24        loadScores();
25        sortScores();
26    }
27    ...
28 }
```

This example shows the definition of the scoreboard class, it just contains an array of score objects which are defined as an inner class. An important point to note is that the constructor fills the array with objects in case there is any problem with loading the file. Then the `loadFile` method is called and the highest scores are put in the array.

The format being used in the file is that first, the score is stored and then the rest of each line contains the name of the player. This means that we can read each score by reading the file one line at a time, then processing the line to find the correct information.

Example: Loading the scores from the file

```
1 private void loadScores() {
2     try (BufferedReader br = new BufferedReader(new FileReader("scores.
3         txt"))) {
4         String line = br.readLine();
5         int i = 0;
6         while (line != null && i < 10) {
7             Scanner s = new Scanner(line);
8             int score = s.nextInt();
9             String name = s.nextLine().trim();
10            scores[i++] = new Score(name, score);
11            line = br.readLine();
12            s.close();
13        }
14    } catch (IOException e) {
15        System.err.println("Error reading score file");
16    }
```

Here inside the loop we use a scanner to break each line up, first reading the score using `nextInt` and then the rest of the text using `nextLine`. To get rid of unwanted spaces at the beginning or end of the name, we use the method `trim` in the string class. The values read are then used to create a new score object and it is put into the array.

Next we need a method to save the scores to the file when the program is complete. This must use the exact same format for saving the file or our load code will not work. Therefore, we write the score first, then add a space and the players name followed by a new line character.

Example: Saving the scores to the file

```
1 public void saveScores() {
2     try (BufferedWriter bw = new BufferedWriter(new FileWriter("scores.
3         txt"))) {
4         for (int i = 0; i < scores.length; i++) {
5             bw.write(scores[i].getScore() + " " + scores[i].getName() + "\n");
6         }
7         bw.close();
8     } catch (IOException e) {
9         System.err.println("Error writing score file");
10    }
```

To return the scores, we only need to create a new array and copy the score objects. This is to prevent someone from changing the data in our array.

Example: Returning the top scores

```

1 public Score[] getScores() {
2     Score[] scs = new Score[10];
3     for (int i = 0; i < scs.length; i++) {
4         scs[i] = scores[i];
5     }
6     return scs;
7 }

```

Another place we could use files is in remembering the shapes of the various ships, we will implement this for the three types of enemy ships as an example. These are all stored in the AlienType enumerated type. Again we need to decide on the formatting of the data, rather than store the information as text, this time we will store it as data in binary format. This means that we will either need to use a hex editor to create the files, or we can write another program to create the data files.

In the files we will remember all of the important data about the shape, this includes the width of the ship, the score we get when it is destroyed and the series of coefficients that we can use to draw it on the screen. If we are storing data, we can easily just write the first two pieces of data as ints. To read the array from the file, we first need to know how many rectangles there are in the shape so we store this number next. Finally, using a loop or two nested loops we put all of the individual coefficients of the shape into the file. Assuming all of this has been done correctly, the following code will create all three ship shapes.

Example: Constructor for the AlienType enum

```

1 public enum AlienType {
2     A("A.dat"), B("B.dat"), C("C.dat");
3     private int width;
4     private int height;
5     private int score;
6     private int [][] shapes;
7
8     private AlienType(String n) {
9         try (DataInputStream dis = new DataInputStream(new FileInputStream(
10             n))) {
11             width = dis.readInt();
12             height = dis.readInt();
13             score = dis.readInt();
14             int nShapes = dis.readInt();
15             shapes = new int[nShapes][4];
16
17             for (int i = 0; i < nShapes; i++) {
18                 for (int j = 0; j < 4; j++) {
19                     shapes[i][j] = dis.readInt();
20                 }
21             }
22         } catch (FileNotFoundException e) {
23             e.printStackTrace();
24         } catch (IOException e) {
25             e.printStackTrace();
26         }
27     }
28     ...

```

13.8 Practical Exercises

1. Define a class method named `countBytes`. This method should take the a string as a parameter. This string represents the name of the file that we should read. The method should read the data from the file using a byte stream and count the number of bytes in the file. If there is any sort of I/O problem, the message "Error reading file" should be printed to the screen.
2. Define a class method named `countCharacters`. This method should take a single String as a parameter. The method should create a character stream to read the data from the file (named by the parameter) and count the number of characters in the file. If there is any sort of I/O problem, then the message "Error reading file" should be printed to the screen.
3. Define a class method called `averageIntegers`. This method should take a string as a parameter. The string is the name of the file from which to read data. Data is stored in this file as a series of integer values. These should be read using a `DataInputStream`. Based on the values read in you should print the following information:
 - The minimum value in the file in the format "Min Value: XXXX"
 - The maximum value in the file in the format "Max Value: XXXX"
 - The average value in the file (as an integer) in the format "Average Value: XXXX"
4. Define a class method named `calculateGrades`. This method should take a String as a parameter (representing the name of the file to be read). The method should read the grade information for each student and then print the students number and their average grade. The information is stored in the file for each student is in the following format: an integer (representing the students number) and 3 doubles (representing their grades). You can try it using the file `grades.dat`
5. Define a class method named `printGrades`. This method should take a string as a parameter. The file with this name should be read using an object stream. The file contains a number of student objects (defined in `Student.java`). Read each of the objects from the file and print the student numbers of each student.
6. Define a class method named `writeFile`. This method should take a string as a parameter. The method should use a scanner to read a single line of text from the user and then use a character stream to write the text into the file named in the parameter.
7. Define a class method named `writeNumbers`. This method should take a string as a parameter. The method should read 3 int values from the user, using a scanner, and write these into the file named in the parameter using a data output stream.
8. Define a class method named `readPrompt`. This method should take no parameters. The method should use an input dialog to ask the user to enter a file name. The method should then attempt to read the entire file into a String and return it. If there is a file not found exception, the user should be asked again for the file name (HINT recursion might be a good way to achieve this). If there is any other kind of I/O error the string "could not read XXXX.XXX" should be returned (where XXXX.XXX is the named of the file).

13.9 Game Completion Tasks

1. Add the method `addScore` to the `ScoreBoard` class. This should take a score object as a parameter and insert it to the array if it is bigger than the smallest score in the array.
2. Add the method `sortScores` to the `ScoreBoard` class. This should take no parameters and sort the scores into descending order based on their value.

The enemy ships never attack as a single ship, they move together as one force. We will define a class to help us capture how the enemy ships will move and attack.

3. Define class named **Swarm**, this will be used to represent the enemy force when they are moving and attacking.
4. Define an array in the swarm class for remembering the enemy objects.
5. Define instance variables for remembering the x and y coordinates of the swarm, as well as the direction that the ships are moving in.
6. Define a method named **getShipsRemaining**. This should take no parameters and return an integer value that is the number of ships that have not been destroyed.
7. Define a method named **getShips**, this should take no parameters and return an array containing only the ships that have not been destroyed.
8. Define a method named **update**, this should take no parameters and return no value. In the method, the update method of each enemy ship that has not been destroyed should be called.
9. Define a method named **getBottomShips**, this should take no parameters and return an array containing Enemy objects. These enemy objects should be the ships at the bottom of the columns, these are the ships that are allowed to fire.

Chapter 14

Advanced Topic - Concurrent Programming/Threads

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the concept of concurrent programming
- ☐ understand how threads are used to implement concurrent programming in Java
- ☐ be able to create multi threaded programs in Java
- ☐ understand the use of join, sleep and interrupt to control the execution of threads

14.1 Concurrent Programming

Concurrent programming is where several computations are executed at the same time (concurrently) instead of one completing before the next starts (sequentially). Concurrent programming can be achieved using multiple processes or multiple threads or both. This is a concept that you will study in your Operating Systems class. In Java, concurrent programming is implemented using multiple **threads**.

The original idea of concurrent programming was to allow multiple programs to share the same processor, but only a single program could be executed at a time. The basic idea is that while one program was performing an action that did not require the use of the CPU, such as reading data from a file, another program could use the CPU to perform calculations.

Processors have developed since then and most processors allow for multiple calculations to be performed at the same time. Processors now have at least 2 cores (units in the CPU that can perform calculations), many desktop CPUs can have as many as 4 or 8 cores. Server processors commonly have many more cores.

By making use of this fact, we can write programs that are able to perform many calculations at the same time and therefore complete their tasks quicker.

What is a Thread?

A thread is a sequence of instructions that are executed in order. This includes all types of statements such as calling methods, if statements and loops. Java programs are executed in a thread created based on the instructions in the main method. This task is handled automatically for us, so every program we write in Java will have at least one thread. If we wish to have our programs complete more than one action at a time we can add more threads to our programs.

Threads typically are associated with a name, this allows us to determine which thread was responsible for a particular exception. Remember every exception begins with the words "**Exception**

in `thread`". The name can be passed as a string to the constructor of the thread class, and returned using the method `getName`. If a name is not supplied by the programmer, the system will automatically generate a name such as `"thread-1"`.

In all of the programs we have written so far we have only created a single thread ourselves. However, other components of the JVM may have created threads to help with the operations that they perform. Examples of this would be any code that we wrote with A graphical interface, these require multiple different threads to manage things such as listening for key presses and mouse actions or redrawing the screen when required. Additionally, features of the language such as memory management require a thread to execute the operations that they perform for use.

When we add more threads to our programs, they are known as **multithreaded**. To do this we must use the `Thread` class.

14.2 Creating Threads

There are two ways that we can create a new thread, the first is to extend the `Thread` class and implement the required method. The second is to implement the `Runnable` interface, create an object based on the class and then pass this object to a thread.

When we have created a thread, we must start its execution in the correct way. If this is not done correctly, the code will be executed sequentially rather than concurrently.

Extending the Thread class

To define a new thread using the `Thread` class, we only have to override a single method called `run`. The signature of `run` is **public void** `run()`. In the `run` method, we add the code that we wish to be executed, in the same way as we normally add the code that we want to be executed to the `main` method. When the an object based on this class is started, these actions will be performed.

To test this, we will write a very simple example class. This class is called `AThread`, and will simply print the letter 'a' to the screen some number of times.

Example: A thread for printing a to the screen 30 times

```

1 public class AThread extends Thread {
2     public void run() {
3         for (int i = 0; i < 30; i++) {
4             System.out.print('a');
5         }
6     }
7 }

```

Here we can see that the code for printing 'a' to the screen is exactly the same as if we had written it in the `main` method.

Implementing the Runnable interface

To define a new thread using the `Runnable` interface, we only have to implement a single method called `run`. The signature of `run` is **public void** `run()`. In the `run` method, we add the code that we wish to be executed, in the same way as we normally add the code that we want to be executed to the `main` method. When the an object based on this class is passed to a thread object, and the thread object is started, these actions will be performed.

To test this, we will write a very simple example class. This class is called `BRunnable`, and will simply print the letter 'b' to the screen some number of times when it is executed.

Example: A runnable for printing b to the screen 30 times

```

1 public class BRunnable implements Runnable {
2     public void run() {
3         for (int i = 0; i < 30; i++) {
4             System.out.print('b');
5         }
6     }
7 }

```

How to choose

When we need to create a thread to perform some actions concurrently, which of the above techniques should we choose? Generally, it is easier to manage when we extend the Thread class. This is the technique we should use whenever possible. However, it is not always possible. Because each class can only have a single superclass, in situations where the class needs to inherit functionality from another class it is not possible to extend the Thread class. In these cases, we should implement the runnable interface instead.

14.3 Executing Threads

These threads as we have defined them will not perform any action, because they have not been started. The execution of a thread is done by calling the method **start** and not by calling the method **run**. If you call the run method instead of the start method, the code will be executed sequentially. When the start method is called, the new thread is set up and then the run method is executed in this new thread.

If we have define a thread by extending the Thread class, then to execute the thread we only have to complete two steps;

- Create an object based on the class that we have defined
AThread at = **new** AThread();
- Call the start method
at.start();

If we have executed this code in the main method, then after they are executed there will be two threads. The first thread executes the main method (any code that comes after the start method is called) and the second thread executes the run method of the AThread object.

If we have defined a thread by implementing the Runnable interface, then we are required to execute three steps;

- Create an object based on the class that we have defined
BRunnable br = **new** BRunnable();
- Create a Thread object, and pass the object top its constructor
Thread t = **new** Thread(br);
- Call the start method of the thread object
t.start();

If we have executed this code in the main method, then after they are executed there will be two threads. The first thread executes the main method (any code that comes after the start method is called) and the second thread executes the run method of the BRunnable object.

Execution order

Consider the following example. What order will the characters be printed in?

Example: Main method creating two extra threads

```

1 public static void main(String[] args) {
2     AThread at = new AThread();
3     at.start();
4
5     Runnable br = new Runnable();
6     Thread t = new Thread(br);
7     t.start();
8
9     for (int i = 0; i < 30; i++) {
10        System.out.print('c');
11    }
12 }
```

The answer is that we do not know what order the letters will be printed in. An example of the output is `aaaaaaaaaaaaaaaaaaaaaaaaccccccccccccccccccccbaaaaabbbcbbbbcbbbbcbbbbcbbbbcbbbbcbcccccc`. The order that the different threads are chosen for execution, and the amount of statements that a thread will execute are based on many different factors out of our control. Such as other programs and operating system functions being executed at the same time.

When the actual order that the operations are carried out does not matter, then this is not a problem. However, sometimes we need to impose some sort of order on when certain actions are carried out. If we have some complicated calculation performed in one thread, and another thread that is supposed to perform some operation with the result, then we need some way to make sure that the calculation is complete before we can use the result. This means that we need some mechanisms to allow us to control the execution of threads.

14.4 Controlling Execution

There are a number of methods in the Thread class that can be used to control the execution of threads in different ways. The most useful of these are the methods `join`, `sleep` and `interrupt`.

Join

API: Thread - join

public final void join() **throws** InterruptedException
Waits for this thread to die.

Throws: `InterruptedException` - if any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

The join method allows us to wait for another thread to finish before we continue performing any more calculations. This ensures that the chosen thread is finished before we do something. If we wish to wait for another thread to finish, we must have a reference to that thread.

Figure 14.1 shows how the execution of two threads is changed by the use of the join method. In this example, at point A the main thread creates and starts the thread `ct`. At point B, the main thread executes the join method for the `ct` thread object. This causes the execution in the main thread to stop, and remain stopped. This changes at point C when the thread `ct` has completed its assigned work and finished. At this point the main thread can continue executing.

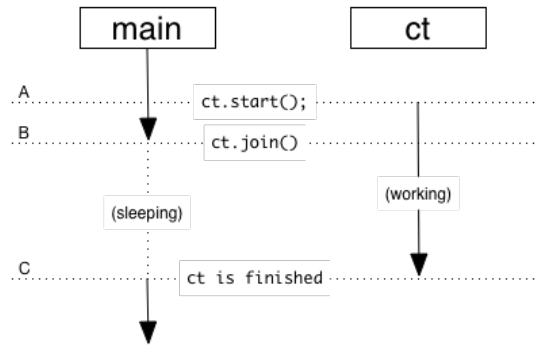


Figure 14.1: How join changes execution of threads

Example: Use of the join method to wait for ct to finish

```

1 public static void main(String[] args) {
2     AThread ct = new AThread();
3     ct.start();
4     // do some other calculations as needed
5     try {
6         ct.join();
7     } catch (InterruptedException e) {
8         System.out.println("main was interrupted");
9     }
10    for (int i = 0; i < 30; i++) {
11        System.out.print('c');
12    }
13 }

```

The example above shows the use of the `join` method. First, the thread is created and started and so begins executing its task. Following, this the main thread would perform some other actions (otherwise there would be no point in having two threads) and once these actions are complete, it needs to wait for `ct` to finish. The `join` method of the thread `ct` is then called by the main thread. This means the the main thread waits until `ct` has finished before continuing. Lastly when `ct` is complete, the rest of the code in the main thread is executed.

This means that whatever order the actions are performed while both threads are finished, the thread `ct` must be complete before the main thread begins printing characters. In this example, this would result in the following output: `a...ac...c` (all `a`'s followed by all `c`'s). If we look at another example, where the main thread is performing some work at the same time as the `ct` thread we will see a different output.

Example: The use of join to make sure c is printed last

```

1 public static void main(String[] args) {
2     AThread ct = new AThread();
3     ct.start();
4     for (int i = 0; i < 30; i++) {
5         System.out.print('d');
6     }
7     try{
8         ct.join();
9     } catch (InterruptedException e){
10        System.out.println("main was interrupted");
11    }
12    for (int i = 0; i < 30; i++) {
13        System.out.print('c');
14    }
15 }

```

In this example, both threads are still performing work at the same time. So the main thread is printing the character 'd', and the `ct` thread is printing the character 'a'. Because both of these are happening at the same time, we cannot know what order the result will be in. However, because the `join` method is called before we begin printing 'c' we can be sure that all c's will be printed last (mix of a's and d's followed by only c's). This is the level of control that can be achieved using the `join` method.

Sleep**API:** Thread - sleep

public static void sleep(long millis) **throws** InterruptedException

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

Parameters: millis - the length of time to sleep in milliseconds

Throws: `IllegalArgumentException` - if the value of millis is negative

`InterruptedException` - if any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

Another way to control the execution of threads is using the `sleep` method. The method `sleep` is a class method in the `Thread` class. This means that we do not need a reference to use the thread and we cannot use it to change how another thread is executed. The method takes a single long as a parameter, this represents a number of milliseconds that the thread should sleep for. The method causes whatever thread that executes it to stop for that amount of time. During this time no code will be executed by this thread, but all other threads will continue as normal.

Figure 14.2 shows two examples of how `sleep` can change the execution of a thread. In the first example (left), the main method calls the `sleep` method at point B (time = 250) with a parameter of 4000. This means that the main method will not execute any code until this time has elapsed. Once the 4000 milliseconds have passed, the main thread continues executing. The use of `Thread.sleep()` in the main thread has no effect on the execution of the thread `ct`, which continues executing as normal and is still executing when the main thread continues.

In the second example, main thread calls the `sleep` method at the same time, but the `ct` thread completes its work and finishes before it continues. Again, the use of `Thread.sleep()` in the main thread has no effect on the execution of the thread `ct`.

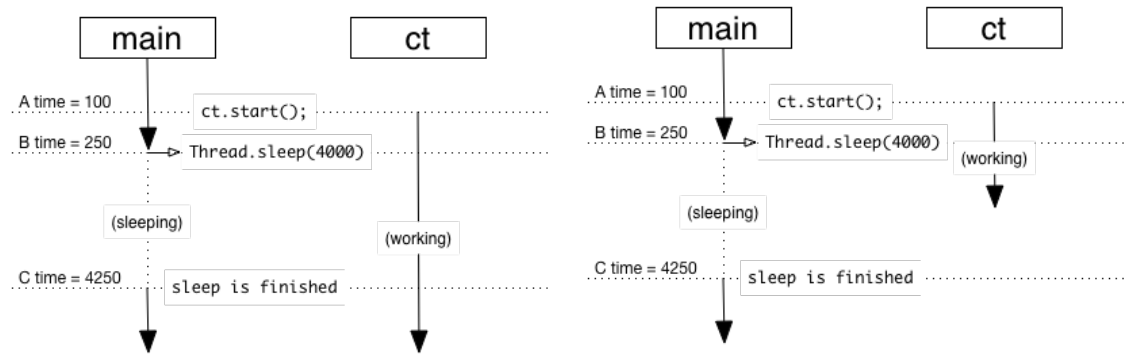


Figure 14.2: Two examples of how sleep changes execution

Example: Use of the sleep method to pause execution of the main thread

```

1 public static void main(String[] args) {
2     AThread ct = new AThread();
3     ct.start();
4     try {
5         Thread.sleep(4000);
6     } catch (InterruptedException e) {
7         System.out.println("main was interrupted while sleeping");
8     }
9     for (int i = 0; i < 30; i++) {
10        System.out.print('c');
11    }
12 }

```

The code above shows the use of the sleep method. Once the method is called, the main thread will sleep for (approximately) 4000 milliseconds. In this example, 4 seconds is a very long time and the thread `ct` is finished long before the main thread continues executing. This might lead to confusion between the two methods, `join` makes this thread wait until the referenced thread is finished (no matter how long or short) and `sleep` makes this thread wait for a defined amount of time. The outcome of using `join` is based on the execution of another thread and the outcome of `sleep` is based only on the parameter passed to the method.

Interrupt

API: Thread - interrupt

public void interrupt()

Interrupts this thread.

Unless the current thread is interrupting itself, which is always permitted, the `checkAccess` method of this thread is invoked, which may cause a `SecurityException` to be thrown. If this thread is blocked in an invocation of the `wait()`, `wait(long)`, or `wait(long, int)` methods of the `Object` class, or of the `join()`, `join(long)`, `join(long, int)`, `sleep(long)`, or `sleep(long, int)`, methods of this class, then its interrupt status will be cleared and it will receive an `InterruptedException`.

If this thread is blocked in an I/O operation upon an `InterruptibleChannel` then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a `ClosedByInterruptException`. If this thread is blocked in a `Selector` then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's wakeup method were invoked. If none of the previous conditions hold then this thread's interrupt status will be set. Interrupting a thread that is not alive need not have any effect.

Throws: `SecurityException` - if the current thread cannot modify this thread

The `interrupt` method sends a signal to a thread to ask it to stop executing. Similar to `join`, we must have a reference to a thread if we wish to interrupt it. One problem with using the `interrupt` method is that the thread may simply ignore the signal and continue executing.

Handling interrupts

This means that when we design a thread to complete some task, we need to check to see if we have been sent the interrupt signal. This is usually done as the condition of a loop. The idea would be that every time we complete a task, before we begin the next task we check the condition. If we have received the interrupt signal, then we exit the loop and stop executing. To do this we use the class method from the `Thread` class `interrupted`. This method returns a boolean value to tell us if we have been interrupted. If the method returns `true` we have been interrupted and should finish executing, if the method returns `false` we have not been interrupted and can continue executing.

Example: A thread that checks if it has been interrupted

```

1 public class InterruptableThread extends Thread {
2     public void run() {
3         while (!Thread.interrupted()) {
4             System.out.println("doing work");
5         }
6     }
7 }

```

The code above uses the binary operator `not` to change the return value of the `interrupted` method. The value returned by the operator is not `true` when the thread has not been interrupted. This allows the while loop to execute as long as the thread has not been interrupted.

Example: Interrupting a thread

```
1 public static void main(String[] args) throws InterruptedException {  
2     InterruptableThread it = new InterruptableThread();  
3     it.start();  
4  
5     Thread.sleep(1000);  
6  
7     it.interrupt();  
8 }
```

The example code above shows how a thread can be interrupted. Here the thread `it` is created and started. The thread `it` has no finishing condition, so it will continue to execute and print "doing work" forever. The main thread sleeps for 1 second, and then it calls the `interrupt` method on the thread `it`. Once received, the thread will continue performing its current task. However, once the task is completed it will check the condition of the loop and discover that it has been interrupted. This means that the condition of the while loop is false (not operator changes it from true) and the loop does not continue. This allows the thread to finish.

14.5 Thread Safety

Threads share the **same data address space**. This means that they can share objects and variables. This means that one thread may change an object in a way that the other is not expecting, it can be difficult to know what the result of this will be.

When variables are shared between threads, we must make sure that they are used in a way that is **safe**. The idea of safety in threads is explained easily with an analogy, consider two trains that must cross the same bridge with only one set of tracks. Without some organisation, a disaster would surely happen the next time that both trains wanted to cross the bridge at nearly the same time.

Synchronisation

Synchronisation is the process of ensuring that these events cannot happen. In computer programming, refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action. The important part here is "sequence of actions". Synchronisation is achieved through the use of **mutual exclusion**. This is another topic that you will study in greater detail in your Operating Systems class.

Mutual exclusion is making sure that a resource can only be used by thread at a time. If we apply it to the analogy above, it means that only one train is allowed to use the bridge at a time. Synchronisation generally uses mutual exclusion to decide the sequence of actions. For example, the easiest solution is that whichever train arrives at the bridge first gets to use it, and the other train must wait.

Lets look at an example of a shared object that could cause problems in a multi threaded application.

Example: Basic class for adding two numbers

```

1 public class Adder {
2     int x, y;
3     public void setXY(int x1, int y1){
4         x = x1;
5         y = y1;
6     }
7     public int sumXY(){
8         return x + y;
9     }
10 }

```

This class only performs a very simple function, but it is designed to represent how shared objects are used in much more complicated classes. The basic function of the methods are to set some state in the object (`setXY`) and then to perform some calculation based on the state of that object.

What happens if we have the same object being used by two thread at the same time? There are different situations to look at so lets have a closer look at the order that the statements might be executed in.

Thread 1	Thread 2
<code>a.setXY(1,2);</code>	<code>a.setXY(3,4);</code>
<code>int w = a.sumXY();</code>	<code>int z = a.sumXY();</code>
<code>System.out.println(w);</code>	<code>System.out.println(z);</code>

1. What happens if thread 1 completes all statements before thread 2 calls `setXY`?
2. What happens if thread 1 calls `setXY` between thread 2 calling `setXY` and `sumXY`?
3. What happens if thread 1 is calling `setXY` while thread 2 is calling `sumXY`?

Lets look at the answers to the three questions above. In the question 1, the answer is pretty easy, thread 1 will print the number 3 and thread 2 will print the number 7. Both threads correctly calculating the result. This is the way that the object is supposed to be used. In question 2, the answer is again pretty easy, thread 1 will print the correct result of 3 and because, the values were set by thread 1 before thread 2 performed its calculations, thread 2 will print the incorrect result of 3.

The last question is the most difficult to answer. The answer will be determined by the order that the individual statements in the methods are executed in. If the addition is performed first, then thread 2 may get the correct answer. However, it is also possible that both assignments will be completed first and that it will get the answer thread 1 was expecting. Lastly, if the addition is performed between the two assignments, we may get an entirely different value based on the first value from thread 1 and the second value from thread 2 giving us the result of 5.

This last result is the most difficult to predict. In much more complicated calculations, there may be no way to determine what the result will be. This is the situation that we are going to learn to prevent.

Synchronised methods

Java allows us to prevent the the last problem by defining methods as synchronised. Only one synchronised method can be used in an object at any time. If another method tries to use a synchronised method while we are executing one, it will have to wait until the method is finished. To declare a method as synchronised, we add the keyword `synchronized` before the return type in the method definition. Note the American English spelling is used for the keyword.

To correctly prevent methods unknown behaviour by method sharing data, we must define all methods that can influence each other as synchronised. Any method that is not synchronised may be used at any time.

Example: The Adder class defined using synchronised methods

```

1 public class Adder {
2     int x, y;
3     public synchronized void setXY(int x1, int y1){
4         x = x1;
5         y = y1;
6     }
7     public synchronized int sumXY(){
8         return x + y;
9     }
10 }
```

As both methods are defined as synchronised, it is not possible to execute both methods at the same time. We had three possible outcomes based on the order of execution of the two threads. In the first, the correct answer was returned. In the second, the incorrect but predictable answer was returned. In the final example, a unknown result was return. It should be noted that now that the methods are synchronised, we can no longer get an unknown result. However, it is still possible that we get the incorrect result.

14.6 Use In our Game

By this point in the progress of the game, we have put in place many of the required pieces to make the game work, but all it does when we start it is show a stationary image on the screen. In this section we are going to add the game engine to our code this will allow us to have events and movement happen in the game. Our game does not need to be multi threaded, but some of the methods we have learned about will be useful in correctly timing our game.

We are going to design the game to update 30 times every second, this will be how often we recalculate where every drawable object should be and also when we draw them on the screen. We will call these two steps update and render

14.7 Practical Exercises

1. Define a class named **PrintNameThread**. This class should extend the **Thread** class. Define a constructor in the class that takes a single string as a parameter and pass it to the superclass. In the run method of this class, the thread should print the name of result of the method **getName** 10 times.
2. Define a class with a main method named **TestThread**. In the main method you should complete the following actions:
 - Construct a **PrintNameThread** object named **threadA** and pass it the parameter "A"
 - Construct a **PrintNameThread** object named **threadB** and pass it the parameter "B"
 - Start both threads
3. Define a class named **PrintNameRunnable**. This class should implement the **Runnable** interface. Define a constructor in the class that takes a single string as a parameter and stores its value as an instance variable called **name**. In the run method of this class, the thread should print the name of value of the variable **name** 10 times.

4. Define a class with a main method named **TestRunnable**. In the main method you should complete the following actions:
 - Construct a **PrintNameRunnable** object named **runA** and pass it the parameter "A"
 - Construct a **PrintNameRunnable** object named **runB** and pass it the parameter "B"
 - Construct a **Thread** object named **threadA** and pass it the parameter **runA**
 - Construct a **Thread** object named **threadB** and pass it the parameter **runB**
 - Start both threads
5. Define a class with a main method named **TestJoin**. In the main method you should complete the following actions:
 - Construct a **PrintNameThread** object named **threadA** and pass it the parameter "A"
 - Construct a **PrintNameThread** object named **threadB** and pass it the parameter "B"
 - Start **threadA**
 - Call the join method to make sure that **threadA** is complete before **threadB** is started
6. Define a class named **PrintSleepThread**. This class should extend the **Thread** class. Define a constructor in the class that takes a single string as a parameter and pass it to the superclass. In the run method of this class, the thread should print the name of result of the method **getName** 10 times. After the name of the thread is printed, the **sleep** method should be called with the parameter of 5.
7. Define a class with a main method named **TestSleep**. In the main method you should complete the following actions:
 - Construct a **PrintSleepThread** object named **threadA** and pass it the parameter "A"
 - Construct a **PrintSleepThread** object named **threadB** and pass it the parameter "B"
 - Start both threads
8. Define a class named **PrimeCheck**. This class should have a single class method named **isPrime**, which takes an integer as a parameter and returns a boolean value indicating if the number is prime or not.
9. Define a class named **PrimeThread**. This class should extend **Thread**. The method should take a single int as a parameter and store this value as an instance variable. In the run method of the class, the **isPrime** method should be called, passing the instance variable as a parameter. If the return value is true, the value of the number should be printed to the screen followed by a space.
10. Define a class with a main method named **TestPrime**. In the main method you should create and start a single **primeThread** object for every number between 1 and 1000.

Chapter 15

Advanced Topic - Generic Programming

Learning Outcomes

After studying this chapter and the related practical work you should...

- ☐ understand the concept of generic programming
- ☐ be able to correctly construct and use generic classes
- ☐ be able to define generic classes and interfaces
- ☐ be able to define generic methods

This chapter introduces a very complicated topic. As there is a great deal of difficult to understand concepts related to generic programming, this chapter will only describe the basics. For further information, see the Java tutorials <https://docs.oracle.com/javase/tutorial/extra/generics/index.html> and <https://docs.oracle.com/javase/tutorial/java/generics/methods.html>.

15.1 Problems With Data Structures

Generally, when we begin designing data structures, we begin with classes that can only be used to store a single type of data. For example, when designing a stack we may implement a stack that works perfectly for integers, but cannot be used for any other type of data. When we apply what we learned in inheritance, we can design the same data structure that can store any type of object (defining the type as `Object`). The Stack interface below gives an example defining a data structure to allow all types.

Example: The Stack interface for all object

```
1 public interface Stack{  
2     public Object top();  
3     public Object pop();  
4     public void push(Object o);  
5     public int size();  
6     public boolean isEmpty();  
7 }
```

Of particular note is the fact the the return type of `top` and `pop` as well as the parameter of `push` are all defined as the type `Object`. Using polymorphism, we can use any other type of object that extends `Object`. As we know, all classes in Java extend `Object`, either directly or indirectly.

This means that if we wish to use a Stack to store `Point` objects, this interface will work. There are some problems with programming in this way. When we construct a stack to store point object, we can also store any other type of object in the same stack. Additionally, when we `pop` an item

from the stack, it will be returned as an object. This means that if we need to use some of the functionality that is unique to the point class, we must typecast the value returned.

Example: Taking a point from the stack and using it

```
1 Stack st = new ArrayStack();
2 // place some point objects on the stack
3 Object o = st.pop();
4 Point p = (Point)o;
5 System.out.println(p.getX() + ", " + p.getY());
```

If an object that was not a point was pushed onto the stack, which is a perfectly correct Java operation, then when we attempt to typecast the returned value there will be an exception in our code. Using polymorphism in this way means that our code is more complicated and it is easier to make mistakes when programming.

There are two possible solutions to this problem. We can create a new copy of the data structure for each type of data that we want to hold. Each copy could only store that type of data and would return values in the correct type. For example, if we needed to use a stack to store point objects, we would create a copy of the interface and implementation classes, perhaps named `PointStack` or something similar, and change the code so that the parameters and return types of the relevant methods are all `Point`.

Example: Example of a copy of the Stack interface for Point objects

```
1 public interface PointStack{
2     public Point top();
3     public Point pop();
4     public void push(Point o);
5     public int size();
6     public boolean isEmpty();
7 }
```

This is not a very good solution. There will be many copies of the same classes and interfaces, each with different return types and parameters. If we want to make even a small change to how the data structure works, we would need to make this change in every class that we have created.

15.2 Generics

The second solution is to use a feature of Java (and most other object-oriented programming languages) called **generics**. Generic programming is a style of programming where algorithms and classes are written in terms of types that will be specified later. Their actual type used is then specified when the object is constructed, rather than when the class is written. This style of programming is particularly useful when applied to creating data structures.

Generics allow us to choose what the type of the parameters, variables and return types will be when we are **using** a data structure, not when we are defining it. This allows us to write a data structure or any other class that can store any type of data. Generic classes do not require us to typecast the values returned or allow different types of objects to be stored in the same object.

The same feature in C++ is called templates. This is because we write code like a template where the important details such as types are filled in later. We can easily use the same template again and again for different types.

Using generics

Before we begin to learn about defining generic classes, we will learn about how generic classes are constructed and used in Java. When an object is constructed based on this class we must also specify what type it will use. When the type is specified, Java fills in the template and we can use the object.

As an example we will look at the `LinkedList` class in the Java API. This is a generic class that can be used to store any type of data. Normally, to construct an object of this class, we would use the code `LinkedList list = new LinkedList();`. This is acceptable code in Java, however it will generally give you warning that you are not specifying the type. When we construct the object this way, it can store any type of data and we would be required to typecast data when we remove it from the list.

Syntax: Constructing an object from a generic class

```
1 ClassName<Type> varName = new ClassName<Type>(params...);
```

The syntax for constructing a generic object is almost the same as constructing any other object. We simply need to add the type inside angle bracket (<>) at the declaration of the variable and at the use of the constructor.

Applying this to the `LinkedList` class, if we wish to create a linked list for storing strings, the code would be `LinkedList<String> strings = new LinkedList<String>();`. If we want to create a linked list that can only store points the code would be `LinkedList<Point> points = new LinkedList<Point>();`.

Restricted types

Once we have constructed a generic object, we are restricted to the correct type for certain parameters. For example, the insertion method of the `strings` object will only accept the type `String`. All other types will cause a compiler error. The insertion methods of the `points` object will only accept the type `Point`. This prevents us from causing errors in our programs at run time by storing the incorrect type of data.

15.3 Defining Generic Classes and Interfaces

To define a class or interface as generic, we need to add a little bit of information to the definition. Generic classes and interfaces can have many type parameters to define different types, however in all of our example we will only be using a single generic type. Defining a class or interface as generic is done by adding angle brackets containing a single upper case character for each generic type we will use after the class or interface name.

For a single generic type, we usually use the letter `T`, for example the definition of a generic stack interface would be `public interface Stack<T>{`. If we are declaring multiple generic parameters, we use a different letter for each generic type and separate them using commas. A typical example of this is the `Map` class, which has a generic type for the key (`K`) and a generic type for the value (`V`). The definition of the `Map` interface in the API is `Interface Map<K,V>{`.

Defining a generic interface

The generic types that we use in the definition can then be used in place of actual types throughout the class or interface. For example in the `Stack` interface, instead of defining the method `public Object top();`, we define it as `public T top();`. This means that when we use the `Stack` interface, the type parameter we supply (`String`, `Point` or any other class) replaces `T` in the method definition.

Example: The stack interface implemented using a generic type

```
1 public interface Stack<T>{  
2     public T top();  
3     public T pop();  
4     public void push(T o);  
5     public int size();  
6     public boolean isEmpty();  
7 }
```

It is important to note that not every type in the interface is changed. Only the types that represent the data we are storing in the stack. It would not make very much sense if the return type of the `size` method changed depending on what type of data we are storing. The size will always be an integer.

Defining a generic class

This example only shows how the interface is changed by the generic type, to have a closer look at how a class with methods and instance variables is changed, we will look at the array based implementation of the stack. First we will look only at the definition of the `ArrayStack` class.

As the stack interface is generic, if we are implementing the interface we must specify the type. For example, if we were defining a array implementation of a stack for only strings we would use the definition `public class ArrayStack implements Stack<String>{`. This would work, but only for strings. Instead we want it to work for different types of data, this means that the `ArrayStack` class must also be generic. This gives us the code `public class ArrayStack<T> implements Stack<T> {`.

Here we can see that there are two sets of angle brackets specifying a generic type. The first states that the array stack class has a generic parameter `T`. The second states that the array class is implementing the `Stack` interface based on the generic type `T`. This basically means that whatever generic type the class is using is also used by the stack interface.

Next we need to define the instance variables that are represented in the stack. As this is the array based version, the first will be an array. However, we do not know what type of data will be stored in the array. In this case, we use the generic type `T`. This means that whatever type is used when the object is constructed, the array will store instances of that type. The second instance variable is an integer that represents the index of the array where the next item is added. It does not matter what type of information is being stored in the array, this will always be an `int`.

Next we need to define the constructor for the class. This constructor is used to specify the size of the array, and therefore the number of items that can be stored in the stack. This is where we run into a problem, it is not possible to create an array of a generic type, e.g. `new T[5];` will not compile. This means that we must come up with some other way to construct the array. The easiest solution is to construct an array of the type `Object`. This can store any type of object, so it can store our generic type. However, to store the constructed array we first need to type cast it to the generic type we are using. This cannot be done for local variables, only instance variables.

Example: Instance variables and constructor of the class

```
1 public class ArrayStack<T> implements Stack<T> {  
2     private T[] items;  
3     private int top;  
4     public ArrayStack(int s){  
5         items = (T[]) new Object[s];  
6         top = 0;  
7     }
```

Implementing the `top` and `pop` methods in the generic array stack class is little different than in the original class. The only change we need to make is to change the return type to the generic type `T`. This makes sense because when the type of data that the class is returning changes, the return types of these methods should match it.

Example: Definition of the `top` and `pop` methods of the class

```
1 public T top() {  
2     return items[top - 1];  
3 }  
4 public T pop() {  
5     T item = items[top - 1];  
6     top--;  
7     return item;  
8 }
```

Lastly, we come to the `push` method. This method is the only way that data can be added to the stack. If we want our stack to store only one type of data, then the parameter of the method is what defines what is allowed to be stored. If we change the type of this to the generic type `T`, then the only type of data that can be added is the same type as the object was constructed to store.

The `size` and `isEmpty` methods do not require any change at all. This is because it does not matter what type of data is being stored, the size will always be an integer value and condition of being empty will always be `true` or `false`.

Example: Definition of the `push` and `size` and `isEmpty` methods

```
1 public void push(T o) {  
2     items[top] = o;  
3     top++;  
4 }  
5 public int size() {  
6     return top;  
7 }  
8 public boolean isEmpty() {  
9     return top == 0;  
10 }  
11 }
```

15.4 Generic Methods

Sometimes it is not necessary for the whole class to be generic. If there are only a small number of methods that deal with generic types, then it may be easier to simply declare a generic method. Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type.

Example: The Util class defining a generic method

```
1 public class Util {  
2     public static <T> T doSomething(T p){  
3         // do something with a generic object  
4         System.out.println(p.getClass().getName());  
5         return p;  
6     }  
7 }
```

The above example, shows a method that takes uses a generic type in a class that is not generic. The method takes the generic parameter, performs some operations and then returns a value of the same type. In this example, all the method does is to print out the the name of the class that was actually passed as a parameter. In order to use this method, the programmer should supply the required type parameters when the method is being called. In both cases, the same method is used to return data that is the same type as the parameter that was passed in. This means that if we pass a String we get a String as a result, and if we pass a point object we get a point object as a result.

Example: Calling the generic class method and passing type parameter

```
1 String result = Util.<String>doSomething("This is a String");  
2 Point location = Util.<Point>doSomething(new Point(1,1));
```

In this example, we see that the type parameter is passed in angle brackets before the name of the method. Passing the generic type parameter also makes sure that the return type of the method is correct.

Type inference

As Java is aware of the type of an object when we use it, it is not actually necessary to specify the generic type parameters for a generic method. Basically, if we have generic types that define the parameters we can pass to the method, then by passing these parameter we are implicitly telling the method what the type parameters are. The compiler is able to infer the types that are needed and we are not required to add the information ourselves. This process is called **type inference**.

Example: Calling the generic class method without passing type parameter

```
1 String result = Util.doSomething("This is a String");  
2 Point location = Util.doSomething(new Point(1,1));
```

The code in this example works exactly the same as in the previous example, even though we have not explicitly told Java what the type parameters are. Java is able to infer that in the first statement the generic type parameter is a String and in the second example the generic type parameter is a Point.

15.5 Reasoning About Generic Types

When we use a generic type in our class, we do not know exactly what type the object will be until the class is used. This means that while we are writing code, we have no knowledge of what methods we can use on the objects of the generic type. This means that the only methods that we can use are those that are available in all classes, i.e. the methods defined in the Object class.

Bounded type parameters

There may be times when you want to restrict the types that can be used as type arguments in a parametrised type. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its upper bound, which in this example is `Number`. Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

This means that where we would normally simply declare a parametrised type as `<T>`, we instead use `<T extends ClassOrInterface>`. This means the the method or class can only be used with types that either are or extend the named class or implement the named interface. Now that we know that the generic type parameter must be of a particular type (or subclass), we can make use of the methods of that class or interface. This means that where previously we were restricted to the methods of objects, the generic type parameters are much more useful once the type is bound.

Example: Generic method using bounded type parameters

```
1 public static <T extends Number, U extends Number> double divide(T x, U
   y) {
2     double z = x.doubleValue() / y.doubleValue();
3     return z;
4 }
```

In this example, the method `divide` is defined to accept type parameters that extend abstract `Number` class. This class is the superclass for all non primitive number representations. In this example, any two types of numbers can be passed as parameters and a double will be returned containing the first divided by the second. Because we have defined the generic type as extending `Number`, we can now use any method that is defined in the number class. This is how we can make use of the method `doubleValue` in the code.

This might seem a trivial method that would never be used, however consider the problem of integer division. This method gets the value of both numbers (no matter what type) represented as a double. These values are then used to perform the calculation meaning that the result will always be non-integer division.

Multiple bounds

It is possible to apply multiple bounds to the same generic type parameter. A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. Multiple bounds are shown by having a list of classes and interfaces separated by the ampersand character (`&`).

Syntax: Defining multiple bounds on a generic type parameter

```
1 <T extends A & B & C>
```

In this definition, it is assumed that `A`, `B` and `C` are all classes or interfaces in Java. Assuming that `A` is a class and `B` and `C` are interfaces, the type that is used must be a subclass of `A` and implement both the interfaces `A` and `B`.

15.6 Why Use Generics

Generic programming is a very useful feature of most modern object-oriented programming languages. However, application programmers (us) very rarely define generic classes. That is because

generics is most useful when developing data structures and most of these have already been provided within the Java libraries. This means that application programmers mostly are required to be able to understand and use the generic classes that are available. But what are the benefits of using generic classes?

There are two main reasons to use generic classes in our applications:

- It enables stronger type checking
- It reduces the use of typecasting

Stronger type checking

When a Java program is being compiled, the compiler compares the type of each parameter in every method against the types that the method is defined to accept. If a parameter is used that is not the same as the parameter in the method definition, then the compiler will show an error.

If we are defining our classes to accept and return `Object`, then the type check will always pass. However, while the program is running there may be a problem because a data structure contained a `String` but we were expecting a `Point`. Replacing this data structure with a generic data structure means that this error will be detected by the compiler, not later when the application is being executed. This allows us to find errors quickly.

For example, if we are using a non-generic stack to store strings, we can push any type of data onto the stack. This means that if we (or another programmer in our group) accidentally push an integer onto the stack there will now be a data type that we did not want in our data structure. When we eventually come to remove the integer, we will attempt to typecast it to a string (because we believe all data in the stack are strings) and the program will throw a `ClassCastException`.

Replacing the stack with a generic stack, allows us to define the type parameter for the stack as `String`. This means that when we (or another programmer in our group) attempt to push an integer onto the stack, we will get a compiler error and the mistake will never appear at runtime.

Reducing the use of typecasting

The use of generic classes means that the return types of our methods will generally be correct. This means that we do not need to typecast objects when they are removed from the data structure that we are using. Typecasting is a useful feature, but it is very easy for a mistake to be made and for this to cause a problem in your program.

Example: Using the non generic stack

```
1 Stack s = new ArrayStack();
2 s.push("S1");
3 String m = (String) s.pop();
```

In this example, every time we remove a string from the stack, it is returned as an object. This means that if we wish to actually use the strings (count characters, search, or get sub strings) we will have to typecast the object that is returned. Every time that a type cast is used, there is potential for the program to fail.

Example: Using the generic stack

```
1 Stack<String> s = new ArrayStack<String>();
2 s.push("S2");
3 String m = s.pop();
```

In this example, there is no typecasting required. Not only is the code a little easier to read, but there is also less potential problems that can happen during execution.

15.7 Generics and Primitive Types

Generics type parameters can only be used with objects. The primitive types that we use, such as `int`, `float`, `double` and `char` are not objects. If we want to have generic class that can store these types of data, we have to use the matching class for that type.

Every primitive type has a class that can be used to represent it and perform certain useful functions. We have already see the use of the `Integer` class to store values such as minimum and maximum values and methods for converting from strings to integers (`parseInt`). These classes are more than just utilities that are useful for performing certain functions, they can also be used to create objects that represent a value of that primitive type. For example `Integer i = new Integer(123);` creates an object that represents the value of the `int` 123.

The classes that represent each of the primitive types are given in the table below:

Primitive type	Class Name
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>

This means that if we wish to use a generic class to represent the primitive type `int`, we would actually declare the object to with the type parameter `Integer`.

Example: Defining a Stack that can store the type `int`

```
1 Stack<Integer> s = new ArrayStack<Integer>();
```

Autoboxing and Unboxing

Now that we can create a generic object for storing primitive types, all we need to do is use it. If the array stack in the example only accepts `Integer` object, then it would seem that we need to create an integer object to add it to the stack.

Example: Creating an integer object from an `int` to add it to the stack

```
1 Stack<Integer> s = new ArrayStack<Integer>();
2 int i = (some calculated computations);
3 s.push(new Integer(i));
```

This added code makes the use of our stack a little more difficult to understand. For this reason, instead we can simply use the `int i` and it will be automatically converted into an `Integer` object for us. This process is called **autoboxing**. This means that we can replace the last line of the example with `s.push(i);`.

Additionally, when an object is returned that represents a primitive value, we can store this value directly in a primitive variable. This process is called **unboxing**. For example, the return type of the array stack (with type parameter `Integer`) is `Integer`, rather than having to use the method `intValue` to return a primitive type we can simply assign the return value to the variable.

Example: Storing returned Integer object as an int

```

1 Stack<Integer> s = new ArrayStack<Integer>();
2 int i = s.pop();

```

This process is done automatically by the compiler every time it is necessary. If a method expects an `int` and the parameter is an `Integer` object, it will be automatically unboxed. If a method expects an `Integer` object and the parameter is an `int`, then the variable will be autoboxed.

15.8 Practical Exercises

1. Define a class named `PrintNameThread`. This class should extend the `Thread` class. Define a constructor in the class that takes a single string as a parameter and pass it to the superclass. In the `run` method of this class, the thread should print the name of result of the method `getName` 10 times.
2. Define a class with a main method named `TestThread`. In the main method you should complete the following actions:
 - Construct a `PrintNameThread` object named `threadA` and pass it the parameter "A"
 - Construct a `PrintNameThread` object named `threadB` and pass it the parameter "B"
 - Start both threads
3. Define a class named `PrintNameRunnable`. This class should implement the `Runnable` interface. Define a constructor in the class that takes a single string as a parameter and stores its value as an instance variable called `name`. In the `run` method of this class, the thread should print the name of value of the variable `name` 10 times.
4. Define a class with a main method named `TestRunnable`. In the main method you should complete the following actions:
 - Construct a `PrintNameRunnable` object named `runA` and pass it the parameter "A"
 - Construct a `PrintNameRunnable` object named `runB` and pass it the parameter "B"
 - Construct a `Thread` object named `threadA` and pass it the parameter `runA`
 - Construct a `Thread` object named `threadB` and pass it the parameter `runB`
 - Start both threads
5. Define a class with a main method named `TestJoin`. In the main method you should complete the following actions:
 - Construct a `PrintNameThread` object named `threadA` and pass it the parameter "A"
 - Construct a `PrintNameThread` object named `threadB` and pass it the parameter "B"
 - Start `threadA`
 - Call the `join` method to make sure that `threadA` is complete before `threadB` is started
6. Define a class named `PrintSleepThread`. This class should extend the `Thread` class. Define a constructor in the class that takes a single string as a parameter and pass it to the superclass. In the `run` method of this class, the thread should print the name of result of the method `getName` 10 times. After the name of the thread is printed, the `sleep` method should be called with the parameter of 5.
7. Define a class with a main method named `TestSleep`. In the main method you should complete the following actions:

- Construct a `PrintSleepThread` object named `threadA` and pass it the parameter "A"
 - Construct a `PrintSleepThread` object named `threadB` and pass it the parameter "B"
 - Start both threads
8. Define a class named `PrimeCheck`. This class should have a single class method named `isPrime`, which takes an integer as a parameter and returns a boolean value indicating if the number is prime or not.
 9. Define a class named `PrimeThread`. This class should extend `Thread`. The method should take a single int as a parameter and store this value as an instance variable. In the `run` method of the class, the `isPrime` method should be called, passing the instance variable as a parameter. If the return value is true, the value of the number should be printed to the screen followed by a space.
 10. Define a class with a main method named `TestPrime`. In the main method you should create and start a single `primeThread` object for every number between 1 and 1000.