UNIVERSITY
OF LONDON

Graphical object-oriented and internet programming in Java
Volume 1

T. Blackwell

CO2220

2009

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London by:

Tim Blackwell

This guide was produced by:

Sarah Rauchas, Department of Computing, Goldsmiths, University of London.

This is one of a series of subject guides published by the University. We regret that due to pressure of work the author is unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

In this and other publications you may see references to the 'University of London International Programmes', which was the name of the University's flexible and distance learning arm until 2018. It is now known simply as the 'University of London', which better reflects the academic award our students are working towards. The change in name will be incorporated into our materials as they are revised.

University of London
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

london.ac.uk

Published by: University of London

Contents

Preface

Introduction

The course is split into two parts, with a separate volume for each part.

- Part I covers object-oriented programming in Java, graphical user interfaces and event-driven systems.
- Part II is concerned with the principles of client-server computing, techniques of interconnectivity in Java and interactive web-based computing systems.

Volume 1, which is this volume, covers the first part of the course. Volume 2 covers Part II of the course.

Aims

The course aims to give students an insight into the object-oriented approach to the design and implementation of software systems. The course also considers specific features of the programming language Java, with particular reference to graphical interfaces and event driven applications.

The second part of the course is intended to give students the necessary background to understand the technical software aspects of how computers communicate across the Internet. Students will be introduced to the underlying principles of client-server computing systems and will gain the required conceptual understanding, knowledge and skills to enable them to produce simple web-based computing systems in Java.

Objectives

1. To re-enforce students' knowledge of object-oriented programming in Java. (Part I)
2. To introduce students to the notion of graphical user interfaces. (Part I)
3. To introduce students to the notion of event-driven systems. (Part I)
4. To teach students the principles of client-server computing. (Part II)
5. To introduce the main techniques for interconnectivity in Java. (Part II)
6. To produce students who are able to develop rudimentary interactive web-based computing systems. (Part II)

Learning outcomes

On completion of this course students should be able to:

1. Analyse and represent problems in the object-oriented programming paradigm. (Part I)
2. Design and implement object-oriented software systems. (Part I)
3. Build an event-driven graphical user interface. (Part I)
4. Explain the main principles for client-server programming. (Part II)
5. Design and implement a rudimentary client side system. (Part II)
6. Design and implement a rudimentary server-side system. (Part II)
7. Integrate their knowledge and skills to produce a rudimentary web-based application. (Part II)

Assessment

Coursework contributes 20 per cent of the final mark on the complete unit. An unseen examination paper will contribute 80 per cent of the final mark.

Important note. The information given above is based on the examination structure used at the time this guide was written. Please note that subject guides may be used for several years. Because of this we strongly advise you to always check the current Regulations for relevant information about the examination. You should also carefully check the rubric/instructions on the paper you actually sit and follow those instructions.

How to use this subject guide

This subject guide is not a self-contained account, but is a companion to the course text *Head First Java 2nd edition* (*HFJ*) by Kathy Sierra and Bert Bates. **It is essential that you obtain this book**.

There are a number of other books, listed in the section below called **Reading**, which expand on a number of topics and you are advised to deepen your understanding by referring to these additional texts where directed.

There are thirteen chapters in this volume. Most chapters are in two parts. The first part is based on a number of readings from *HFJ*. The second part is devoted to programming. Here you will find programming examples and activities based on the material covered in the first part of the chapter. Just two chapters, Chapter 3 Object programming and Chapter 10 Object lifetime, are mainly concerned with conceptual and/or descriptive information, and consequently there is no programming element.

The material in the earlier chapters overlaps with your previous Java course, and you may find that you proceed quickly. Nevertheless you are advised to study these chapters carefully.

In short, chapters are comprised of:

- readings from *HFJ* (not Chapter 3), followed by a summary of the key points from each reading
- a bulleted chapter summary
- programming (not Chapters 3 and 10)

- learning outcomes.

It is important that you read *HFJ* when directed, and then read the commentary to check that you have understood the main points. The commentaries are not sufficient in themselves. You must refer to *HFJ* and you are recommended to engage with the many interesting activities that the authors suggest. The chapter summaries collect together the main points. All chapter summaries are reproduced in the revision chapter. These summaries can be used to ensure that you are on top of the material, and as a revision guide.

The programming sections are an integral part of the course. Aside from the program examples, you will find programming activities. You MUST attempt these activities before reading on. The activities are followed by a programming solution. Please realise that there is rarely a unique solution to a programming exercise, so do not feel disheartened if your solution differs from mine. However you should read my program, and the accompanying commentary, to understand my solution. Moreover, new material, especially concerning Java graphics, will be found in the commentaries.

The CD-ROM The accompanying CD-ROM provides all the source code and compiled programs. Many activities are centred on making a drawing or an animation. You should run these programs before attempting the activity so that you can see what to aim for (but do not peek at the code!).

The cd contains the following:

- an Index which serves to navigate through the folders
- demonstration programs
- source and compiled code for all programming examples and exercises
- source and compiled code for Goo, a special animation package developed for this course
- the Goo API, which is the reference document for the Goo code
- the Java API, which is the reference document for the Java code.

You may wish to develop your programs with an integrated program development environment (IDE) such as Eclipse. It is beyond the scope of this course to show you how to use Eclipse but it is well worth investing some time in learning to use this valuable programming tool yourself. Eclipse can be downloaded for free from `http://www.eclipse.org/`. This guide tells you how to write code in a simple editor and compile and run from the command line. However an IDE such as Eclipse simplifies many programming tasks and greatly helps with debugging.

At the end of each volume, there is a revision guide that summarises the main concepts you should have acquired from each chapter, and also gives you some sample examination papers that can guide some of your study.

Reading

Essential reading

Head First Java (second edition), Kathy Sierra and Bert Bates (Sebastopol, Calif.: O'Reilly 2005) [ISBN 0596009208 (pbk)].

Recommended reading

Java in a Nutshell, David Flanagan(Sebastopol, Calif.: O'Reilly, 2005) [ISBN 0596007736].
Learning Java, Patrick Niemeyer and Jonathon Knudsen (O'Reilly, 2005).
Effective Java (second edition), Joshua Bloch (Upper Saddle River, NJ; Harlow: Addison Wesley, 2008) [ISBN 0321356683 (pbk)].
Java Network Programming, Elliotte Rusty Harold, (Sebastopol, CA; Farnham: O'Reilly, 2005) [ISBN 0596007213 (pbk)].
Java Cookbook, Ian F. Darwin (O'Reilly Media Inc., 2004) [ISBN 0596007019; 978-05960007010].

Notation

`Java keywords, source code, variable names, method names and other source code` are printed in typewriter font. **Filenames, directories and the command line** in bold type. Three dots, . . . , denotes omitted source code in a code excerpt. Concepts and things, where they are distinguishable from their representative Java names (classes, interfaces, method names . . .), are printed in normal type.

Before you do anything else

Insert the CD-ROM into your computer, and open and print the Index for reference. Then open the **demo** folder, and run the programs therein. Some of my favourites are GooDrops, GooByStarlight and MovingPolygon.

You might think that the code for these animations is complicated and unreachable. However you will be writing your first animations within a couple of weeks of starting this course (in fact in Chapter 2).

This is made possible thanks to an animation package, Goo, that I have been using with students here at Goldsmiths for the last few years.

Goo is designed to get you started with animations and drawings quickly; the principles of object programming are illustrated with graphical examples right from the start.

As you progress through this volume you will learn more and more about Java graphics until you will reach the point when you can even write your own Goo! In other words you will know how to develop a tool that enables other developers to code graphics quickly.

Within a few weeks you will know how to do all this.

Goo’d Luck!

Chapter 1

Java without objects

Essential reading

HFJ Chapter 1.

1.1 Introduction

We begin, not at the beginning, but somewhere in the middle.

You have already spent some time studying the Java programming language and writing small programs. This course will considerably extend your skills and knowledge, so that you can write graphical interfaces, animations and link computers on the Internet.

This chapter looks at some basic features of Java, things that you may already know in part. The chapter starts by introducing a Java machine: the machine responsible for interpreting Java bytecode into machine instructions. The Java machine accepts syntactically correct programs; so the programmer has to understand what is legal Java. The syntax of the Java language is therefore briefly explained. Although not necessary for programming, putting names to parts of the language will help us to talk about concepts further on in the course, and will reveal how the language is put together. You will also find out about procedural programming and three features of an alternative paradigm, the object oriented approach.

1.2 Java Machines

Reading: pp. 1–3 of *HFJ*.

If we had a Java Actual Machine then we could talk directly to the machine using the Java language. No-one, however, intends to build such a complicated thing. In common with all high level languages, *source code*, which consists of a sequence of statements in that language, must first be translated into the machine language of your computer.

Java source code is compiled and saved in a **class** file by running the **javac** command from the command line. This class file is made up of Java *bytecode*, the language the Java *Virtual* Machine (*JVM*) understands. A virtual machine is a program, written in the native language of the computer, which emulates a higher level machine. The JVM, which is invoked by the `java` command, sucks in the class file as input and interprets the bytecode into machine instructions. The instructions are executed one by one. Different machines will have different JVMs, but all JVMs

read the same language. As long as a particular platform/operating system (e.g.Windows, Mac OS X, Linux) has a JVM, the class file will run. This is the sense in which Java is platform independent.

You will be learning how the JVM runs your code as you progress through this subject guide. An understanding of the JVM is important for successful Java programming.

1.3 Syntax

Reading: pp. 4–10 of *HFJ*.

The source file has one **class definition**. A class consists of **methods** and **variables**. A method is a list of instructions or **statements** and can be thought of as a function or a procedure, or as a behaviour. The statements must be enclosed by curly braces.

The JVM first searches for a `main` method in the class that you have specified at the command line. `main` will call other methods, and these might call methods again. You can think of each statement as an instruction to the JVM. (In fact the JVM may convert your statement into many machine instructions.)

Classes are grouped into packages. The complete structural hierarchy of a Java program is therefore:
$packages \rightarrow classes \rightarrow methods \rightarrow statements$.

Methods are made from a sequence of statements, each ending with a semi-colon. That's how the JVM separates out our code. We can think of a statement as a single command that is executed by the JVM. Statements themselves are made from *expressions* and an expression is a combination of *operators*, *literals* and *variables*.

A variable has a *name* and a *type*. Types are a very important programming concept. For example, when the bit sequence 0100 0001 enters a register in the CPU, what does it represent? Is it the character 'A' or the integer number 65? One of the jobs of the Java compiler is to check our statements for type correctness. Type hugely reduces programming blunders.

A *literal* (also known as a constant) is a primitive value (a number or a character or a Boolean value) or a `String` or `null`.

Primary expressions are literals or variables. The JVM evaluates a primary expression, returning the value of the literal, or the value of the variable. Primary expressions can be combined into larger, more complex, expressions by using operators. Subexpressions (i.e. parts of complex expressions) are evaluated in order by the JVM, and at each evaluation the value is available for the next subexpression.

For example, `x` and `17` are primary expressions and `x * 17` is an expression. Two primary expressions have been combined by the multiplicative operator `*`. Most operators associate from left to right i.e. the expression `a * b / 5` is equivalent to `(a * b) / 5`. Operators also have an order of precedence. For example, multiplication has a higher precedence than addition so that `a + b * 5` is evaluated by the JVM as `a + (b * 5)`. Parentheses can force an order of evaluation, or can ensure that the right order is carried out if we are uncertain of the rules. The rules of precedence and associativity are given in large tables; see Chapter 1 of *Java in a Nutshell* for example.

The effect of the assignment expression `x = 3` is to place the value '3' in the memory location 'x'. The assignment statement `int x = 3;` declares that the variable with name `x` stores integer values i.e. the *type* of `x` is `int`.

It is important to realise that sub-expressions 'return' a value. It's rather like jotting down an intermediate calculation on a sheet of rough working. So a strange statement like `a = b = 5;`, which means `a = (b = 5)` because = associates from right to left, is equivalent to `a = (value of sub-expression)` where the value of the sub-expression is 5 (i.e. `b` is set to 5 and the value 5 is returned).

1.4 Program flow

Reading: pp. 11–17 of *HFJ*.

A computer excels at doing very simple calculations very quickly. A program is therefore a very large number of simple steps. Each step is processed in order as the machine works its way through the program. It might seem, therefore, that a huge number of statements are needed before the machine can do anything useful. Luckily, many tasks can be subdivided into smaller units which can be repeated and written in a few lines. These are *loops*. For example,

```
set i to 1
loop 100 times:
   print i
   add one to i
end
```

does the same work as 100 consecutive statements

```
print 1
print 2
.
.
.
print 100
```

These two programs excerpts are written in *pseudocode*, an informal textual description of a programming task.

Like all machines, and unlike people, a computer infallibly performs simple, repetitive tasks. It does this by looping through a code block. In order to loop, the code must be able to tell the machine to jump back to the beginning of the block and it must also be able to test for completion. The *while*-loop loops as long as the conditional test is `true`.

At its simplest, a program is a sequence of instructions, with jumps and branches.

Jumps are handled in two ways in Java. Firstly, there are loop statements – the `while`, the `do-while` and the `for`. These cause jumps within a method (i.e between the end and start of the loop statement, and they must be enclosed in a method). Secondly, program control might jump out of a method and into another. The jumped-to method will complete and control will pass back to the jumped-from method, picking up at the next statement.

A program will branch at an `if` or a `switch` statement i.e. one of two or more blocks of code will be executed depending on the outcome of an evaluation.

That's it: just variables to hold data, expressions and operators to manipulate this data; `if`s and loops to control program flow; and methods, classes and packages to organise our code. Well, that's all there is to *procedural* programming.

There is, however, another way of programming, object orientated programming (or more simply, *object programming*), which is what Java is really all about. You *can* use Java for procedural programming too and that's where static methods come in. In order to understand and appreciate object programming you must understand:

- abstract data types
- inheritance
- dynamic binding.

You will spend some time studying these ideas in the course.

Another aspect of object programming, and this relates to program design rather than coding, is decentralisation. Programs—and this is especially true of Part II (in Volume 2) when we look at distributed and networked Java—are actually groups of objects that perform their computations by sending messages to each other, rather than to a central module which co-ordinates program activity.

1.5 The JVM and the Compiler

Reading: pp. 18–19 of *HFJ*.

This fireside chat reveals some important differences between the JVM and the compiler. Make sure you are clear in your own mind on the role each plays in Java development.

1.6 Summary

- Source code, compiled code and the JVM.
- Statements terminate with a semicolon (;).
- Code blocks are defined by a pair of curly braces.
- Assignment operator =.
- Equals operator ==.
- A while loop runs through its block as long as the conditional test is true.
- Execution passes to the code immediately after the loop block if the conditional test is false.
- Conditional branching is handled using `if` and `if/else`.
- String arrays.

1.7 Programming

The original Eliza is a computer psychotherapist. You can find out about Eliza and its variants by googling "Eliza computer therapist". A session between Eliza and a client may run something like this:

```
C: I am feeling sad
E: This is a common problem
C: Maybe, but that's not helping
E: Have you always felt this way?
C: No
E: Have you felt this way for long?
C: No, just recently
E: Please continue
C: It started about two months ago
E: Don't worry you will start to feel better soon.
C: Oh
E: That will be 100 pounds. Goodbye.
```

Learning activity

Write a procedural Eliza program based on the example dialogue above.

You will need to know how to set up a package-source code file structure, and how to compile and run programs from the command line, even if you are using an IDE such as Eclipse.

Source files of Java classes must be saved in files with the same name as the class name, followed by the extension **.java**. These files must be placed in a directory with the same name as the declared package. Follow this procedure and adapt it for all your programming on this course:

1. Make a directory named 'simplejava' and place it, for example, in your 'Computing220' directory, or wherever you wish to keep your programs.
2. Open a text editor and type the code for `Eliza`. Save this file as **Eliza.java** in **Computing220/simplejava**.
3. Open a terminal (command-line prompt) in **Computing220**.
4. Type **javac simplejava/Eliza.java** to compile your program. Check to make sure **Eliza.class** has been created.
5. Run the program by typing **java simplejava/Eliza**

1.8 Eliza

```
package simplejava; // A

import java.util.Scanner; // B

public class Eliza {

public static void main(String[] args) { // C

Scanner scanner = new Scanner(System.in); // D

String[] lines = { "Why do you feel that way?", // E
"Have you felt this way for long?",
"Have you always felt like this?",
"Do you think that other people feel like this too?",
"This is a common problem",
"Please continue"
 };

System.out.println("Hello!"); // F

int i = 0; // G
while(i < 10){ // H

scanner.nextLine(); // I
int randomInt = (int)(Math.random() * lines.length); // J
System.out.println(lines[randomInt]); // K

i = i + 1; // L
}
System.out.println("That will be 100. Goodbye"); // M
}
}
```

Our Eliza is not very intelligent, but she does illustrate the procedural style of programming.

All programs, including those that you write, should be commented. Comments help a reader – and the writer at a later date – understand what the program does, and why. Normally comments are written directly in the program; however since there is rather more to say about the code than usual, the comments follow below.

A. Java classes are grouped together in packages. The source files must be saved in a directory of the same name.

B. The Java system includes a vast library of useful classes. The compiler needs to find all the class files used by the program. This line tells the compiler that this class will refer to the `Scanner` class, which is in a package called `java.util`

C. The `main` method is the point of entry for all Java programs. This method has a complicated *signature*, containing the *modifiers* `public` and `static` and the *return type* `void`.

`void` declares the return type of `main` to the compiler. The return type must be a single value – for example the final number of a calculation. This number is 'returned' to the calling method. We shall see exactly what this means shortly. For now, note that `main` does not return anything to the calling program, and this is denoted by the keyword `void`. Methods must declare a return type, even if nothing is returned! (One exception is the constructor, which is a special type of method invoked by the use of `new`, which must never be declared with any return type, not even void.)

`public` and `static` are *modifiers*. One of the modifiers – in this case `public` – specifies the *visibility* of the method. The visibility of a method defines access to the method from other classes. `public` gives unrestricted access. The modifier `static` tells us that this method is a *class method*. If `static` is omitted, the method is an `instance` method. We shall see what these mean in due course.

`main` is the method name, and `main`'s `parameter list` is enclosed in parentheses. The caller for `main` is actually the JVM.

D. The `new` keyword tells the JVM to construct a `Scanner` object in memory. This object is referenced by the variable `scanner`. What exactly this means will only become evident as the course progresses, but for now regard the `scanner` variable as a 'remote control' on the `Scanner` object.

E. An array of `String`s is populated with a few phrases.

F. The standard Java idiom for printing to the command line. `System.out` is a variable of `java.lang.System` (this class does not need to be declared in a package statement; the `java.lang` package is so fundamental that the compiler always imports this for you).

G. An integer variable is declared and initialised to one. This variable will serve as a counter in the `while` loop.

H. The `while` loop. The expression in parentheses is evaluated at the start of each iteration through the loop's block. If the `boolean` value of the expression is `true` then the loop continues; otherwise the JVM skips to the first statement after the `while`'s closing brace, if there is one.

I. Ask the `scanner` object to read a line typed at the terminal. It's rather like pointing the remote control at the scanner object and pressing the nextLine button.

J. Generate a random index into the `lines` array. The call to `Math.random()` produces a 'random' number between zero and just less than one. The `(int)` operation converts a double value to an integer value, by cutting off the decimal places. This is an example of what java calls a *cast*.

K. Print the random line.

L. increment the loop counter by one.

M. The loop terminates when `i` reaches the value 9 (i.e. 10 iterations in total) and Eliza bids you goodbye.

1.9 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- understand the meaning of the following special terms: *source code, Java Virtual Machine (JVM), bytecode, class definition, method, variable, statement, expression, operator, literal, variable name, variable type, primitive value, literal, association and precedence, loops, jumps, branches, pseudocode, procedural programming*
- be able to write a simple procedural Java program contained in `main` and using the language constructs you met in Chapter 1
- set up a package structure for your Java projects and compile and run java programs from the command line.

Chapter 2

Objects

Essential reading

HFJ Chapter 2.

2.1 Introduction

Object programming is is not just programming with objects. The so-called object oriented (OO) approach to program design uses some special features of the language. Object programming is a paradigm, and we begin our explanation in this chapter.

2.2 A first encounter with inheritance

Reading: pp. 27–33 of *HFJ*.

Subclasses are more specific versions of their more abstract superclass. What is a shape? Is it a triangle, or a square, or a circle? Have you ever seen a pure shape? `Shape` abstracts the common behaviour of actual shapes such as circles and squares; in this case `rotate` and `playSound`. Object programmers say that subclasses *inherit* the attributes and behaviour of their superclass. If a circle object is asked to rotate itself, the `rotate` method in `Shape` is called. This same method is also called if a square is asked to rotate.

However an amoeba has rather different behaviour. An `Amoeba` object has its own `rotate` and `playSound` methods. The `Amoeba` class *overrides* these two superclass methods.

Let's consider another example. Suppose we are asked to produce an animation of some falling drops. Some drops fall quickly; some are grey and some are red.

One way forward would be to write separate classes for `Drop`, `RedDrop` and `GreyDrop` code `move()` and `draw` methods for each class. However drops, irrespective of their colour, move in a similar way, and we would have identical code in several class definitions.

Later we may add different kinds of drops to our animation. These new drops have a similar appearance to standard drops, but they move differently (for example they wobble from side to side as they fall). After a while we notice that many classes have identical draw methods and many other classes have identical move methods. This makes code update very tedious and error prone. Suppose we wish at a later stage to

add code to `draw` in order to make the image appear smoother: we would need to hunt through each class and change every draw method. This is not an object solution.

What we need is to hold all the common code in one place, so that changes can be made to one code block only. And this is where the power of object programming really comes in.

2.3 Classes and their objects

Reading: pp. 34–37 of *HFJ*.

An object has *state*; this is what it knows about itself. In other words, state is the current value of the properties of the object. Properties are coded as instance variables. For a drop we might have:

Drop	
Properties	Instance variables
position	xpos, ypos
velocity	xvel, yvel
size	size

An object can also do things. It has behaviour–instance methods. Continuing with the drop example:

Drop	
Behaviour	Instance methods
movement	move()
appearance	draw()

Putting the two tables together into one gives the following class design:

`Drop`

`int xpos`
`int ypos`
`int xvel`
`int size`

`draw()` `move()`

(This representation of a class as a class box will be familiar to you from your study of UML in CIS226.)

The class, when compiled, tells the JVM how to make objects, what state these objects have, and what messages (methods calls) they respond to. The class serves as a design blueprint. There is only one `Drop` class, but potentially hundreds of `Drop` objects.

2.4 A simple Java application

Reading: pp. 38–40 of *HFJ*, omit *Java takes out the garbage* for now.

This application has three classes; a game class, a player class, and a game launcher. The game launcher is a very simple class with just one method, a `main`. The launcher makes a game object ('instantiates' i.e. makes an **instance** of the class), which in turn makes three player objects. Notice how the application, when launched, consists of four objects and the game itself is enacted by the objects in communication.

2.5 The garbage collectible heap

Reading: pg. 40 of *HFJ*, *Java takes out the garbage*.
Reading: pg. 41 of *HFJ*.

Objects are created in a section of memory known as the *(garbage-collectible) heap*, or *heap* for short. The JVM allocates exactly enough space on the heap to store the object. Later, if the object is no longer used by the program (the object is 'garbage'), the JVM reclaims this storage and liberates memory for new objects. The JVM automatically maintains memory, unlike the situation in some languages where the programmer has to proactively allocate and de-allocate storage space. This important topic is covered in Chapter 10 of the subject guide.

2.6 Summary

- Class boxes show instance variables and methods.
- Object programming lets you extend a program without having to touch previously-tested code.
- All Java code is defined in a class.
- A class describes how to make an object of that class time. A class is like a blueprint.
- An object knows about things and does things.
- Things an object knows are called instance variables. They represent the state of that object.
- Things an object does are called methods. They represent the behaviour of an object.
- When you create a class, you may also wish to create a separate test class which you'll use to create objects of your new class type.
- `main` can be used as a launcher for your application, and as a class tester.
- A class can inherit instance variables and methods from a more abstract superclass.
- At runtime a Java program is nothing more than objects 'talking' to other objects.
- Objects are placed on the garbage-collectible heap; the garbage collector clears objects away from the heap when they are no longer needed by the program.

2.7 Programming

You will now implement the drops application, and in doing so take your first look at Java graphics. This is a big and complicated subject, so I have made life easier for

you by supplying a drawing and animation package, `Goo`. This package contains some of the more difficult graphics code; later you will learn how to write such a package from scratch.

Hold on tight: very soon you will be creating your own drawings and animations!

Learning activity

Write a `SimpleDrop` class based on the design box:

SimpleDrop
int xpos int ypos int xvel int yvel int size
draw() move()

Make a **simpleobjects** directory alongside your **simplejava** directory and save `SimpleDrop` as **CIS220/simpleobjects/Box.java**. You will not yet be able to add code to `draw()` so that the drop is drawn to the window, but try and write code so that a call to `move()` causes the drop to fall by an amount that is determined by the velocity.

Open the command line in **CIS220** and compile `SimpleDrop`:

javac simpleobjects/SimpleDrop.java.

Correct any errors.

2.8 SimpleDrop

```
package simpleobjects; // A

import java.awt.Color; // B
import java.awt.Graphics;

public class SimpleDrop {

  int xpos, ypos, xvel, yvel, size; // C

  public SimpleDrop(int x, int y, int vx, int vy, int sz){ // D

    xpos = x;
    ypos = y;
    xvel = vx;
    yvel = vy;
    size = sz;
  }

  public void move(int width, int height){ // E

    xpos = xpos + xvel;
    ypos = ypos + yvel;
  }

  public void draw(Graphics g){ // F

    g.setColor(Color.GRAY);
    g.fillOval(xpos, ypos, size, size);
  }
}
```

A. This class is declared as a member of `simpleobjects`.

B. Two classes are imported from `java.awt`. This is one of the two fundamental graphics packages (the other is `javax.swing`).

C. There are five instance variables that specify the state of each instance of this class.

D. Every class must have a constructor (some have several constructors). The constructor tells the JVM how to make the actual object. Constructors have the same name as the class; they are like methods in the sense that they receive values, but, unlike methods, they do not return any value. `SimpleDrop`'s constructor receives five values which are known locally as `x, y, vx, vy` and `size`.

   Classes can be considered as blueprints for objects. In this case the JVM builds an *instance* of the class by making a `SimpleDrop` object. It does this by setting aside some memory space in RAM and creating five integer instance variables, `xpos, ypos, xvel, yvel and size`, each initialised to the values of the method *parameters* `x, y, xv, yv, sz`.

E. A straightforward implementation of `move`. Every time `move` is called, `xpos` and `ypos` are updated by adding the `x` and `y` velocities. The width and height of the drawing window are passed as parameters, although they are not used in this implementation.

F. The draw method. There is a single parameter in the method argument, a `java.awt.Graphics` reference variable, `g`. Methods can be called on the `Graphics` object, kindly supplied by the JVM, by using the dot operator on the graphics variable. The Java graphics system calls your `draw` and performs the actions that you specify.

   In this case a message setColor is sent to the graphics object. The argument of `setColor` is a variable known as `Color.GRAY` (known to the system, but not defined in your class).

   `fillOval` draws an oval. The method call sends the values of four variables. You can find out what to send `fillOval` by referring to the Java API (on the CD-ROM, or download from `http://java.sun.com`) for the `Graphics` class.

   We find this information:

```
fillOval

public abstract void fillOval(int x,
                              int y,
                              int width,
                              int height)

Fills an oval bounded by the specified rectangle with the current color.

Parameters:
  x - the x coordinate of the upper left corner of the oval to be filled.
  y - the y coordinate of the upper left corner of the oval to be filled.
  width - the width of the oval to be filled.
  height - the height of the oval to be filled.
```

You should also look up `java.awt.Color` to see what other colours are available.

Learning activity

Copy the `goo` package from the CD-ROM and paste in your **CIS220** directory alongside the **simplejava** and **simpleobjects** directories.

Study the next program, GooDrop, and type it into an editor. Save in **simpleobjects** and compile.

2.9 GooDrop

```
package simpleobjects;

import goo.Goo; // A
import java.awt.Graphics;

public class GooDrop extends Goo { // B

  SimpleDrop drop;

  public GooDrop(int width, int height) { // C

    super(width, height);

    int xpos = width / 2;
    int ypos = 0;
    int xvel = 0;
    int yvel = 1;
    int size = 10;

    drop = new SimpleDrop(xpos, ypos, xvel, yvel, size);
  }

  public void draw(Graphics g) { // D

    drop.move(getWidth(), getHeight());
    drop.draw(g);
  }
}
```

A. `goo.Goo` is not in the Java library; it's in your library! The compiler will search for your own library classes in any directories that lie below the current directory (i.e. where the compiler is launched). It should find `Goo`, if you have pasted **goo** in the correct place.

B. `GooDrop` is declared as a subclass of `Goo` with the `extends` keyword. `Goo` is an animation program. A `Goo` object sets up a window and then calls its own `draw` method at a fixed number of times per second (the frame rate). However, `GooDrop` overrides `Goo`'s `draw`, and the JVM executes the code in `GooDrop`'s `draw` instead.

C. The constructor. The first line calls the superclass constructor and makes a `Goo` object. This uses the special `super` syntax. Don't worry about this now, we'll have more to say on this topic later on. However what you do need to know is that `width` and `height` are the dimensions of the drawing window. The `Goo` object does the hard work of setting up a window of that size.The final line of the constructor block creates a `SimpleDrop` and points the instance variable `drop` at the new `SimpleDrop` object.

D. The overridden draw method. This method is called (for example) 50 times per second. The method parameter `g` is a reference to a `Graphics` object. This object has already been created by the Java graphics system, and it contains all the state and behaviour needed to perform actual rendering (i.e. drawing). In other words your program can instigate drawing by sending messages to the `Graphics` object. You do this by calling methods with the dot operator on `g`. GooDrop's draw relays the

message to the SimpleDrop object. The Graphics reference is passed as a parameter to the SimpleDrop's draw.

Notice that the height and width of the window are obtained by calling getWidth and getHeight. These methods are not defined in GooDrop so the compiler looks for definitions in the superclass, Goo. Goo is able to determine the width and height dynamically i.e. even if the window has been resized. Window width and height might have been stored as instance variables in GooDrop and used by move, but window resizing would not then be taken into account.

Learning activity

Write an application (this is what *HFJ* calls a launcher) class, `GooDropApp` which makes a GooDrop object of width 800 and height 500 pixels. The animation can be stated by calling `go` on your GooDrop object.

Open the command line in **CIS220**, compile `GooDropApp` and run by typing **java simpleobjects/GooDropLauncher**.

2.10 GooDrop application

```
package simpleobjects;

public class GooDropApp {

  public static void main(String[] args) {

    int width = 800;
    int height = 500;
    GooDrop gd = new GooDrop(width, height);
    gd.smooth();
    gd.go();
  }
}
```

The application code is quite simple; a `GooDrop` variable `gd` is initialised to point to a new `GooDrop` object. Two methods are then called on `gd`, `smooth` and `go`. GooDrop does not define these methods; the JVM passes on the call to the superclass `Goo` object. You will find definitions for these methods in `Goo.java`.

(You are not expected to have known about `smooth`. This call tells the Java graphics to apply an anti-aliasing algorithm so that slanting straight lines appear less jagged.)

`go` starts the animation. The Goo object enters an eternal loop; the call never returns and main never reaches its closing right brace. `draw` is called many times a second. At each call, the drop is drawn at a slightly different position, giving an impression of movement.

Learning activity

One drawback of `SimpleDrop` is that the drop disappears from the bottom of the drawing window. Add code to `SimpleDrop` so that the drop reappears at the top of the window and save your edited code as `Drop.java`. Remember to change the class name to `Drop`. Modify `GooDrop` so that your animation runs with `Drop` rather than `SimpleDrop`.

2.11 Drop

```
package simpleobjects;

import java.awt.Color;
import java.awt.Graphics;

public class Drop {

  int xpos, ypos, xvel, yvel, size;

  public Drop(int x, int y, int vx, int vy, int sz){

    xpos = x;
    ypos = y;
    xvel = vx;
    yvel = vy;
    size = sz;
  }

  public void move(int width, int height){

    xpos = xpos + xvel;
    ypos = ypos + yvel;

    if (ypos > height) {

      ypos = 0;
      xpos = (int)(Math.random() * width);
    }
  }

  public void draw(Graphics g){

    g.setColor(Color.GRAY);
    g.fillOval(xpos, ypos, size, size);
  }
}
```

A conditional block has been added to `move`. The origin of any computer graphics co-ordinate system is at the top left of the window or screen, with *y* increasing downwards. So the conditional expression `ypos > height` returns `true` if the drop leaves the window. As a consequence the drop is repositioned at a random position at the top of the window.

Learning activity

Write a subclass, `RedDrop extends Drop`, which appears red rather than grey.

Modify `GooDrop` accordingly.

2.12 RedDrop

```
package simpleobjects;

import java.awt.Color;
import java.awt.Graphics;

public class RedDrop extends Drop{

  Color color = Color.RED;

  public RedDrop(int xpos, int ypos, int xvel, int yvel, int size
      ){

    super(xpos, ypos, xvel, yvel, size);
  }

  public void draw(Graphics g){

    g.setColor(color);
    g.fillOval(xpos, ypos, size, size);
  }
}
```

```
package simpleobjects;

import goo.Goo;

import java.awt.Graphics;

public class GooDrop2 extends Goo {

  Drop drop;

  public GooDrop2(int width, int height) {

    super(width, height);

    int xpos = width / 2;
    int ypos = 0;
    int xvel = 0;
    int yvel = 1;
    int size = 10;

    drop = new RedDrop(xpos, ypos, xvel, yvel, size);
  }

  public void draw(Graphics g) {

    drop.move(getWidth(), getHeight());
    drop.draw(g);
  }
}
```

In this solution, `RedDrop` has a single instance variable, `color`, initialised to `Color.RED`. `RedDrop`'s constructor calls the superclass constructor using the `super`

syntax. We saw a similar call in `GooDrop`'s constructor. A `Drop` object is created; you can imagine this lives 'inside' the `RedDrop` object. The pictures on pp. 250–251 of *HFJ* illustrate the general idea.

`GooDrop2` shows the modification to `GooDrop`. A `RedDrop` object is created, and assigned to a `Drop` variable. This is allowed in Java and in fact is a standard technique of object programming: a superclass variable can point to a subclass object. This is an example of polymorphism, one of the distinguishing features of object programming.

Learning activity

Subclass `Drop` once more to define a drop which wobbles from side to side as it falls.

2.13 WobblyDrop

```
package simpleobjects;

public class WobblyDrop extends Drop {

  public WobblyDrop(int xpos, int ypos, int xvel, int yvel, int
      size) {

    super(xpos, ypos, xvel, yvel, size);
  }

  public void move(int width, int height) {

    xpos = xpos + (int)(4 * (Math.random() - 0.5));
    ypos = ypos + yvel;

    if (ypos > height) {

      ypos = 0;
      xpos = (int) (Math.random() * width);
    }
  }
}
```

This time we override `move` but not draw. The wobble is performed by generating a random integer between -2 and 2 and adding this to the x position of the drop.

Learning activity

Write an application that has all three types of drops. The drops could change type when they reappear at the top of the window.

2.14 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- describe the two main Java graphics packages
- find out about the classes in these packages by referring to the API
- construct a class hierarchy
- create an animation or a drawing program by extending `Goo`
- fill ovals
- set and change colour.

Chapter 3

Object programming

Essential reading

There is no specific reading for this Chapter. Some explanations of the topics contained in this short essay are scattered around *Head First Java* (try looking things up in the index). You might also benefit from glancing at a Software Engineering book such as Roger Pressman's *Software Engineering*, published by McGrawHill.

3.1 Introduction

Object Oriented programming is characterised by three distinguishing features: abstract data types, inheritance and dynamic binding.[1] Data hiding and polymorphism are also closely related to the object approach.

Abstraction, to programmers, means trimming away unnecessary detail. A thing is represented by only its most significant attributes. In many ways it is like modelling; an abstraction is frequently a software model of an actual entity. The *abstract data type* is a software module that includes data and operations on that data. Java enables us to define our own ADTs (i.e. classes). The important aspect of an ADT is that the internal representation of the entity is hidden from the program units (the *clients*) that may use it. So GooDrop can ask a Drop to draw itself, and to move, but it does not need to know how the drawing is made or how the movement is calculated. `Drop` is an abstraction; real drops have many attributes determined by their chemical and physical makeup, but for our purpose we only need to know position and size. It is an ADT; clients interface with `Box` objects by calling the 'visible' methods, `draw` and `move`.

Data hiding. Furthermore the internal representation of the Drop is also irrelevant. In this case, `Drop` stores top left corner coordinates and width and height. It could just as easily store the central coordinates and the lengths of the major and minor axes of the ellipse. The details of the representation should be hidden from the clients, so client program units interact with a `Drop` object only by the declared interface, namely the public methods. This means that we are free to change the internal representation of a Drop, and the details of how the methods work, without requiring all the clients to also change their code.

Inheritance allows a programmer to modify an ADT if a new requirement demands a slightly different behaviour. Rather than define new top-level ADT's for each new requirement (a wobbly drop, a red drop, . . .), descendant classes can inherit the behaviours of their parent class yet *override* some details of behaviour where necessary. This means that code can be *re-used*, rather than redefined in several places.

[1] Sebesta, R., *Concepts of Programming Languages.* (Addison Wesley, 2009).

Are there any drawbacks to inheritance? One problem is that the class hierarchy introduces a dependency between program modules. The subclasses depend on their superclasses for some of their method definitions. This restricts the changes that can be made to these superclasses. And this dependency in turn makes code difficult to read.

Polymorphism means having many shapes. In a programming context it means that an object could appear to have many types. Similarly a variable could, at different times, reference objects of different types. Consider:

```
Drop drop = new Drop(200, 200, 0, 10, 10);
RedDrop redDrop = new RedDrop(50, 75, 0, 12, 10);
...
drop = redDrop; \\  drop now points to the RedDrop object
...
drop.move(g);
```

drop is a polymorphic variable because it references both a Drop and later a RedDrop. The RedDrop object is polymorphic because it is referenced by a Drop and also by a RedDrop variable. A RedDrop object might appear in some contexts as a Drop, and in others as a RedDrop.

Dynamic binding. The compiler performs *static* type checking, i.e. it checks that each statement is syntactically correct. `drop.move(g)` *is* syntactically correct because the class of the variable is `Drop` and `Drop` does declare and define `move()`, even though drop points to a RedDrop object.

The compiler generates code for method calls whenever it can; but this is not possible when methods can be overridden and the type of the receiving object is not known at compilation. Instead, the appropriate method is dynamically chosen at runtime.

In the following code, the compiler cannot know the type of the Drop object referred to by `drop` without actually running the program. However, at runtime, the JVM will decide which draw method to execute based on `drop`'s class definition, and any superclasses it may have.

```
Graphics g;
....
public void draw(Drop drop){

    drop.draw(g);
}
```

Learning activity

Explain the meaning of the following concepts in your own words:

- abstraction
- abstract data type
- clients of a class
- data hiding
- inheritance
- dynamic binding
- polymorphism.

In each case you should provide code excerpts to illustrate your explanation.

3.2 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should have an understanding of the following concepts:

- abstraction
- Abstract Data Type
- clients of a class
- data hiding
- inheritance
- dynamic binding
- polymorphism.

Chapter 4

Reference types

Essential reading

HFJ Chapter 3.

4.1 Introduction

Variables have a name, a type and a *value*. There are two kinds of type: primitive types and reference types. The values of primitive types are quite easy to understand. The value of an `int` variable `i`, after initialisation by the statement `int i = 3;` is, well, 3. But what is the value of `drop` after initialisation `Drop drop = new Drop()`?

To help us understand how reference types such as `Drop` are used in Java, we shall use a diagrammatic representation of the JVM: a memory diagram. This memory diagram will help to visualise the connection between a variable and its object, and will explain some of the strange things that happen when object references are passed to methods. Memory diagrams will help us to understand inheritance and other important object techniques.

4.2 Primitive Type

Reading: pp. 49–53 of *HFJ*.

Figure 4.1 below shows part of your computer's RAM, with four words (a word is two bytes) at addresses 1000–1004. You can imagine that the JVM's portion of RAM is laid out as a grid.

Symbols are much easier to use (by humans) than raw addresses. The JVM builds a symbol table, mapping symbols to addresses (which are easier for machines to use).

```
Symbol        Address
--------------------
...
i             1000
j             1002
...
--------------------
```

The command `java MyProgram` starts the JVM. The JVM asks the Operating System (OS) for a block of memory. The symbol table, variable values, intermediate values

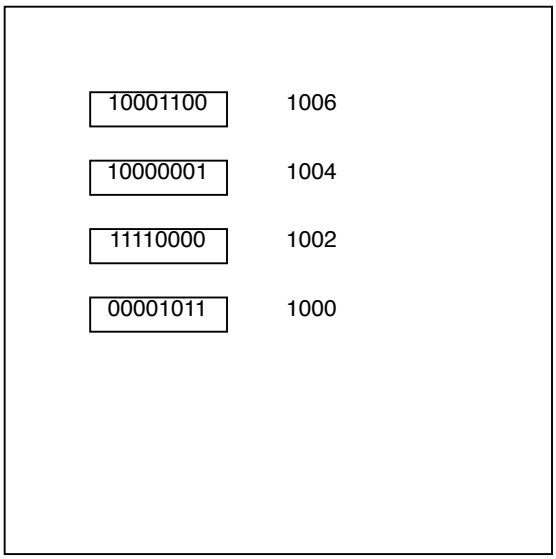

Figure 4.1: A portion of RAM showing four consecutive words of memory at addresses 1000, 1002, 1004 and 1006

used in expressions, etc. are all put in this block. The block also contains two other important sections: the stack (or stacks) and the heap.

We see from Figure 4.1 that the word at memory at address 1000 is 00001011. But what does the binary number 00001011 mean? By this we mean "what does it *represent*"? At a physical level, the number represents a state of some logic gates or at an electronic level, the state of some circuitry. At a higher (more abstract) level, the number is a value in a programming language. 00001011 might be an integer for example.

The representation is specified in a computer language as a type. Type helps us to program meaningfully. Type is checked by the compiler, and helps us to avoid some programming mistakes. Type also tells the JVM how to handle the value.

A variable has a name and a type. When a variable is declared, as in:

```
int luckyNumber;
```

the JVM sets aside space in memory to hold an integer value and puts `i` in the symbol table.

The statement:

```
int luckyNumber = 11;
```

declares and initialises a variable. Now the memory address `luckyNumber` contains the bit sequence 00001011.

Integers, characters, floats, etc. are primitive types. The value of a primitive type is just what we would expect. Programming with primitives is very limiting because it is often useful to bundle data together in an aggregate type. All related data can then be referred to by a single name (i.e. a single symbol).

These aggregates, or objects, are reference types. Objects, though, are rather more than just data structures. Objects have methods – these tell the JVM how to manipulate the data.

4.3 Reference Types

Reading: pp. 53–56 of *HFJ*.

Suppose the variable `dogName` is a `String`, declared and initialised in this statement:

```
String dogName = "Bongo";
```

`"Bongo"` is an object, so it looks as if `dogName` is an object variable, just as `luckyNumber` is a primitive variable in the example above. In fact `dogName` does not hold the object in the way that `luckyNumber` holds the primitive value. `dogName` is a *reference* variable. The value of a reference type is the address where the object lives in memory. The object itself is the word "Bongo"; `dogName` refers to (points at) this object.

Let us look at another example:

```
Box b = new Box();
```

and the memory diagram for this line of code, which is in Figure 4.2.

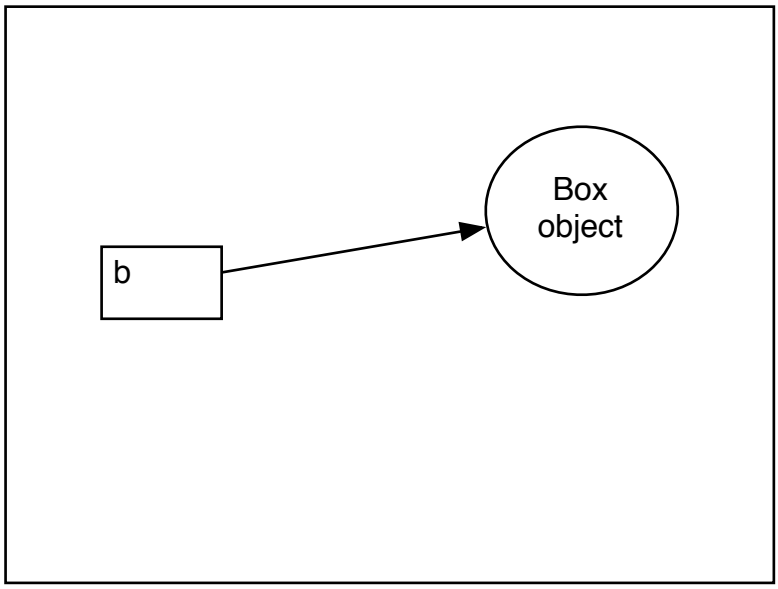

Figure 4.2: Memory diagram showing a reference variable `b` pointing at a `Box` object.

The reference is shown on the left in a box, the object on the right as a blob. They are connected by an arrow to show the relationship 'b refers to `Box`'. You can imagine a grid of memory locations beneath and surrounding the box and the blob; or you can just regard the diagram as an abstract picture of the memory. In any case,

we shall call this type of diagram a *memory diagram*. Memory diagrams help us to explain and understand the workings of objects and references.

The JVM divides the memory into two parts: a place for the stack (or stacks), and the heap. The heap is an unstructured area of memory. Objects are created on the heap with just enough space to hold their instance variables, but they do not lie in any particular position.

Local variables live on the stack. Unlike the heap, the stack is very structured. We can give an idea of the relationship between the stack and the heap by considering a memory diagram for this block of code:

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

        // 3 local variables
        Box b1 = new Box();
        Box b2 = new Box();
        Box b3 = new Box();
    }
```

Three local (to `main`) variables, `b1, b2, b3` are declared, three `Box` objects are instantiated and the references between variables and objects are set-up. The memory diagram of the JVM just after the last statement is shown in Figure 4.3.

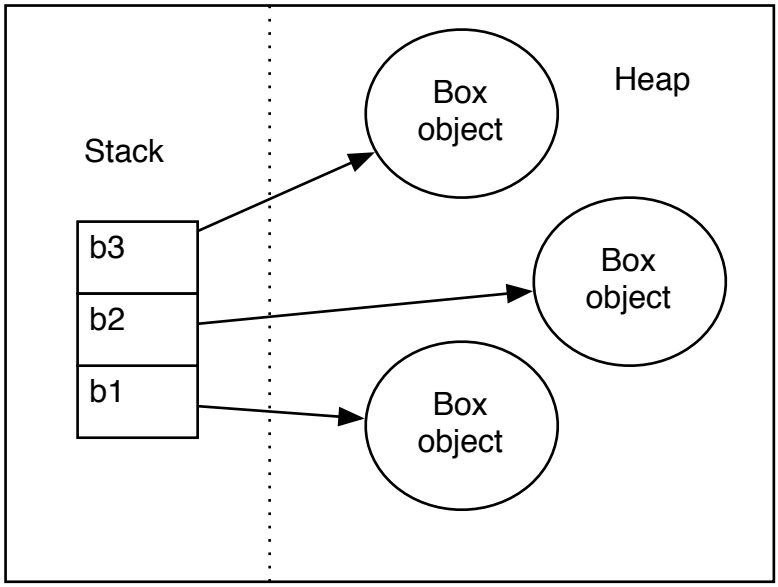

Figure 4.3: Memory diagram showing three `Box` reference variables and their objects

Local variables – those that are declared within methods – live on the stack. However, instance variables are declared outside methods. Where do they live? Look at Figure 4.4.

Instance variables live with their containing object, on the heap.

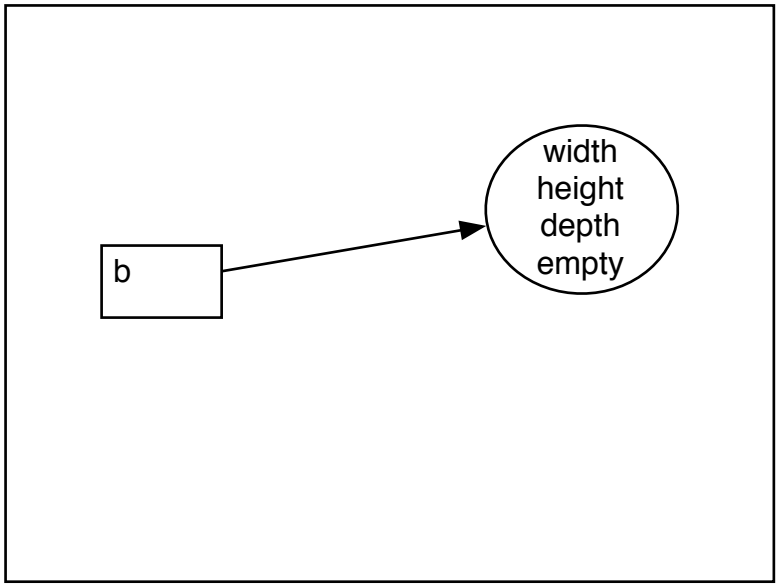

Figure 4.4: Instance variables live with their containing object, on the heap

4.4 Life on the garbage-collectible heap

Reading: pp. 57–58 of *HFJ*.

These sequences of memory diagrams are very helpful when trying to work out the effect of code blocks such as:

```
Book b = new Book();
Book c = new Book();
Book d = c;
c = b;
```

and

```
Book b = new Book(); // Object 1
Book c = new Book(); // Object 2
b = c;
c = null;
```

Sequences of memory diagrams such as those on pp. 57 – 58 are very important and you should make sure that you understand how they are formed and what they explain.

The memory diagrams on pp. 57 – 58 depict object life and death. A reference may be *active* or *null* and an object may be *reachable* or *abandoned*. Abandoned objects are eligible for garbage collection, and are effectively lost from the program since they are not reachable.

4.5 Object arrays

Reading: pp. 59–60 of *HFJ*.

```
Box[] boxes = new Box[10];
```

`boxes` is an object array. After initialisation, each element, for example `boxes[3]`, refers to an object. `boxes` is an array of references to objects on the heap.

Arrays are just one of Java's *data structures*. They allow fast access to a random element through the use of sub-scripting. Arrays also tend to be laid out in adjacent memory cells and are therefore very efficient.

Any array is actually an object itself, so `boxes` points to an array object on the heap. Each element of this object itself points to another object on the heap, or is set to null.

4.6 Remote controlling an object

Reading: pp. 54–61 of *62*, HFJ.

The dot operator acts on an object to return the value of a variable, or to invoke (call) a method. You might like to think of the object reference as a remote controller for the object. An instruction such as `dogs[i].bark()` is analogous to pressing the button `bark()` on controller `drops[i]`. The controller sends the message to the actual dog object on the heap.

4.7 Summary

- There are two flavours of variables: primitive and reference.
- Variables must always be declared with a name and a type.
- The value of a primitive variable is the bits representing the value (e.g. 5, 'a', true, 3.1416, etc.)
- A reference variable is like a remote control. The dot operator (.) is like pressing a button on the remote control to access a method or instance variable.
- A reference variable has the value `null` when it is not referencing an object.
- An array is always an object, even if the array is declared to hold primitives. There is no such thing as a primitive array, only an array that holds primitives.
- Memory diagrams illustrate object life on the heap.
- References may be active or null; objects may be reachable or abandoned.

4.8 Programming

Learning activity

Write an animation, `GooDrops`, of many differently sized drops falling at various speeds. Figure 4.5 captures a single frame to show you what to aim for, or you can run GooDrops from the CD-ROM to look at the whole animation. You should place `GooDrops` in a new package, `ReferenceTypes` and you should copy **simpleobjects/Drop.java** and save in **CIS220/ReferenceTypes** along with `GooDrops.java`.

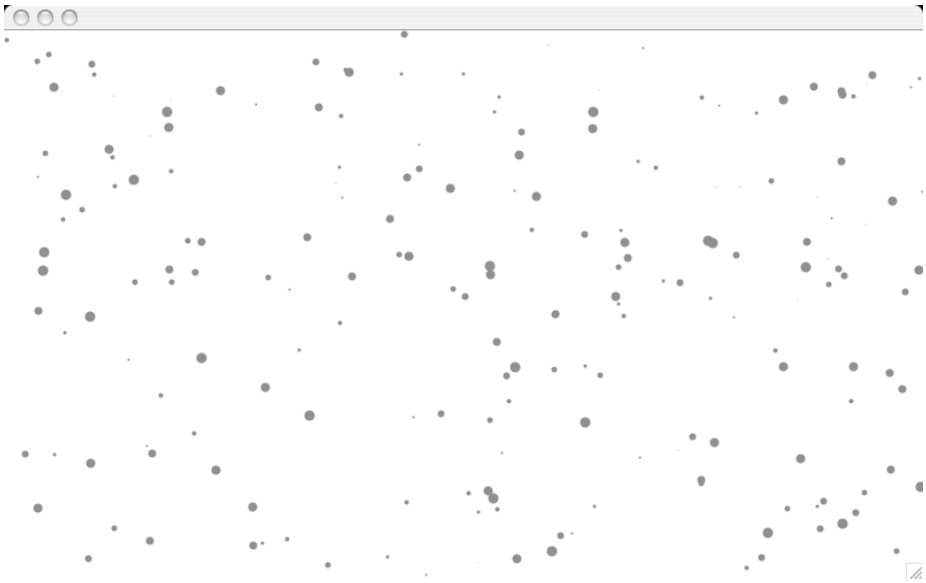

Figure 4.5: Screenshot of GooDrops

4.9 GooDrops

```
package referencetypes;

import java.awt.Graphics;
import java.util.Random;

import goo.Goo;

public class GooDrops extends Goo {

  private Drop[] drops;
  private int numDrops, maxSize = 9, maxVel = 9;
  private Random random;

  public GooDrops(int w, int h, int nd) {

    super(w, h);
```

```
    numDrops = nd;

    drops = new Drop[numDrops];
    random = new Random(1962);

    for (int i = 0; i < numDrops; i++) {

      int xpos = random.nextInt(w);
      int ypos = random.nextInt(h);
      int xvel = 0;
      int yvel = 1 + random.nextInt(maxVel);
      int size = 1 + random.nextInt(maxSize);
      drops[i] = makeDrop(xpos, ypos, xvel, yvel, size);
    }
  }

  public Drop makeDrop(int xpos, int ypos,int xvel,int yvel,
            int size){

    return new Drop(xpos, ypos, xvel, yvel, size);
  }

  public void draw(Graphics g) {

    for (int i = 0; i < numDrops; i++) {

      drops[i].move(getWidth(), getHeight());
      drops[i].draw(g);
    }
  }

  public Drop[] getDrops(){

    return drops;
  }

  public Random getRandom(){

    return random;
  }

  public static void main(String[] args) {

    int width = 800;
    int height = 500;
    int numDrops = 200;
    GooDrops gd = new GooDrops(width, height, numDrops);

    gd.smooth();
    gd.go();
  }
}
```

Here is an implementation of `GooDrops`. It is similar to `GooDrop`; the main difference is the use of a `Drop` array, declared as an instance variable, to hold the drops. `Drop` itself is imported from our `simpleobjects` package.

The constructor includes a loop through the `Drop` array, calling an instance method `makeDrop`. This one-line method may seem unnecessary, but it has been included

with a view to inheritance; subclasses of `GooDrops` can override `makeDrop` in order to fill the drop array with a different type of drop.

A `java.util.Random` object is used to generate pseudorandom integers because it is more convenient than calling `Math.random()`, and because we can guarantee the same sequence of pseudorandom numbers, so the animation looks the same each time it is run. If we did not want this feature, we could instantiate `Random` with a different seed at each invocation, for example by writing `random = new Random(System.currentTimeMillis());`

The draw method accesses each drop from the array one by one. Each drop is asked to move, and then to draw itself.

Notice that the instance variables have been marked as `private`. This means that other objects cannot access these variables directly. Instead they have to call *getters*. The getters in this class are `getRandom()` and `getDrops()`. The idea behind this complication is the object design principle known as *data-hiding* (see Chapter 3) or, synonymously as encapsulation (see Chapter 5).

The application is launched from `GooDrops`' own main, rather than using a separate launcher program.

Learning activity

Implement a coloured drop, **ColourDrop.java**, which draws a drop of any colour. Write an application, `GooDropsInColour`, to show falling, colourful drops.

Hint. The Java API documentation on `java.awt.Color` class shows various `Color` constructors. Colours can be represented in several ways. In the RGB colour space, each red, green and blue component of the colour can be quantified either with a number in the range [0, 1.0f], or as an integer between 0 and 255 – see below.

```
Color

public Color(int r,
             int g,
             int b)
Creates an opaque sRGB color with the specified red, green, and blue
values in the range (0 - 255). The actual color used in rendering
depends on finding the best match given the color space available
for a given output device. Alpha is defaulted to 255.
Parameters:
r - the red component
g - the green component
b - the blue component
```

4.10 Drop in Colour

```
package referencetypes;

import java.awt.Color;
import java.awt.Graphics;
```

```
public class ColourDrop extends Drop {

  Color color;

  public ColourDrop(int x, int y, int vx, int vy, int sz, Color c
      ) {

    super(x, y, vx, vy, sz);
    color = c;
  }

  public void draw(Graphics g){

    g.setColor(color);
    g.fillOval(xpos, ypos, size, size);
  }
}
```

By subclassing `Drop` we save duplicating code. Since a Colour Drop moves in just the same way as a Drop, we can simply subclass Drop in order to retain this behaviour, but override `draw` to render a Drop in colour. The colour itself is saved as an instance variable of type `java.util.Color`.

4.11 GooDrops in Colour

```
package referencetypes;

import java.awt.Color;
import java.util.Random;

import referencetypes.Drop;
import referencetypes.GooDrops;

public class GooDropsInColour extends GooDrops {

  public GooDropsInColour(int w, int h, int nd) {

    super(w, h, nd);
  }

  public Drop makeDrop(int xpos, int ypos, int xvel, int yvel,
      int size) {

    Random random = getRandom();
    Color color = new Color(random.nextInt(256), random.nextInt
        (256),
        random.nextInt(256));
    return new ColourDrop(xpos, ypos, xvel, yvel, size, color);
  }

  public static void main(String[] args) {

    int width = 800;
    int height = 500;
    int numDrops = 200;

    GooDrops gd = new GooDropsInColour(width, height, numDrops);
```

```
    gd.background(0); // black background
    gd.frameRate(25); // 25 frames per sec
    gd.smooth();
    gd.go();
  }
}
```

The application `GooDropsInColour` also uses inheritance to save us work. By subclassing `GooDrops` we only need to override `makeDrop`. The random R, G and B integers are generated inside the constructor call.

The launcher, `main` sets the display background to black with a call to Goo's `background(int greyscale)`, where greyscale can be set between 0 (black) and 1 (white). Another call to Goo's `framerate` asks for a framerate of 25 frames per second.

4.12 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- understand the following concepts:
  - primitive types, reference types, and the difference between them
  - active references, null references and the difference between them
  - reachable objects and abandoned objects
- explain the limitations of programming with primitive types
- understand the importance of memory diagrams, how they are formed and what they explain
- understand that variables must always be declared with a name and a type
- understand that the value of a primitive variable is the bits representing the value
- explain how a reference variable can be seen as working like a remote control
- explain how a reference variable has a value *null* when it is not referencing an object
- describe how an array is always an object, even if the array is declared to hold primitives
- explain how memory diagrams can be used to illustrate the operation of the heap.

Chapter 5

Object behaviour

Essential reading

HFJ Chapter 4.

5.1 Introduction

We know that objects have state and behaviour. But how are these related? In this chapter we look at the interaction between them.

5.2 Methods and instance variables

Reading: pp. 71–73 of *HFJ*.

Look carefully at the class box on page 72 of *HFJ*. This is the UML representation of a class, showing clearly the separation between instance variables (state) and methods (behaviour). Every instance of a class (i.e. an object) has the same methods, but instances might be in different states, and might hence behave differently, because methods make use of instance variables to perform their task. Notice how the instance methods are declared in the `Dog` class on page 73 of *HFJ*.

Reading: pp. 74–78 of *HFJ*.

A Java programme, remember, is a group of communicating objects. A `Dog`, for example, responds to the message *bark*. If we want to know what messages we can send to an object, and how the object may react as a consequence, we need to refer to the API. Reference books such as *Java in an Nutshell* provide comprehensive lists of method *signatures*. The method signature defines everything you may need to know about the method before calling it. The signature is the method specification, and defines the API for the method.

The method signatures can also be used during class design. Sometimes programmers will write a class skeleton, consisting of instance variables and method signatures. The class method bodies are then filled in during implementation.

The method `signature` for bark is:

```
void bark(int numOfBarks)
```

and a general method signature is:

```
modifier(s) return-type method-name( param-list ) [ throws exceptions ]
```

You will be provided with a list of method signatures in the appendix to your examination paper because you are not expected to remember everything. In real life, programmers develop their programs with frequent reference to the API. Don't worry about [throws exceptions]—you will learn what this means in Part II of the course, so you can ignore it for now.

`bark` has no modifier (`private` and `public` are examples of modifiers), its return type is `void` (i.e. nothing is returned), it has a single parameter in the param-list, an integer, and no exceptions are thrown.

Calling objects send the message `bark` to a dog object (this is known as *method invocation*) and they will not expect any object or primitive to be returned as a result. Furthermore they must send an integer parameter in their message. Assuming that the caller has a reference to a `Dog` object `patch`, the invocation is, for example:

```
int timesToBark = 3;
patch.bark(timesToBark);
```

The caller sends one or more *arguments* to the receiving object. These arguments must match, in terms of type and number, the argument list in the methods declaration. The values of the passed variables are known, inside the receiving method, as parameters. The parameter `numOfBarks` in `bark` is in fact a local variable. This is a *copy* of the caller's argument; Java passes the *value* of an argument. The method on page 74 of *HFJ* alters the value of the local `numOfBarks` variable, but the caller's `timesToBark` will remain at its assigned value, i.e. *3*.

5.3 Encapsulation

Reading: pp. 80–82 of *HFJ*.

This section explains one of the chief characteristics of object programming: encapsulation, or data-hiding. The basic idea is not to allow any other object to access a given object's instance variables. In order to change the state of an object, another object must call a particular kind of method known as a *setter*. The instance variable is declared `private`, but the setter has `public` access. This means that all alterations to state are funnelled through a single block of code. The setter has the sole responsibility for ensuring that illegal state changes do not happen. If access to state was unrestricted, and illegal states were possible, code to filter out illegal state changes would be distributed throughout the program, occurring in many objects and making code maintenance difficult. Encapsulation is another example of code-reuse, the overarching principle of all object design.

Furthermore, all the caller wishes to know from an object is a particular value of a property. Implementation using a single point of control means that the object can compute and return this value in an arbitrary way; the details are unnecessary as far as the caller is concerned, and the system is more flexible because the details of the method can be changed at a later date. For example, suppose a `WeatherStation` object is asked for its temperature using `getTemp()`. It does not matter to the caller if the Weather Station object holds a temperature instance variable, or if the Station dynamically computes the temperature using calls such as `readThermometer()` to other internal methods.

5.4 Local and instance variables

Reading: pp. 84–85 of *HFJ*.

Instance variables are declared within a class, but not inside a method. These are initialised to the default values *0, 0.0, false* and `null` for `integer, floating point, boolean` and `object` types respectively. But local variables—those that are declared within a method—must be initialised before use. If they are not initialised, the compiler will complain. Arguments, which are local variables, will be initialised automatically when the method is called.

5.5 Comparing variables

The == operator returns true if two primitives have the same value. However, the value of a reference variable is just the bit string address of the object on the heap. Hence the == operator between object variables returns true if two references point at the same object. Use the `equals()` method to discover if two *different* objects are "equal" in the sense that they have identical state.

5.6 Summary

- Classes define what an object knows and what it does.
- Things an object knows about are its instance variables (its 'state').
- Things that an object does are its instance methods (behaviour).
- A method signature defines everything you need to know about a method. It has the general form:

  modifier(s) return-type method-name(param-list) [throws exceptions]
- A method can have parameters, which means that you can pass one or more values to the method.
- The number and type of values you pass in must match the order and type of the parameters declared by the method.
- Java uses pass-by-value.
- Values passed in and out of the methods can be implicitly promoted to a larger type, or explicitly cast to a smaller type.
- You can pass a literal (e.g. 5, 'a', true, 3.1416, etc.) or a variable of the declared parameter type (e.g. `x` where `x` has been declared as an `int` variable.)
- A method must declare a return type. A `void` return type means that the method does not return anything.
- If a method declares a non-void return type, it must return a value compatible with the return type.
- According to the principle of encapsulation, instance variables should be declared private, and setters/getters methods declared public.
- Instance variables are automatically initialised by the compiler; but local variables must be initialised explicitly.
- Variables can be compared using the equals operator == and by calling `equals()`.

5.7 Programming

Learning activity

Write code for a Solid Ellipse class. A Solid Ellipse is specified by the position of the top left corner of the bounding rectangle, and by its width and height, and, when drawn, its interior is filled. Figure 5.1 shows a Solid Ellipse at (150, 150) and a bounding rectangle of width and height 200.

Remember to encapsulate your class.

Save your programs for this chapter in **CIS220/howobjectsbehave**.

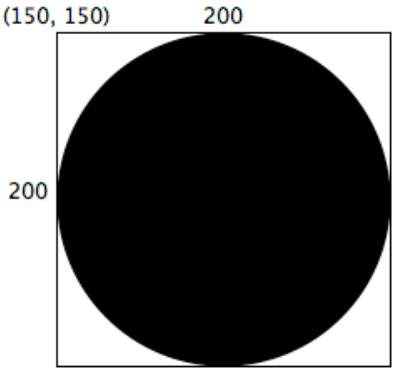

Figure 5.1: Ellipse at (150, 150). The picture shows the bounding rectangle of width and height 200.

5.8 Ellipse

```
package howobjectsbehave;

import java.awt.Color;
import java.awt.Graphics;

public class SolidEllipse {

  private int x, y, width, height;
  private Color color;

  public SolidEllipse(int x, int y, int w, int h, Color c) {

    this.x = x;
    this.y = y;
    width = w;
    height = h;
    color = c;
  }

  public void fill(Graphics g) {

    g.setColor(color);
    g.fillOval(x, y, width, height);
  }

  public int getX() {
    return x;
  }

  public void setX(int x) {
    this.x = x;
  }

  public int getY() {
    return y;
  }

  public void setY(int y) {
    this.y = y;
  }

  public int getWidth() {
    return width;
  }

  public void setWidth(int w) {
    width = w;
  }

  public int getHeight() {
    return height;
  }

  public void setHeight(int h) {
    height = h;
```

```
  }

  public Color getColor() {
    return color;
  }

  public void setColor(Color c) {
    color = c;
  }
}
```

The Solid Ellipse class looks long, but this is due to the get and set methods. Notice that the instance variables are marked `private`. The constructor demonstrates a new bit of syntax; `this` is a reference that all objects have to themselves. This distinguishes the instance variable `x` (referred to as `this.x`) from the local variable `x`, defined in the constructor parameter.

Learning activity

The figure below shows a hoop. Write a `GooHoop` and an application to display it. Take care to encapsulate instance variables.

Figure 5.2: Screenshot of a Goo Hoop

5.9 Hoop

```
package howobjectsbehave;

import java.awt.Color;
import java.awt.Graphics;

public class Hoop {

  private SolidEllipse outer, inner;

  public Hoop(int x, int y,
              int outerDiameter, int innerDiameter) {

    outer = new SolidEllipse(x, y, outerDiameter, outerDiameter,
        Color.BLACK);

    int thickness = (outerDiameter - innerDiameter) / 2;
    inner = new SolidEllipse(x + thickness, y + thickness,
                  innerDiameter, innerDiameter, Color.WHITE);
  }

  public void fill(Graphics g) {
    outer.fill(g);
    inner.fill(g);
  }

  public int getX() {
    return outer.getX();
  }

  public void setX(int x) {

    int thickness = getThickness();
    outer.setX(x);
    inner.setX(x + thickness);
  }

  public int getY() {
    return outer.getY();
  }

  public void setY(int y) {

    int thickness = getThickness();
    outer.setY(y);
    inner.setY(y + thickness);
  }

  public int getDiameter() {
    return outer.getHeight();
  }

  public int getThickness() {
    return inner.getX() - outer.getX();
  }
}
```

This hoop implementation has just two instance variables, an inner and an outer ellipse. The draw method carefully fills one Solid Ellipse inside the other. Notice how, in this implementation, some of the access methods (get/set) do not change and retrieve instance variables directly, but they compute or retrieve values using other method calls.

The class is encapsulated because the details of the implementation are hidden to the object that is making the call (the client). Later we may wish to change our implementation and add two instance variables, `thickness` and `diameter`. The get/set methods would then access the instance variables directly. However none of the client code would need to be changed; all the client needs to know (i.e. for its own computations) is the thickness and diameter of the hoop. The client doesn't care if these values are stored or are computed afresh (as is the case here) each time.

5.10 Hoop App

```
package howobjectsbehave;

import goo.Goo;

import java.awt.Graphics;

public class HoopApp extends Goo {

  Hoop hoop;

  HoopApp(int w, int h, Hoop wa) {

    super(w, h);
    hoop = wa;
  }

  public void draw(Graphics g) {

    hoop.fill(g);
  }

  public static void main(String[] args) {

    // Create a hoop
    Hoop hoop = new Hoop(100, 66, 60, 30);

    // Create a test instance and run
    HoopApp app = new HoopApp(500, 300, hoop);
    app.smooth();
    app.noloop();
    app.go();
  }
}
```

The test class is straightforward. The only new code is the `noloop` which tells goo to draw a single frame (i.e. no animation).

Learning activity

Write a `MovingHoop` class by subclassing `Hoop` and adding a `move(int width, int height)` method. A moving hoop should be able to move in any direction across the screen; when it hits the edge of the window it should bounce back, like a billiard ball hitting the side cushion.

Write an application to demonstrate your moving hoop.

You can run MovingGooHoop from the CD-ROM to see what you should be aiming at.

5.11 Moving Hoop

```
package howobjectsbehave;

public class MovingHoop extends Hoop {

  private int xVel = 2;
  private int yVel = 4;

  public MovingHoop(int xin, int yin, int d, int t) {
    super(xin, yin, d, t);
  }

  public void move(int width, int height) {

    int x = getX();
    int y = getY();

    x = x + xVel;
    y = y + yVel;

    int diam = getDiameter();

    if (x + diam >= width) {
      xVel *= -1;
      x = width - diam;

    } else if (x <= 0) {
      xVel *= -1;
      x = -x;
    }

    if (y + diam >= height) {
      yVel *= -1;
      y = height - diam;

    } else if (y <= 0) {
      yVel *= -1;
      y = -y;
    }

    setX(x);
    setY(y);
  }
}
```

The moving hoop only needs two more instance variables, the `x` and `y` velocities, and a single method, `move`.

`move` firstly adds velocity to position to form an updated position. The hoop, of diameter d, will have crossed the right boundary if $x + d$ is larger than the width, w, of the window. In which case, the x component of velocity needs to be reversed.

5.12 Moving Hoop App

```
package howobjectsbehave;

import java.awt.Graphics;

import goo.Goo;

public class MovingHoopApp extends Goo {

  MovingHoop hoop;

  public MovingHoopApp(int w, int h, MovingHoop gh) {

    super(w, h);
    hoop = gh;
  }

  public void draw(Graphics g) {

    hoop.fill(g);
    hoop.move(getWidth(), getHeight());
  }

  public static void main(String[] args) {

    MovingHoop hoop = new MovingHoop(100, 75, 60, 30);
    Goo app = new MovingHoopApp(500, 500, hoop);
    app.smooth();
    app.frameRate(50);
    app.go();
  }
}
```

Another short animation program, illustrating the power of Goo. The moving hoop is instantiated in `main` and a reference passed into the constructor. `draw` asks the moving hoop to draw itself and then to move.

Learning activity

Suppose that two hoops are considered to be 'equal' if they have the same diameter and thickness. Write a class `HoopWithEquals` which subclasses `Hoop`, and has two new methods, as follows:

1. An `equals` method for the comparison of this hoop with another, as in:

   ```
   HoopWithEquals hoop, hoop2;
   .
   .
   .
   boolean areEqual = hoop.equals(hoop2);
   ```

2. A `clone` method which returns a new `HoopWithEquals` object which is identical to the calling hoop, and is positioned at the same place, as in HoopWithEquals hoop = new...; HoopWithEquals hoop2 = hoop.clone();

```
package howobjectsbehave;

public class HoopWithEquals extends Hoop {

  public HoopWithEquals(int xin, int yin, int d, int t) {

    super(xin, yin, d, t);
  }

  public boolean equals(HoopWithEquals w) {

    return (getDiameter() == w.getDiameter() &&
                getThickness() == w.getThickness());
  }

  public HoopWithEquals clone() {

    int outerDiameter = getDiameter();
    int innerDiameter = outerDiameter - 2 * getThickness();
    return new HoopWithEquals(getX(), getY(), outerDiameter,
        innerDiameter);
  }

  public static void main(String[] args) {

    // Create a hoop
    HoopWithEquals hoop = new HoopWithEquals(100, 66, 90, 20);

    // An identical hoop, but a different object
    HoopWithEquals hoop1 = hoop.clone();
    System.out.println("hoop1 == hoop? " + (hoop1 == hoop) + "\t"
        + "hoop1.equals(hoop)? " + hoop1.equals(hoop));

    // Point hoop1 to same object that hoop refers to
    hoop1 = hoop;
    System.out.println("hoop1 == hoop? " + (hoop1 == hoop) + "\t"
        + "hoop1.equals(hoop)? " + hoop1.equals(hoop));
  }
}
```

`equals` returns true if the diameters and thicknesses match. Notice the use of a single complex conditional expression; this could be split into a couple of sub-expressions.

`clone` returns a `HoopWithEquals`, so the method signature is as follows:

```
public HoopWithEquals clone().
```

Learning activity

The main method above demonstrates the use of = and `equals` on reference variables. What is the output of `main`? You can check your answer by running the program. Make sure you understand how and why it works this way.

5.13 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- demonstrate an understanding of the following concepts:
  - classes
  - instance variables
  - instance methods
  - method signatures
- describe how classes define what an object knows and what it does
- describe the form of a method signature
- describe how a method can have parameters and how the number and type of values that you pass must match the order and type of parameters declared by the method
- describe how Java uses pass-by-value
- describe how values passed to and from methods can be implicitly promoted to a larger type, or explicitly cast to a smaller type
- explain how you can pass a literal or a variable of the declared parameter type
- write a method that has a return type, and decide when this is necessary
- explain the difference between a void and a non-void return type
- describe the principle of encapsulation, the difference between instance and local variables and how they are initialised
- explain how variables can be compared.

Chapter 6

Program development

Essential reading

HFJ Chapter 5.

6.1 Introduction

HFJ Chapter 5, provides an example of coding a simple application from start to finish.

The chapter also shows you some useful features of the Java Programming Language: the enhanced `for` loop, the decrement and increment operators, how to get the integer value of a number String such as ”56”, useful helper code for entering user input into a programme, how to generate a pseudorandom number, and, finally, explains *casts* of primitives.

6.2 Design, then implement

Reading: pp. 95–109 of *HFJ*.

Much of this material will be familiar to you from your Software Engineering course, except here the development process is scaled down. In this example, the project begins with a high-level design, expressed here as a flow chart. Pseudocode (called “prep code” by the *HFJ* authors) is a bridge between design and implementation. In the Extreme Programming methodology, a test framework is written **before** the classes are coded.

6.3 Additional features of the JPL

Reading: pp. 110–117 of *HFJ*.

The enhanced `for` can be used for looping through what Java calls *collections*. An array, and an `ArrayList`, which you will encounter in the next chapter are collections. Otherwise you will probably use the regular `for`, in preference to a `while` for looping a set number of times through a code block.

The compiler will complain if you try to assign a `long` value to an `int` variable, or a `float` to an `int`. The cast operator overrides this complaint, but will reduce accuracy and even produce strange results, so it must be used carefully. Floats, for

example, when cast to integers, will lose all their decimal places, rather than being rounded. 3.9f will become 3 when cast as an `int`. One frequent use of casts is in the generation of pseudorandom integers, as the code on page 111 of *HFJ* illustrates.

6.4 Summary

- Start with a high level design.
- Write pseudocode and then test the pseudocode.
- Then write the actual code.
- `for` loops are preferred when you know how many loops will be performed.
- Post- and pre-decrement/increment operators.
- `Integer.parseInt()` converts, if possible, a string into an integer.
- `break` forces an exit from a loop.
- The enhanced `for` loop.
- Casting primitives.

6.5 Programming

Learning activity

The next project is to develop an animation of the night sky. We will design the system in three stages: first by thinking what classes we may need, then by completing class boxes, and finally by writing some pseudocode.

To begin, think what objects are in the night sky and write class boxes.

6.6 Goo By Starlight

I found three possible classes; Sky, Star and Moon. Star and Moon classes are easy to outline because they are fairly similar to the drops and the hoops of earlier projects:

Star
int xpos int ypos int size
draw() twinkle()

Moon
int xpos int ypos int size
draw() move()

The sky is a less obvious candidate since it only has one behaviour, but it could store the interesting objects (Moon and Stars), and start the animation.

Sky
Star[] stars Moon moon Color color
draw()

Learning activity

Write pseudocode for Sky. The pseudocode should show how the objects are instantiated and the animation sequence.

6.7 Pseudo Sky

```
Class Sky, extends Goo

    Star[] stars, Moon moon, Color color

    Sky

       declare an array of stars

       initialise color

        for each star in stars
            instantiate a star object at a random location and assign to star
        end

        calculate position (xMoon, yMoon) for a given angle
        instantiate Moon at (xMoon, yMoon)

    end

    draw(g)

        draw sky - black background

        for each star in sky
            twinkle
        end

        draw moon
        move Moon

    end

end Class
```

The pseudocode outlines the set-up and the animation sequence.

Learning activity

Implement `Sky` and write test classes for `Moon` and `Star`. Do not implement the draw, twinkle and move of Moon and Star methods.

6.8 Sky

```
package writingaprogram;

import goo.Goo;

import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;

public class Sky extends Goo {

  private Star[] stars;
  private Moon moon;
  private Color color;

  public Sky(int numStars) {

    super(500, 500);

    stars = new Star[numStars];
    Random random = new Random(2008);
    color = Color.black;

    int width = getWidth(), height = getHeight();
    for (int i = 0; i < stars.length; i++) {
      stars[i] = new Star(random.nextInt(width), random.nextInt(
          height));
    }

    moon = new Moon();
  }

  public void draw(Graphics g) {

    background(color);

    for (Star star : stars) {
      star.twinkle(g);
    }

    moon.draw(g);
    moon.move(getWidth(), getHeight());
  }

  public static void main(String[] args) {

    Sky sky = new Sky(100);
    sky.smooth();
    sky.go();
  }
}
```

`Sky` subclasses `Goo` and sets up the animation by providing the major loop method `draw`.

6.9 Star

```
package writingaprogram;

import java.awt.Graphics;

public class Star {

  int xpos, ypos, size = 2;

  public Star(int xpos, int ypos) {

    this.xpos = xpos;
    this.ypos = ypos;
  }

  public void twinkle(Graphics g) {
    System.out.println("star: twinkling");
  }
}
```

The star is a very simple class; the important `twinkle` method is not implemented in this test version.

6.10 Moon

```
package writingaprogram;

import java.awt.Graphics;

public class Moon {

  int xpos, ypos, size = 40;
  double r, theta;

  public Moon(){

    r = 0;
    theta = 0;
  }

  public Moon(double r, double theta) {

    this.r = r;
    this.theta = theta;
    // code to initialise position
  }

  public void draw(Graphics g) {
    // drawing code
    System.out.println("moon: draw");

  }

  public void move(int width, int height) {
    // movement
    System.out.println("moon: moving");
  }
}
```

The Moon moves in an arc across the sky. This motion is easily calculated in radial coordinates (r, θ), and then converted to screen (x, y) coordinates for rendering.

Learning activity

If you are happy that your test program is running, it is time to implement the difficult twinkle and move methods.

Watch the Goo by starlight animation on the CD-ROM to see what to aim at.

Have a go, and compare to my Goo by starlight program below.

6.11 GooStar

```
package writingaprogram;

import java.awt.Color;
import java.awt.Graphics;

public class GooStar extends Star {

  Color color;

  public GooStar(int xpos, int ypos, Color color) {

    super(xpos, ypos);
    this.color = color;
  }

  public void twinkle(Graphics g) {

    g.setColor(color);

    int s = (int) (size/2 + (Math.random() - 0.5));

    g.drawLine(xpos - s, ypos - s, xpos + s, ypos + s);
    g.drawLine(xpos, ypos - s, xpos, ypos + s);
    g.drawLine(xpos + s, ypos - s, xpos - s, ypos + s);
    g.drawLine(xpos - s, ypos, xpos + s, ypos);

  }
}
```

For simplicity I have subclassed `Star` and taken the opportunity to add colour. The twinkle algorithm is one way of imitating this effect. See if you can figure out how it works.

6.12 GooMoon

```
package writingaprogram;

import java.awt.Color;
import java.awt.Graphics;

public class GooMoon extends Moon {

  double deltaTheta = 0.001;

  public GooMoon(int width, int height) {

    // Moon starts on left edge of window, half way up.
    super(Math.sqrt(height * height / 4 + width * width / 4),
        0.75 * Math.PI);

    // call move() in order to calculate (x, y)
    move(width, height);
  }
```

```
  public void draw(Graphics g) {

    g.setColor(Color.WHITE);
    g.fillOval(xpos, ypos, size, size);
    g.setColor(Color.BLACK);
    g.fillOval(xpos + size / 4, ypos - size / 8, size, size);
  }

  public void move(int width, int height) {

    theta = theta - deltaTheta;

    // Transform from (r, theta) to (x, y)
    double x = width / 2 + r * Math.cos(theta);
    double y = height - r * Math.sin(theta);

    // Calculate coordinates of top left bounding rectangle
    xpos = (int) Math.round((x - size / 2));
    ypos = (int) Math.round(y - size / 2);
  }
}
```

`Moon` has also been subclassed and `move` and `draw` have been overridden. The coordinate transformation in `move` might look a bit baffling; don't worry, the details are not important. A simple trick of overlaying two ovals has been used to draw a crescent.

6.13 Goo by starlight

```
package writingaprogram;

import goo.Goo;

import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;

public class GooByStarlight extends Goo {

  Star[] stars;
  GooMoon moon;

  public GooByStarlight(int numStars) {

    super(500, 500, true);

    stars = new Star[numStars];
    Random random = new Random(2008);

    int width = getWidth(), height = getHeight();
    for (int i = 0; i < stars.length; i++) {
      Color color = new Color(random.nextInt(256), random.nextInt
          (256),
          random.nextInt(256));
      stars[i] = new GooStar(random.nextInt(width), random
          .nextInt(height), color);
    }

    moon = new GooMoon(width, height);
  }

  public void draw(Graphics g) {

    background(0);

    for (Star star : stars) {
      star.twinkle(g);
    }
    moon.draw(g);
    moon.move(getWidth(), getHeight());
  }
}
```

The `Sky` has been renamed as `GooByStarlight` and the moon and the stars have been replaced by `GooMoon` and `GooStar`.

`GooByStarlight` calls the superclass constructor with a Boolean argument. If `false`, Goo makes a window with dimensions equal to the display dimensions of your computer. If `true`, the window is again scaled to the size of your screen, but Java enters "full screen exclusive" (FSE), if this is supported on your machine. This means that Java graphics takes over control of your entire screen. An "undecorated" window is displayed, which is one without any toolbars. The result is that the starry sky occupies the whole screen.

Since Java has, when in FSE, control of the entire screen, you will lose the command line and sight of your IDE if you are using one. Only Java can quit your program, but since the program is running in an eternal loop, this will never happen. Without intervention, you would have to manually shut down your machine. Luckily Goo allows mouse and keyboard interaction even in FSE. Goo will quit if you type escape, or control-q or control-c. It does this by calling `System.exit(0)` which closes down any Java application.

Here is the straightforward launcher code:

```
package writingaprogram;

public class StarlightApp {

  public static void main(String[] args){

    GooByStarlight sky = new GooByStarlight(500);
    sky.smooth();
    sky.go();
  }
}
```

6.14 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- write pseudocode and test whether it will do what you intend it to
- write actual code based on what you did in the pseudocode testing stage
- decide when `for` loops are preferred and when `while` loops are better
- explain post- and pre-decrement/increment operators
- describe how `Integer.parseInt()` converts a string into an integer
- explain how **break** forces an exit from a loop
- describe the enhanced `for` loop
- explain what is meant by casting primitives.

Chapter 7

The Java library

Essential reading

HFJ Chapter 6.

7.1 Introduction

The chapter explains the Java library in more detail, and how to use it, and shows you a very useful library class, the `ArrayList`. Some other aspects of the Java language are also covered, mainly by way of revision.

7.2 Using the API

Reading: pp. 158–160 of *HFJ*.

A large part of being a good Java programmer is knowing (part of) the Java API (the Java Library). Apart from the wasted time in coding for yourself a class that might already exist for free, an equivalent Java library class has been written by experts and is quite possibly better than yours! Code re-use, remember is a central aspect of object programming.

You will learn some of the most useful library classes in this course, and by reading *HFJ*. Otherwise you will pick them up by browsing through the Java docs or a book such as *Java in a Nutshell*, or by studying other people's programs.

7.3 The ArrayList

Reading: pp. 132–137 of *HFJ*.

The ArrayList is extremely useful. It is the preferred data structure when the number of elements that might be placed in the structure is unknown, and if you want similar performance to the array. (Arrays are mapped directly to memory and provide very fast access.)

An `ArrayList` is rather like an array that can grow or shrink. Pages 136–137 compare the `ArrayList` to a regular array.

Although an array is an object, it uses special syntax for creation and access to its elements. Arrays are special objects because they relate closely to how the JVM

manages memory. If your code asks for an array of 10 `int`s, the JVM will set aside 10 contiguous memory locations, with each location just big enough to hold an `int`. The JVM knows the memory location of the first element, and if your code accesses, for example, the third element of the array, the JVM just adds three times the size of an `int` on to the address of the start of the array. This is known as *random access* and it is very fast.

ArrayLists are also random access and are just about as fast as arrays. The only difference is that primitives must be 'wrapped' up as objects before they are inserted. Wrapping and unwrapping are automatic in Java 5 and subsequent versions. In fact ArrayLists are parameterised types in Java 5. For example:

```
ArrayList<String> names;
```

declares a list of Strings. This means that the compiler will check that only Strings can be inserted into the list. Prior to Java 5, any object could be inserted into a particular `ArrayList`; retrieved objects had to be cast back again to the correct type.

Java 5 introduces the *enhanced for* (*for each*) for iterating through collections such as the ArrayList;

```
for (String item : names){ // loops through all Strings in names
   // do something with item
}
```

7.4 Boolean expressions

Reading: pg. 151 of *HFJ*.

Make sure you know how the short circuit operators work, and do not use the non-short circuit operators in Boolean expressions. Remember to use parentheses to specify the order of evaluation in long Boolean expressions, unless you are very confident about the rules of precedence.

7.5 Packages and imports

Reading: pp. 154–157 of *HFJ*.

The classes are packed into packages. You can either refer to the full name in the form `package.class`, or simply import the package at the head of your programme. The java.lang package is so fundamental (that's where the class `System` is found, for example) that this is imported automatically anyway and doesn't need to be declared by the programmer.

7.6 Summary

- The `ArrayList` is a class in the Java API.
- You can consult the API by looking at the javadocs, or at a summary such as *Java in a Nutshell*.

- Useful `ArrayList` methods are `add()`, `remove()`, `indexOf()`, `isEmpty()`, `contains()`. The number of elements in an array is available as the `length` variable; the length of an `ArrayList` is returned by the `size()` method.
- An `ArrayList` resizes to whatever size is appropriate.
- `ArrayLists` can be paramaterised using a type parameter in angle brackets `< >`.
- An `ArrayList` can only hold objects; primitives are automatically wrapped into objects, and unwrapped back to primitives on insertion and retrieval.
- Classes are grouped into packages.
- The full name of a class is of the form *packagename.classname*.
- A class from the API must either be imported, or specified by its full name (with the exceptions of java.lang which contains the most fundamental class of the API).
- `&&` and `||` are short-circuit operators.
- `&` and `|` are not short-circuit operators and are typically used for manipulating bits.
- Use `expressionA != expressionB` or `!(expressionA = expressionB)` to test if two expressions do not evaluate to the same result.
- You can either learn the order of precedence or use parentheses to specify the order of evaluation of long Boolean expressions.

7.7 Programming

The following program, `SimpleMouseAndKey`, shows how someone can interact with your Goo program using the keyboard and mouse. If you look at the Goo source code, you will see that Goo 'implements' `KeyListener`, `MouseListener` and `MouseMotionListener`. Chapter 9 explains the meaning of this keyword in some detail, but all you need to know now is that Goo has key and mouse methods such as `keyPressed` and `mouseClicked` which are called by the JVM in response to user interaction. And your Goo subclass can override these methods and take whatever action you program into the method bodies.

7.8 Simple Mouse and Keyboard interaction

```
package thejavalibrary;

import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.MouseEvent;

import goo.Goo;

public class SimpleMouseAndKey extends Goo {

  public void keyPressed(KeyEvent e) {
    System.out.println("Kep pressed " + e.getKeyCode());
  }

  public void keyTyped(KeyEvent e) {
    System.out.println("Kep typed " + e.getKeyChar());
```

```
  }

  public void mouseClicked(MouseEvent e) {
    System.out.println("Mouse clicked at (" + e.getX() + ", " + e
        .getY()
        + ")");
  }

  public void mouseDragged(MouseEvent e) {
    System.out.println("Mouse dragged at (" + e.getX() + ", " + e
        .getY()
        + ")");
  }

  public void draw(Graphics g) {
  }

  public static void main(String[] args) {
    new SimpleMouseAndKey().go();
  }


}
```

Learning activity

Write an application that draws connected points (see Figure 7.1). A point is added to the image each time a user clicks the mouse on the screen (the point is located at the mouse cursor itself) and lines are drawn connecting the points on the order in which they are added, with the last point connected to the very first one. The user can move a point by placing the cursor over the point and dragging it across the screen.

Run `LinesAndPoints` from the CD-ROM to get an idea of how it works.

Hint: Think carefully about the data structures you will use. Explore the Graphics API to find out how to draw lines and ovals.

Extend your program so that the points move across the window and bounce like billiard balls from the window edge. The animation should toggle between stop and start by typing 'p'.

Further extend your program so the points and lines are removed from the screen of the user types 'c'.

Save your classes in the package `thejavalibrary`.

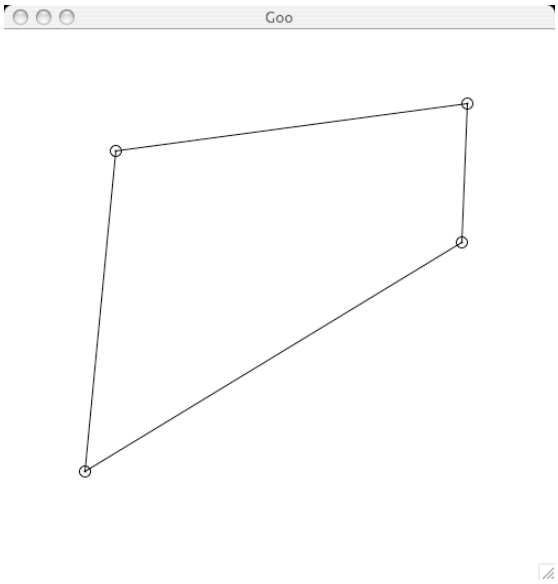

Figure 7.1: Four connected points.

7.9 Moving lines and points

```
package thejavalibrary;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.MouseEvent;
import java.util.ArrayList;

import goo.Goo;

public class LinesAndPoints extends Goo {

  private ArrayList<Point> points = new ArrayList<Point>();
  private boolean stopAll = true;

  public LinesAndPoints() {
    super(500, 500);
  }

  public void draw(Graphics g) {

    if (points.size() > 0) {

      Point previousP = points.get(points.size() - 1);
      g.setColor(Color.BLACK);

      for (Point p : points) {

        p.draw(g);
```

```
        g.drawLine(previousP.getX(), previousP.getY(), p.getX(),
            p.getY());
        previousP = p;
      }

      for (Point p : points) {

        if (!stopAll)
          p.move(getWidth(), getHeight());

        previousP = p;
      }
    }
  }

  public void mouseClicked(MouseEvent e) {

    points.add(new Point(e.getX(), e.getY()));
  }

  public void keyPressed(KeyEvent e) {

    if (e.getKeyChar() == 'c')
      points.clear();
    else if (e.getKeyChar() == 'p') {
      stopAll = !stopAll;
    }

  }

  public void mouseDragged(MouseEvent e) {

    Point mouseP = new Point(e.getX(), e.getY());
    for (Point p : points) {

      if (p.distance(mouseP) <= p.getRadius()) {

        p.setX(e.getX());
        p.setY(e.getY());
        break;
      }
    }
  }

  public static void main(String[] args) {

    Goo goo = new LinesAndPoints();
    goo.smooth();
    goo.go();
  }
}
```

The idea of a point has been encapsulated into a class of its own (see the following program). Since we do not know how many points there will be at any time, an `ArrayList` is the obvious choice for a points container. Notice how `mouseClicked` creates a new point and adds it to the array list.

`Graphics.drawLine()` connects the points together inside an enhanced for loop.

A Boolean variable `stopAll` starts/stops the movement.

Points can be repositioned manually by dragging them across the screen; `mouseDragged()` contains the required code.

`keyPressed()` gets the character from the `keyEvent` and compares against 'c' and 'p'. The point array list is emptied by calling `clear()` and the `stopAll` is toggled each time 'p' is keyed.

The code in the next section shows the Point class.

7.10 Point

```
package thejavalibrary;

import java.awt.Graphics;

public class Point {

  private int x, y, vx, vy, maxvel = 5, radius = 5;

  Point(int x, int y) {

    this.x = x;
    this.y = y;
    vx = vy = 0;

    while (vx == 0 || vy == 0) {
      vx = (int) (2 * maxvel * Math.random() - maxvel);
      vy = (int) (2 * maxvel * Math.random() - maxvel);
    }
  }

  public void move(int width, int height) {

    x = x + vx;
    y = y + vy;

    if (x + radius >= width) {

      vx *= -1;
      x = width - radius;

    } else if (x - radius <= 0) {

      vx *= -1;
      x = radius;
    }

    if (y + radius >= height) {

      vy *= -1;
      y = height - radius;

    } else if (y - radius <= 0) {

      vy *= -1;
      y = radius;
```

```
    }
  }

  public void draw(Graphics g) {

    int diameter = 2 * radius;
    g.drawOval(x - radius, y - radius, diameter, diameter);
    g.fillOval(x - 1, y - 1, 2, 2);
  }

  public double distance(Point p) {

    double delxsq = x - p.x;
    delxsq *= delxsq;

    double delysq = y - p.y;
    delysq *= delysq;
    return Math.sqrt(delxsq + delysq);
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public void setRadius(int radius) {
    this.radius = radius;
  }

  public int getRadius() {
    return radius;
  }

  public void setX(int i) {
    x = i;
  }

  public void setY(int i) {
    y = i;
  }
}
```

The point class has position, velocity and size instance variables and draw and move methods. A utility method `distance(Point p)` calculates the distance (using Pythagoras' rule) between the object that receives this message and a comparison point `p`.

Point movement is very similar to hoop movement from a previous program. The axle is a small filled oval and the rim is drawn using `drawOval`.

7.11 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- understand that `ArrayList` is a class in the Java API, and describe the `add()`, `remove()`, `indexOf()`, `isEmpty()` and `contains()` methods
- calculate the number of elements in an array, and explain that the length is returned using the `size()` method
- describe how to resize and how to parameterise anything that is an `ArrayList`
- understand that an `ArrayList` can only hold objects, and what is done to include primitives into an `ArrayList`
- describe how classes are grouped
- discuss the relation between classes and packages
- import a class from an API
- use notation for short-cut operators and manipulating bits
- test whether two expressions will evaluate to the same result or not
- specify the order of evaluation of long Boolean expressions.

Chapter 8

Inheritance

Essential reading

HFJ Chapter 7.

8.1 Introduction

Inheritance is one of the three defining characteristics of object programming. (The other two are dynamic binding and abstraction.) You were introduced to inheritance in Chapter 2, Objects, and you have seen it in use in the programming sections of most chapters. Here we provided a fuller account of the technical details of inheritance in Java, and discuss its use in program design.

8.2 Understanding inheritance

Reading: pp. 165–169 of *HFJ*.

Subclasses inherit members—instance variables and methods—from their superclass parent. They are also said to extend their more abstract superclass. A subclass can add more methods and instance variables of its own and it can *override* superclasses methods. It will do this to add more specialised behaviour; otherwise the object will use the superclass method.

Instance variables should not be overridden, because they don't define any special behaviour.

8.3 Designing inheritance

Reading: pp. 170–176 of *HFJ*.

1. Look for objects that have common attributes and behaviour.
2. Design a class that represents their common state and behaviour.
3. Decide if a subclass needs specific behaviours - these will be implemented as overridden methods.
4. Look for more opportunities for abstraction. One way to do this is to check if there are two or more subclasses that have common behaviour.
5. Finish the class hierarchy.

Which method is called? The JVM works its way up the inheritance hierarchy starting at the lowest (most specific) class until it finds a match. Study carefully the example on page 175. This is known as dynamic binding since it happens at runtime. The JVM has the responsibility of making the correct call, and not the compiler.

Reading: pp. 177–181 of *HFJ*.

Subclassing is a possible way of organising your code when the relationship between objects is IS-A or HAS-A. Inheritance is also a good design choice when there is shared behaviour amongst multiple classes of the same general type. However, do not use inheritance just for the sake of code reuse. Although code reuse is usually beneficial, it must not dominate a good programming principle: *avoid obscurity*; your programs should be easily understandable. Your mapping of the problem into computer code must make sense. A class stands for an idea of a classification of actual things. Grouping together very dissimilar things does not lead to clear thinking and clear code.

8.4 Advantages and disadvantages of inheritance

Reading: pp. 182–189 of *HFJ*.

8.4.1 Advantages of inheritance

There are two advantages: you avoid duplicating code, and you define a common protocol for a group of classes. Inheritance ensures that all subclasses have the same public methods that the supertype has.

But even more than this, inheritance exploits *polymorphism*.

The three steps of object declaration and assignment are, with the statement `Dog dog = new Dog()` as an example:

1. Declare a reference variable, `Dog dog`
2. Create an object, `new Dog()`
3. Link the object and the reference `Dog dog = new Dog();`

Polymorphism allows us to link a superclass reference to a subclass object. Semantically this is because of the IS-A relationship. A dog IS-AN animal, and if Dog subclasses Animal, then `Animal animal = new Dog()` is valid Java.

A corollary of this is that methods can have polymorphic arguments and return types. The great of advantage is that if you write a method that accepts Animals, then any subclass object (Dogs, Cats, Mice) can be passed to your method, even objects of classes that don't yet exist! This means that the object programmer can use polymorphism to write general purpose code that can be extended at a later date.

8.4.2 Disadvantages of inheritance

In practice, inheritance trees are wide, but not deep. Although there are no limits to how much subclassing you can do, subclassing will increase the complexity of your

program by moving code away from where it is invoked, to a different class. This works against another good programming principle: *localisation*: keep all definitions close to where they are used.

8.5 Rules for overriding and overloading

Reading: pp. 190–191 of *HFJ*.

The public methods form a contract that subclasses must uphold. The compiler checks that a particular method can be called on that reference type, but at runtime, the JVM looks at the actual vehicle object on the heap. Overriding only works if method parameters are the same and return types are compatible. Methods must also not be less accessible; a private method cannot override a public one.

Method overloading does not involve polymorphism. Overloading is when two methods have the same name but different parameter lists. The return type can be different, but only if the parameter list has been changed as well. Accessibility is irrelevant because overloaded methods do not have to fulfil any contract.

8.6 Summary

- A subclass extends a superclass.
- A subclass inherits all public members of the superclass.
- Inherited methods can be overridden.
- Instance variables cannot be overridden, although they can be redefined in a subclass (not recommended).
- Use the IS-A test to verify that inheritance is valid.
- The lowest overridden method is invoked at runtime (dynamic binding).
- Overridden methods must have the same arguments, return types must be compatible and the method cannot be less accessible.
- Overloaded methods must have a different argument list.

8.7 Programming

Requirements

A graphic designer has asked you to write an application to generate arbitrary shapes. She hands you some examples of what she is after: Figures 8.1, 8.2, 8.3 and 8.4.

Analysis

Generalising from the examples, we suppose that shapes are closed figures composed of either straight or curved lines. An n-sided shape has n lines connecting n vertices. The vertices are placed at random locations on the drawing area. A shape may have its outline drawn in a single colour or filled with a single colour.

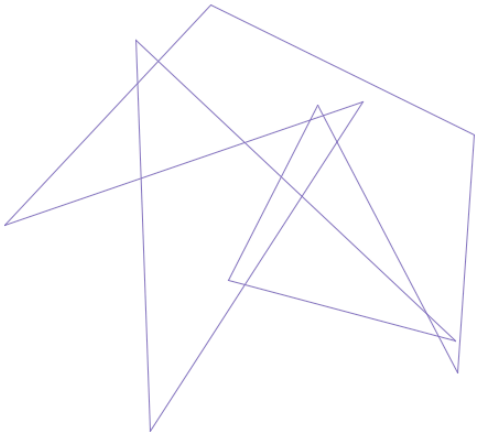

Figure 8.1: A 10-sided polygon.

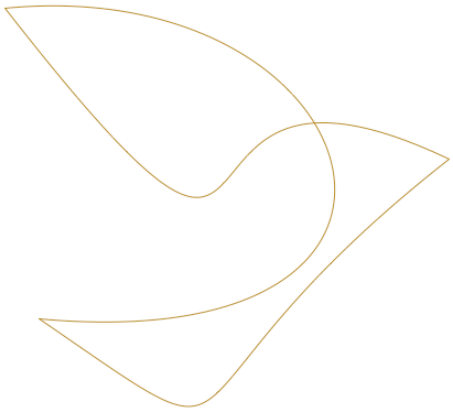

Figure 8.2: A 3-sided curvy shape.

An immediate programming problem is how to fill arbitrary shapes. We already know how to fill ovals using Graphic's `fillOval` method. Turning to the Java API for `java.awt.Graphics` we find an equivalent method for polygons (straight line shapes, according to our definition):
`fillPolygon(int[] xPoints, int[] yPoints, int nPoints)`.
This method fills a closed polygon defined by arrays of x and y coordinates.

Apart from ovals, we have not drawn shapes made of curved rather than straight lines. We notice that Graphics also has a `drawArc` method. However the API tells us that this method only draws outlines of ellipses or circles, and 'S' shaped curves are

Figure 8.3: A filled polygon.

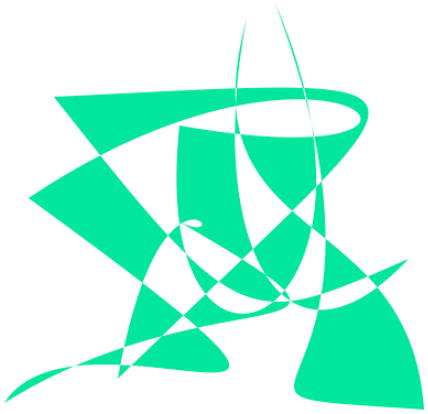

Figure 8.4: A filled curvy shape.

seen on the example sketch. Graphics has a subclass, `Graphics2D` which might be worth exploring because we know that subclasses offer more specialised behaviour.

The API tells us that there are `fill(java.awt.Shape s)` and `draw(java.awt.Shape s)` methods. Turning now to `java.awt.Shape s` we find that this is an *interface* and not a class. We will find out about interfaces in the next chapter, but for now we can just regard them as a second reference type. Unlike classes, however, interfaces cannot be instantiated, but they can be *implemented*. Implemented classes have the same type as their interface, as well as their own type (or types if they are in an inheritance tree).

The known implementing classes of `java.awt.Shape s` (i.e. the classes within the Java distribution, not counting all the other implementations that other developers might have made) include `GeneralPath`. A general path is made by moving to a start vertex, and then adding straight lines and, more appropriately, curves, to the next vertex. More lines can be added connecting the subsequent vertices.

`GeneralPath.curveTo()` allows us to add a cubic Bezier curve to the path. You might recall from earlier maths courses you have taken, that cubic curves can be S-shaped.

The object g that we have been using in the draw methods is in fact an instance of Graphics2D. (The API does not tell you this directly, unfortunately.) g, needs to be 'cast' to a Graphics2D reference before we can call methods on it.

Here is a code excerpt demonstrating how to create and draw a General Path:

```
int[] x, y // the vertices
int[]x1, y1, x2, y2 // control points
.
.
GeneralPath path =  new GeneralPath(GeneralPath.WIND_EVEN_ODD);
.
.
path.moveTo(x[0], y[0]);
for (int i = 0; i < nVertices - 1; i++) {
    path.curveTo(x1[i], y1[i], x2[i], y2[i], x[j], y[j]);
}
.
.
Graphics2D g2d = (Graphics2D) g;
g2d.draw(path);
```

Two control points parameterise the shape of the 'Bezier' curve that connects adjacent vertices.

Learning activity

We decide to implement straight-lined shapes first (i.e. polygons).

Design

A simple class hierarchy is immediately apparent: polygons and curvy shapes are specialisations of a `Shape` class.

Decide on a data structure for representing the points of a shape, *(x, y)*, and write class boxes for `Shape`, `Polygon` and `CurvyShape`.

8.8 Design

A vertex is a point *(x, y)*. One design possibility is to encapsulate a vertex in a class,

Vertex
int x int y
getX() setX(int x) getY() setY(int y)

However fillPolygon take arrays of *x* and *y* components. If we used a vertex class, we would have to place the components of each vertex in the polygon into an array each time fill is called. Alternatively we could maintain separate arrays of each component - but then we would not need a vertex class at all. In fact the vertex class does not have any interesting behaviour - the instance methods just access the internal data. This means that is not such a good class candidate.

Proceeding with the design, we develop Shape and Polygon class boxes:

Shape
int[] x int[] y int nPoints Color color
draw(Graphics g) fill(Graphics g)

Polygon extends Shape
draw(Graphics g) fill(Graphics g)

The curvy shape class needs its own instance variables; a general path, and arrays of control points, *(x1, y1)* and *(x2, y2)*.

CurvyShape extends Shape
GeneralPath path int[] x1, y1, x2, y2
draw(Graphics g) fill(Graphics g)

Learning activity

Implementation and Testing

Write Shape and Polygon classes and an application class to run your programs and generate random filled or drawn polygons.

8.9 Shape

```
package inheritance;

import java.awt.Color;
import java.awt.Graphics;

public class Shape {

  private int[] x, y;
  private int nVertices;
  private Color color = Color.BLACK;

  public Shape(int n) {

    setNVertices(n);
    setX(new int[n]);
    setY(new int[n]);
  }

  public Shape(int[] x, int[] y, int n) {

    this.x = x;
    this.y = y;
    nVertices = n;
  }

  public void draw(Graphics g) {
  }

  public void fill(Graphics g) {
  }

  public Color getColor() {
    return color;
  }

  public void setColor(Color c) {
    color = c;
  }

  public void setX(int[] x) {
    this.x = x;
  }

  public int[] getX() {
    return x;
  }

  public void setY(int[] y) {
    this.y = y;
  }

  public int[] getY() {
    return y;
  }
```

```
  public void setNVertices(int nVertices) {
    this.nVertices = nVertices;
  }

  public int getNVertices() {
    return nVertices;
  }
}
```

Two constructors initialise the instance variables `x`, `y` and `nVertices`. Notice how the constructors use setters for initialisation. This looks like an unnecessary complication... but what if we add code later on to limit the number of vertices? The code additions would occur in several places: in the constructors and the public setter. The principle of code reuse suggests therefore that initialisation is channelled through a single method.

The draw and fill methods are left empty so that subclasses can provide their own implementations.

The rest of the class definition is spent defining getters and setters for the four private instance variables.

8.10 Polygon

```
package inheritance;

import java.awt.Graphics;

public class Polygon extends Shape {

  public Polygon(int n) {

    super(n);
  }

  public Polygon(int[] x, int[] y, int n) {

    super(x, y, n);
  }

  public void draw(Graphics g) {

    g.setColor(getColor());
    g.drawPolygon(getX(), getY(), getNVertices());
  }

  public void fill(Graphics g) {

    g.setColor(getColor());
    g.fillPolygon(getX(), getY(), getNVertices());
  }
}
```

Polygon provides implementations of draw and fill using `Graphics.drawLine` and `Graphics.fill`.

`Graphics.fill` has to decide on what areas are 'inside' the polygon. It does this using the even-odd rule: a point is interior to the shape if a line drawn from the point crosses an odd number of polygon lines before escaping from the polygon altogether.

We have to write one application launcher for each shape; anticipating that applications will share code, we use inheritance to place common blocks in one class.

8.11 Shape application

```
package inheritance;

import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.MouseEvent;
import java.util.Random;

import goo.Goo;

public class ShapeApp extends Goo {

  private Shape shape;
  private Random random;
  private boolean fill = true;

  public ShapeApp(int w, int h, boolean tryFSE) {

    super(w, h, tryFSE);
    setRandom(new Random(2001));
  }

  public void init(int n) {
  }

  public void draw(Graphics g) {

    if (!fill)
      getShape().draw(g);
    else
      getShape().fill(g);
  }

  public void mouseClicked(MouseEvent e) {

    init(shape.getNVertices());
    repaint();
  }

  public void keyPressed(KeyEvent e) {

    char key = e.getKeyChar();
    if (key == 'f')
      fill = true;
    else if (key == 'd')
      fill = false;

    repaint();
  }
```

```
  public void setRandom(Random random) {
    this.random = random;
  }

  public Random getRandom() {
    return random;
  }

  public void setShape(Shape shape) {
    this.shape = shape;
  }

  public Shape getShape() {
    return shape;
  }
}
```

The initialisation of a particular shape will be coded in a subclass by overriding `init(int nVertices)`. ShapeApp does not have a main method because we intend to extend this class in order to draw polygons and other actual shapes. In the next chapter we will learn how to use the `abstract` keyword to make our intention more evident.

The application allows the user to generate a new random polygon by a mouse click, and to choose between fill and draw by pressing 'f' or 'd' keys.

8.12 Polygon application

```
package inheritance;

import java.awt.Color;
import java.util.Random;

import goo.Goo;

public class PolygonApp extends ShapeApp {

  PolygonApp(int w, int h, boolean tryFSE) {

    super(w, h, tryFSE);
  }

  public void init(int nVertices) {

    int width = getWidth();
    int height = getHeight();

    Random random = getRandom();
    int[] x = new int[nVertices];
    int[] y = new int[nVertices];

    for (int i = 0; i < nVertices; i++) {

      x[i] = random.nextInt(width);
      y[i] = random.nextInt(height);
    }
```

```
    setShape(new Polygon(x, y, nVertices));

    Color color = new Color(random.nextInt(255), random.nextInt
        (255),
        random.nextInt(255));
    getShape().setColor(color);
  }

  public static void main(String[] args) {

    Goo goo = new PolygonApp(500, 500, false);
    ((PolygonApp) goo).init(10);
    goo.noloop();
    goo.smooth();
    goo.go();
  }
}
```

The first statement in `main` constructs a Goo object. The third argument, `false`, tells Goo not to attempt full screen exclusive. If this argument is set to `true`, the drawing window will occupy the entire screen on your monitor, if your computer supports this option. If not, the window width and height will be set to the values of the first two arguments.

`noLoop` instructs Goo to draw the picture once, and not to enter the animation loop. Subsequent calls to `repaint` ask Goo to refresh the drawing window as a result of user interaction.

Learning activity

Write a curved shape class. (Hint. Refer back to the earlier code sample for an example of how to create and draw a general path.)

8.13 CurvyShape

```
package inheritance;

import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.GeneralPath;

public class CurvyShape extends Shape {

  GeneralPath path;
  int[] x1, y1, x2, y2;

  public CurvyShape(int n) {

    super(n);

    path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);

    x1 = new int[getNVertices()];
    y1 = new int[getNVertices()];
```

```
    x2 = new int[getNVertices()];
    y2 = new int[getNVertices()];
  }

  public CurvyShape(int[] x, int[] y, int[] x1, int[] y1, int[]
      x2, int[] y2,
       int n) {

    super(x, y, n);

    this.x1 = x1;
    this.y1 = y1;

    this.x2 = x2;
    this.y2 = y2;
    path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
    path = makeGeneralPath(x, y, x1, y1, x2, y2, n);

  }

  public GeneralPath makeGeneralPath(int[] x, int[] y, int[] x1,
      int[] y1, int[] x2, int[] y2, int n) {

    GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD)
        ;

    path.moveTo(x[0], y[0]);
    for (int i = 0; i < n; i++) {

      int j = (i + 1) % n;
      path.curveTo(x1[i], y1[i], x2[i], y2[i], x[j], y[j]);
    }
    return path;
  }

  public void draw(Graphics g) {

    g.setColor(getColor());
    Graphics2D g2d = (Graphics2D) g;
    g2d.draw(path);
  }

  public void fill(Graphics g) {

    g.setColor(getColor());
    Graphics2D g2d = (Graphics2D) g;
    g2d.fill(path);
  }
}
```

`makeGeneralPath` constructs a general path from arrays of vertices *x, y* and arrays of control points *(x1, y1), (x2 ,y2)*. A loop iterates around the path, joining *n* vertices with curved lines;

```
path.moveTo(x[0], y[0]);
for (int i = 0; i < n; i++) {

    int j = (i + 1) % n;
```

```
    path.curveTo(x1[i], y1[i], x2[i], y2[i], x[j], y[j]);
}
```

The modulus operator % is necessary since the last vertex, *(x[n-1], y[n-1])* connects to the first *(x[0], y[0])*.

Learning activity

Write a curvy shape application.

Hint: generate random vertices and random control points.

8.14 CurvyShapeApp

```
package inheritance;

import java.awt.Color;
import java.util.Random;

import goo.Goo;

public class CurvyShapeApp extends ShapeApp {

  CurvyShapeApp(int w, int h, boolean tryFSE) {

    super(w, h, tryFSE);
  }

  public void init(int nVertices) {

    int width = getWidth();
    int height = getHeight();

    Random random = getRandom();
    int[] x = new int[nVertices];
    int[] y = new int[nVertices];

    int[] x1 = new int[nVertices];
    int[] y1 = new int[nVertices];

    int[] x2 = new int[nVertices];
    int[] y2 = new int[nVertices];

    for (int i = 0; i < nVertices; i++) {

      x[i] = random.nextInt(width);
      y[i] = random.nextInt(height);
      x1[i] = random.nextInt(width);
      y1[i] = random.nextInt(height);
      x2[i] = random.nextInt(width);
      y2[i] = random.nextInt(height);
    }

    setShape(new CurvyShape(x, y, x1, y1, x2, y2, nVertices));
```

```
    Color color = new Color(random.nextInt(255), random.nextInt(
        255), random.nextInt(255));

    getShape().setColor(color);
  }

  public static void main(String[] args) {

    Goo goo = new CurvyShapeApp(500, 500, false);
    ((CurvyShapeApp) goo).init(10);
    goo.noloop();
    goo.smooth();
    goo.go();
  }
}
```

The application extends ShapeApp and overrides `init` to initialise the control points and construct a curvy shape.

Learning activity

The designer is happy with our applications, but it occurs to her that our shapes might move! Rather than translate across the screen, she wishes to explore the effect of moving each vertex independently.

Subclass Polygon and CurvyShape as MovingPolygon and MovingCurvyShape.

Write application programs to test your coding.

8.15 MovingPolygon

```
package inheritance;

public class MovingPolygon extends Polygon {

  int[] vx, vy;

  public MovingPolygon(int n) {

    super(n);
    vx = new int[getNVertices()];
    vy = new int[getNVertices()];
  }

  public MovingPolygon(int[] x, int[] y, int vx[], int[] vy, int
      n) {

    super(x, y, n);
    this.vx = vx;
    this.vy = vy;
  }

  public void move(int left, int right, int top, int bottom) {

    for (int i = 0; i < getNVertices(); i++) {
```

```
      getX()[i] += vx[i];
      getY()[i] += vy[i];

      if (getX()[i] >= right) {

        vx[i] *= -1;
        getX()[i] = right;

      } else if (getX()[i] <= left) {

        vx[i] *= -1;
        getX()[i] = left;
      }

      if (getY()[i] >= bottom) {

        vy[i] *= -1;
        getY()[i] = bottom;

      } else if (getY()[i] <= top) {

        vy[i] *= -1;
        getY()[i] = top;
      }

    }
  }
}
```

`MovingPolygon` subclasses Polygon; the new behaviour is movement and is coded in `move`. New instance variables have been added for x and y velocity components of each vertex. The polygon vertices move within a rectangle with bounds specified by `left, right, top, bottom`. This gives greater freedom to experiment with the animation; movement can take place within a rectangle other than the window itself.

8.16 MovingPolygonApp

```
package inheritance;

import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;

public class MovingPolygonApp extends PolygonApp {

  int maxSpeed;

  MovingPolygonApp(int w, int h, boolean tryFSE, int ms) {

    super(w, h, tryFSE);
    maxSpeed = ms;
  }

  public void init(int nVertices) {

    int width = getWidth();
    int height = getHeight();
```

```
    int[] vx = new int[nVertices];
    int[] vy = new int[nVertices];

    Random random = getRandom();
    int[] x = new int[nVertices];
    int[] y = new int[nVertices];

    for (int i = 0; i < nVertices; i++) {

      x[i] = random.nextInt(width);
      y[i] = random.nextInt(height);
      vx[i] = 1 + random.nextInt(maxSpeed - 1);
      if (random.nextDouble() > 0.5)
        vx[i] *= -1;

      vy[i] = 1 + random.nextInt(maxSpeed - 1);
      if (random.nextDouble() > 0.5)
        vy[i] *= -1;
    }

    setShape(new MovingPolygon(x, y, vx, vy, nVertices));

    Color color = new Color(random.nextInt(255), random.nextInt
        (255),
        random.nextInt(255));
    getShape().setColor(color);
  }

  public void draw(Graphics g) {

    super.draw(g);
    MovingPolygon movingPoly = (MovingPolygon) getShape();
    movingPoly.move(0, getWidth(), 0, getHeight());
  }

  public static void main(String[] args) {

    MovingPolygonApp goo = new MovingPolygonApp(500, 500, false,
        10);
    goo.init(100);
    goo.smooth();
    goo.go();
  }
}
```

The application overrides `init` to ensure that a MovingPolygon is constructed. The velocity components of each vertex, `vx, vy` are randomised to one of {*-maxSpeed, ... -2, -1, 1, 2, ... maxSpeed*}

`draw` has three statements. `super.draw(g);` is the first, and it calls draw on the superclass, which in this case is PolygonApp. The call is transferred to PolygonApp's superclass, ShapeApp, since PolygonApp does not override `draw`.

`getShape()` returns a `Shape` reference. `Shape` does not define a move method, so `getShape.move` would not pass the compilers static type checking. We must cast the type of the returned reference to `MovingPolygon`,

```
(MovingPolygon}getShape();
```

In general, casting a reference to a subclass type is risky; we must be certain that the object the reference variable points to is of this subclass (or a subclass of the subclass). We shall see a tidier way of doing this when we study interfaces in the next chapter.

Finally, draw calls move on the MovingPolygon object.

8.17 Moving Curvy Shape

```
package inheritance;

public class MovingCurvyShape extends CurvyShape {

  int[] vx, vy;

  public MovingCurvyShape(int n) {

    super(n);
    vx = new int[getNVertices()];
    vy = new int[getNVertices()];
  }

  public MovingCurvyShape(int[] x, int[] y, int[] x1, int[] y1,
      int[] x2,
       int[] y2, int[] vx, int[] vy, int n) {

    super(x, y, x1, y1, x2, y2, n);
    this.vx = vx;
    this.vy = vy;
  }

  public void move(int left, int right, int top, int bottom) {

    for (int i = 0; i < getNVertices(); i++) {

      getX()[i] += vx[i];
      getY()[i] += vy[i];

      if (getX()[i] >= right) {

        vx[i] *= -1;
        getX()[i] = right;

      } else if (getX()[i] <= left) {

        vx[i] *= -1;
        getX()[i] = left;
      }

      if (getY()[i] >= bottom) {
```

```
        vy[i] *= -1;
        getY()[i] = bottom;

      } else if (getY()[i] <= top) {

        vy[i] *= -1;
        getY()[i] = top;
      }

    }

    path = makeGeneralPath(getX(), getY(), x1, y1, x2, y2,
        getNVertices());

  }

}
```

A MovingCurvyShape is just like a CurvyShape except that is has a move method, and instance variables for velocity components. Notice how `move` has to reconstruct the path by calling `makeGeneralPath`; there is no way to access the vertices of a Java path.

8.18 Moving Curvy Shape App

```
package inheritance;

import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;

public class MovingCurvyShapeApp extends CurvyShapeApp {

  int maxSpeed;

  MovingCurvyShapeApp(int w, int h, boolean tryFSE, int ms) {

    super(w, h, tryFSE);
    maxSpeed = ms;
  }

  public void init(int nVertices) {

    int width = getWidth();
    int height = getHeight();

    Random random = getRandom();
    int[] x = new int[nVertices];
    int[] y = new int[nVertices];
    int[] x1 = new int[nVertices];
    int[] y1 = new int[nVertices];
    int[] x2 = new int[nVertices];
    int[] y2 = new int[nVertices];
    int[] vx = new int[nVertices];
    int[] vy = new int[nVertices];

    for (int i = 0; i < nVertices; i++) {
```

```
      x[i] = width / 3 + random.nextInt(width / 3);
      y[i] = height / 3 + random.nextInt(height / 3);
      x1[i] = random.nextInt(width);
      y1[i] = random.nextInt(height);
      x2[i] = random.nextInt(width);
      y2[i] = random.nextInt(height);

      vx[i] = 1 + random.nextInt(maxSpeed - 1);
      if (random.nextDouble() > 0.5)
        vx[i] *= -1;

      vy[i] = 1 + random.nextInt(maxSpeed - 1);
      if (random.nextDouble() > 0.5)
        vy[i] *= -1;
    }

    setShape(new MovingCurvyShape(x, y, x1, y1, x2, y2, vx, vy,
        nVertices));

    Color color = new Color(random.nextInt(255), random.nextInt
        (255),
        random.nextInt(255));

    getShape().setColor(color);
  }

  public void draw(Graphics g) {

    super.draw(g);

    MovingCurvyShape movingCurly =  (MovingCurvyShape)getShape();
    movingCurly.move(0, getWidth(), 0, getHeight());
  }

  public static void main(String[] args) {

    MovingCurvyShapeApp goo = new MovingCurvyShapeApp(500, 500,
        true, 10);
    goo.init(100);

    goo.smooth();
    goo.frameRate(24);
    goo.go();
  }
}
```

This application looks long but a lot of the code is occupied with randomising the vertices and the control points. The main method makes it easy to experiment with different numbers of vertices, speeds and animation speeds. These numbers may need to be altered depending on the capability of your machine. Notice that the animation runs in full screen exclusive. (If your machine is spending a lot of time drawing to the screen, it may take a while before the application terminates after you press 'escape' (or Cntrl-C or Cntrl-Q)).

WARNING. Do no worry if you have not managed to get this far on your own. This is an advanced bit of Java programming. Just sit back and enjoy the animation!

8.19 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- understand inheritance as one of the defining characteristics of object programming
- understand the relationship between subclasses and superclasses
- describe how inherited methods can be overridden
- understand that instance methods cannot be overridden
- use the IS-A test to verify that inheritance is valid
- understand that the lowest overridden method is invoked at runtime (using dynamic binding)
- understand that overridden methods must have the same arguments, return types must be compatible and the method cannot be less accessible
- understand that overloaded methods must have a different argument list.

Chapter 9

Abstraction

Essential reading

HFJ Chapter 8.

9.1 Introduction

We encountered polymorphism in the last chapter when we saw how to reference an object by a superclass reference variable. Polymorphism means *many forms*. Although a `Cat` may appear as an `Animal`,

```
Animal pet = new Cat();
pet.eat();
```

could a `Cat` also appear to be a `Car`? If a cat and car share some behaviour then under certain contexts they might be equivalent. For example, in an animation of a street, both cats and cars move:

```
Car car = new Car();
car.move();
cat.move();
```

This chapter shows you how a new reference type – the `Interface` – can enable this sort of polymorphism. You will also learn about abstract classes and abstract methods, and discover how these can be used within an inheritance tree.

9.2 Abstract Classes

Reading: pp. 197–202 of *HFJ*.

In order to introduce interfaces we need to understand what it means to declare a class abstract, and why we should want to do this.

The `Animal` hierarchy on page 198 of *HFJ* has a strange problem: what does it mean to have an `Animal` object?

An animal is a concept rather than an actual thing. Some objects in the world breathe and eat and move around; collectively we call these objects 'animals'. We can later modify our concept – perhaps animals don't need to move. But a wolf is a wolf. A wolf is not a theoretical concept, it's an actual thing.

We may wish to preserve this distinction in our programming. After all what might it mean to instantiate an animal? To prevent such semantic accidents, we can mark these classes as `abstract`. It is possible to have an abstract reference, but the referenced object must be of a non-abstract type. Non-abstract classes are also known as concrete classes.

```
abstract class Animal{
    // class definition
}
class Cat{
    // class definition
}
class App{
    ...
    main(){
        // OK - an abstract reference to an object of a concrete type
       Animal pet = new Cat();

       // won't compile - cannot instantiate an abstract class
       // Animal animal = new Animal();
       // Cat patch = new Animal();
      }
}
```

Classes that are not meant to be instantiated are declared `abstract`. The compiler will complain if your code tries to instantiate an abstract class.

9.3 Abstract Methods

Reading: pp. 203–207 of *HFJ*.

Although an abstract class cannot be instantiated, it can be extended, so that instances of a subclass can exist at runtime. There are lots of abstract classes in the Java API; take a look at the Swing library for example. An abstract class might contain an *abstract* method, and any abstract method must be contained in an abstract class. An abstract method must be implemented by a concrete subclass.

Abstract methods define a protocol – the public interface – of the class.

An abstract method has no body. The declaration ends with a semicolon.

Interface has (at least two) meanings in Java. The interface of a class is the set of public methods that a class defines, and these are the possible messages that an object of the class can receive. The API documents the public interface of the library classes.

An `interface` is a type. There are five reference types in Java 5: classes, interfaces, arrays, enums and annotations. The last two will not be covered in this course.

Before we discover what an `interface` is, we must learn about the ultimate superclass of all.

9.4 A class called Object

Reading: pp. 206–217 of *HFJ*.

It may surprise you to learn that there is an Object class, but more disturbingly there is also a class called Class! We won't be exploring Class here, but Object is worth a look.

You will now have seen how a list of Dogs can be generalised to hold a list of any Animal. But can it be generalised still further? Every class extends Object. This happens automatically and we don't need to write in extends Object. Any class that doesn't explicitly extend another class is regarded by the compiler as implicitly subclassing Object.

Object represents the lowest common denominator of all classes (since all classes are a specialisation of Object). So what is it that all classes have in common? You can find this answer in the API and page 209 of *HFJ* shows the most useful methods of Object.

The generalisation we have been seeking is a list of Objects. Such an object would be capable of storing anything. Happily, it already exists - it is the ArrayList<Object>. An object of any class may go into ArrayList<Object>, but it will always be retrieved as an Object.

One of the jobs of the compiler is to check reference type. The compiler decides whether a method can be called on an object based on the reference type. So in the code chunk

```
class Dog{
    ...
    public void bark(){
    ...
    }
}

class App{
    ...
    ...main(){
        ArrayList<Object> list = new ArrayList<Object>;
        Dog bongo = new Dog();
        list.add(bongo);
        ...
        Dog fido = list.get(0); // won't compile
        ...
        Object fido = list.get(0); // OK
        fido.bark(); // won't compile
    }
}
```

We can't ask fido to bark because its remote control is an Object controller, even though the retrieved object actually is a Dog and the Dog class has a bark method.

An object contains everything that it inherits from its superclasses. There is only one

object on the heap. Think of the `Object` part of any object being contained in the centre of the object blob, as shown on page 214 of *HFJ*.

So what is the use of `ArrayList<Object>` if we cannot retrieve the full object? Actually we can, with a cast. If we know that `ArrayList<Object>` only contains `Dogs` we can cast the object to a Dog:

```
ArrayList<Object> objList = new ArrayList<Object>;
Dog bongo = new Dog();
objList.add(bongo);
...
Dog fido = (Dog)objList.get(0); // cast the object to a Dog
fido.bark(); // fine. fido barks
```

If the object at `objList(0)` is not a Dog, the cast will fail at runtime. You can use the `instanceof` operator if there is any doubt (pg. 216 of *HFJ*).

If the list is already parameterized, then the compiler will put in the casts automatically:

```
ArrayList<Dog> dogList = new ArrayList<Dog>;
Dog bongo = new Dog();
dogList.add(bongo);
...
Dog fido = list.get(0); // cast inserted by the compiler
fido.bark(); // fido barks
```

9.5 Changing the contract

Reading: pp. 218–223 of *HFJ*.

We may want to use the Animal class structure for another application, for a Pet Shop simulation, for example. Pets have special behaviours, such as `play` and `beFriendly`, but not all animals are, or could be, pets. *HFJ* suggests three options, none of which is satisfactory. One solution uses two superclasses, as in the class diagram on page 222. However multiple inheritance has a particularly bad property – -an ambiguity known as the Deadly Diamond of Death.

9.6 The `Interface`

Reading: pp. 224–227 of *HFJ*.

The interface gives you the benefits of multiple inheritance, but avoids the ambiguity problem because all the methods are abstract. The subclass must implement all the abstract methods so at runtime the JVM won't be confused between which method of the two inherited versions it has to call.

Look carefully at pp. 224 – 225 of *HFJ* to see how an interface type is implemented in Java. The interface allows for even greater polymorphism because a class might have various behaviours, each one associated with an interface (a class can implement any number of interfaces). A method that accepts an interface type is

extremely flexible because, just as long as the interface's abstract methods are implemented in a particular class, an object of this class can be passed to the method. The interface establishes a protocol and the method knows it can call any method from this protocol on the object. In this way we can code for *behaviours* rather than for members of a particular superclass.

How can you tell when to use inheritance, an abstract class, or an interface?

If a class doesn't pass an IS-A test for another type, don't extend anything; otherwise, when you find a specialisation of a particular class, then subclass.

Use abstract classes when you wish to define a template and there is some code that all subclasses may use; or in order to guarantee that instantiations of this class are impossible.

Use an interface in order to define a role that other classes may play, regardless of their place in any inheritance tree.

9.7 Invoking a superclass method

Reading: pg. 228 of *HFJ*.

Sometimes a superclass method is almost what you want. You just want to extend the superclass method with some extra code. In that case you can override the superclass method, but make a call to this method using the `super` syntax. Study the example on page 228 of *HFJ*.

9.8 Summary

- An abstract class cannot be instantiated.
- An abstract class can have abstract and non-abstract methods.
- If a class has one or more abstract methods, it must be declared abstract.
- An abstract method has no body, and the declaration ends in a semicolon.
- All abstract methods must be implemented in the first concrete class in the inheritance tree.
- All classes are sublasses of `Object`.
- Methods can be declared with Object arguments and/or return types.
- Method calls can only be made if the methods are in the class (or interface) of the reference variable type, regardless of the object type.
- A reference variable can be cast to a subtype in order to make a call to a subtype method, but at runtime the cast will fail if the object on the heap is not compatible with the cast, e.g.

  ```
  Animal pet  = new Cat();
  ((Cat)pet).miaow();
  ```

- Java only supports single inheritance.
- Languages with multiple inheritance have special rules to disambiguate method calls. This makes programming more difficult.
- An interface defines only abstract, public methods.

- An interface definition uses the keyword `interface` in place of `class`.
- A class may implement one or more interfaces by using the keyword `implements`.
- Interface implementation means that the class must provide method bodies for all the interface methods.
- Use `super` to invoke a superclass method of an overriden method.

9.9 Programming

A class diagram for the Shape inheritance tree developed in the previous chapter is shown in the figure below. (An open-headed arrow joins sub- and superclass.)

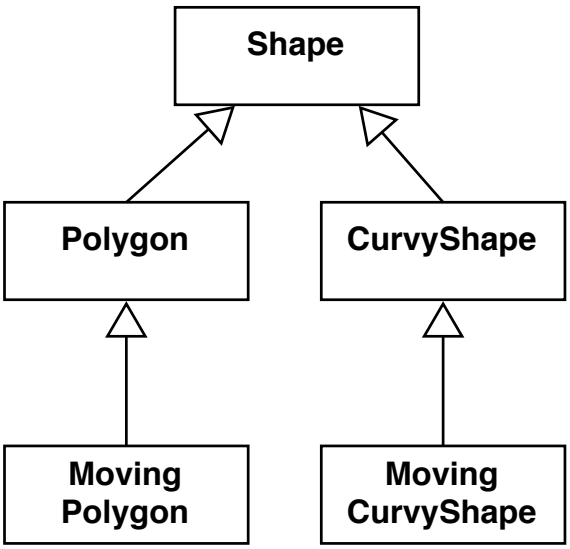

Figure 9.1: The Shape class hierarchy.

With the current design, a Shape can be instantiated but draw and fill methods are empty; calls to these methods are allowed, but nothing will happen. After all, what does a shape look like? A shape is an idea, rather than a tangible thing such as a polygon, or curvy lines. It seems therefore that Shape should be declared abstract.

Objects from the inheritance package may also move. The (concrete) shapes are `drawable` and `moveable`. If you can find a shared behaviour amongst a group of classes, it is worth thinking if interfaces can simplify your design. Class candidates may be suggested by nouns from the requirement specification; possible Interfaces may be suggested by verbs.

For example, we might be developing an application which displays shapes and various other things such as text, cats, cars... All these need to be drawn. The application might look something like:

```
class App extends Goo{
    ...
    ArrayList<Shape> shapes = ...
```

```
    ArrayList<Animal> animals = ...
    ArrayList<Vehicles> vehicles = ...
    ...
    public void draw(Graphics g) {

        for (Shape s : shapes) {
            s.draw(g);
        }

        for (Animal a : animals) {
            a.draw(g);
        }

        for (Vehicle v : vehicles) {
            v.draw(g);
       }
    }
}
```

This would work, but what if we later wish to add a new type of drawable object? We would have to edit and recompile the code to include a new list, and a new for-loop in `draw`, to iterate over this list. If we were working in a team, or had released our classes on the Internet, we would them have to circulate the revised application so that everyone is up to date.

It would be much simpler if we – -or indeed other developers – -could add objects without having to edit and recompile. This is where interfaces come in.

We define a `Drawable` interface with an abstract `draw(Graphics g)` method:

```
ArrayList<Drawable> drawables = new ArrayList<Drawable>();
```

and include an `add(Drawable d)` to the application. The application then looks like:

```
class App extends Goo{
    ...
    ArrayList<Drawable> drawables = ...
    ...
    public void add(Drawable d){

        drawables.add(d);
    }
    ...
    public void draw(Graphics g) {

        for (Drawable d : shapes) {

            s.draw(g);
        }
    }
}
```

Other developers can define classes that you don't know about, and add objects of these types to the application without worrying about editing and recompiling – just as long as these classes implement Drawable. The calls to `draw` will not fail because

the compiler has checked that only classes implementing the drawable interface can be added to the list, and any class implementing drawable must define a draw method (the compiler checks this too).

Learning activity

A modified class diagram for the inheritance package is shown below. (The dashed arrow designates the `implements` relationship.)

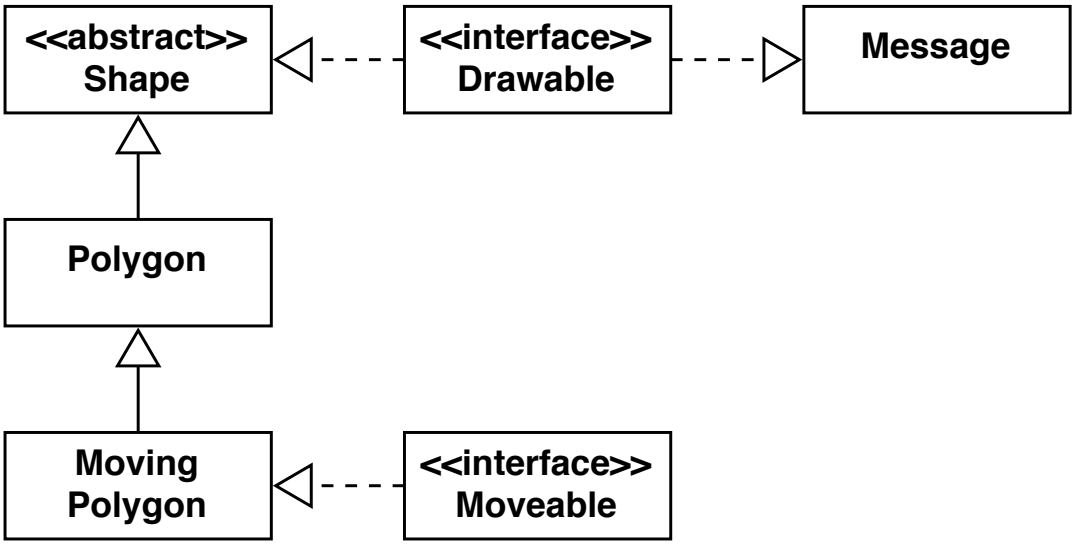

Figure 9.2: Class diagram showing Moveable and Drawable interfaces.

Notice that a moveable interface has been included, and a new class, `Message` has been added. This class stores a message string and draws it at a specific location on the window.

The programming task for this chapter is to implement this class diagram. You should start by writing `Drawable` and `Moveable` interfaces, and then rewrite `Shape` so that it is abstract. Declare an abstract fill method in `Shape`.

Decide which classes should implement which interfaces and make the necessary adjustments.

Write a Message class and add it to your package.

Finally, write an application class to test your classes.

9.10 Implementing the Shape class diagram

From the class diagram, we note that Shape and Message must implement Drawable, and that MovingPolygon must implement Moveable.

```
public interface Drawable {
    public abstract void draw(Graphics g);
}

public interface Moveable {
    public abstract void move(int left, int right, int top, int bottom);
}
```

Shape is declared abstract and has a single abstract method, fill.

```
public abstract class Shape implements Drawable{
    ...
    public abstract void fill(Graphics g);
    ...
}
```

Polygon does not need to be changed, but MovingPolygon must implement Moveable, as well as extend Polygon.

```
public class MovingPolygon extends Polygon implements Moveable{
     ...
}
```

From the API, we see that Graphics has a drawString method and this can be used to define a simple draw method in `Message` ,

```
public class Message implements Drawable {

    int x;
    String message;

    public void draw(Graphics g) {

        g.setColor(Color.BLACK);
        g.drawString(message, (int) (x + 0.5), 50);
    }
}
```

The resulting package, complete with a test application, is printed in the next few pages.

9.11 Drawable and Moveable

```
package abstractclassesandinterfaces;

import java.awt.Graphics;

public interface Drawable {

  public abstract void draw(Graphics g);
}
```

```
package abstractclassesandinterfaces;

public interface Moveable {
  public abstract void move(int left, int right, int top, int
      bottom);
}
```

`move` has four parameters, to allow for movement in a smaller (or even bigger) rectangle than the actual window.

9.12 Shape

```
package abstractclassesandinterfaces;

import java.awt.Color;
import java.awt.Graphics;

public abstract class Shape implements Drawable{

  protected int[] x, y;
  protected int nVertices;
  protected Color color = Color.BLACK;

  public Shape(int n) {

    nVertices = n;
    x = new int[n];
    y = new int[n];
  }

  public Shape(int[] x, int[] y, int n) {

    this.x = x;
    this.y = y;
    nVertices = n;
  }

  public abstract void fill(Graphics g);

  public Color getColor(){

    return color;
  }
```

```
  public void setColor(Color c){

    color = c;
  }
}
```

Shape implements Drawable, but surprisingly, we do not need to supply an implemented draw. Shape is abstract; an instance cannot be created, so the problem of not having a concrete implementation will never arise. However draw must be implemented by the first concrete subclass lying below Shape in the inheritance tree.

9.13 Polygon

```
package abstractclassesandinterfaces;

import java.awt.Graphics;

public class Polygon extends Shape {

  public Polygon(int n) {

    super(n);
  }

  public Polygon(int[] x, int[] y, int n) {

    super(x, y, n);
  }

  public void draw(Graphics g) {

    g.setColor(color);

    for (int i = 0; i < nVertices - 1; i++) {

      g.drawLine(x[i], y[i], x[i + 1], y[i + 1]);
    }

    g.drawLine(x[nVertices - 1], y[nVertices - 1], x[0], y[0]);
  }

  public void fill(Graphics g){

    g.setColor(color);
    g.fillPolygon(x, y, nVertices);
  }
}
```

Polygon, which is a concrete subclass of the abstract `Shape implements Drawable`, *must* provide a code body for `draw`. It must also provide an implementation of `fill`, which was declared abstract in Polygon's superclass, Shape.

9.14 Message

```
package abstractclassesandinterfaces;

import java.awt.Color;
import java.awt.Graphics;

public class Message implements Drawable {

  int x;
  String message;

  public Message(String msg, int x) {

    message = msg;
    this.x = x;
  }

  public void draw(Graphics g) {

    g.setColor(Color.BLACK);
    g.drawString(message, (int) (x + 0.5), 50);
  }
}
```

9.15 Moving Polygon

```
package abstractclassesandinterfaces;

public class MovingPolygon extends Polygon implements Moveable{

  int[] vx, vy;

  public MovingPolygon(int n) {

    super(n);
    vx = new int[nVertices];
    vy = new int[nVertices];
  }

  public MovingPolygon(int[] x, int[] y, int vx[], int[] vy, int
      n) {

    super(x, y, n);
    this.vx = vx;
    this.vy = vy;
  }

  public void move(int left, int right, int top, int bottom) {

    for (int i = 0; i < nVertices; i++) {

      x[i] += vx[i];
      y[i] += vy[i];
```

```
      if (x[i] >= right) {

        vx[i] *= -1;
        x[i] = right;

      } else if (x[i] <= left) {

        vx[i] *= -1;
        x[i] = left;
      }

      if (y[i] >= bottom) {

        vy[i] *= -1;
        y[i] = bottom;

      } else if (y[i] <= top) {

        vy[i] *= -1;
        y[i] = top;
      }

    }
  }
}
```

As required by the contract `implements Moveable`, MovingPolygon, which is concrete, must supply a method body for `move`.

9.16 Shape and Message application

```
package abstractclassesandinterfaces;

import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;
import java.util.Random;

import goo.Goo;

public class App extends Goo {

  ArrayList<Moveable> movingObjects;
  ArrayList<Drawable> drawingObjects;

  Random random;
  public App(int w, int h) {

    super(w, h);
    random = new Random();

    movingObjects = new ArrayList<Moveable>();
    drawingObjects = new ArrayList<Drawable>();

    int nVertices = 10;
    int maxSpeed = 10;

    int width = getWidth();
```

```
    int height = getHeight();
    int[] x = new int[nVertices];
    int[] y = new int[nVertices];
    int[] vx = new int[nVertices];
    int[] vy = new int[nVertices];

    for (int i = 0; i < nVertices; i++) {

      x[i] = random.nextInt(width);
      y[i] = random.nextInt(height);
      vx[i] = 1 + random.nextInt(maxSpeed - 1);
      if (random.nextDouble() > 0.5)
        vx[i] *= -1;

      vy[i] = 1 + random.nextInt(maxSpeed - 1);
      if (random.nextDouble() > 0.5)
        vy[i] *= -1;
    }

    MovingPolygon poly = new MovingPolygon(x, y, vx, vy,
        nVertices);

    Color color = new Color(random.nextInt(255), random.nextInt
        (255),
        random.nextInt(255));
    poly.setColor(color);

    drawingObjects.add(poly);
    movingObjects.add((Moveable)poly);

    Message drawString = new Message("You are watching a moving
        polygon", 50);
    drawingObjects.add(drawString);
  }

  public void draw(Graphics g) {

    for (Drawable obj : drawingObjects) {
      obj.draw(g);
    }
    for (Moveable obj : movingObjects) {
      obj.move(0, getWidth(), 0, getHeight());
    }
  }

  public static void main(String[] args) {

    Goo goo = new App(500, 500);
    goo.smooth();
    goo.go();
  }
}
```

The application looks long, but most of the code is taken up with object initialisation. Notice how `draw` iterates through a list of Drawables, asking each object to draw itself, and a list of Moveables, asking each object to move.

9.17 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- understand that an abstract class cannot be instantiated, and that it can have abstract and non-abstract methods
- decide when a class must be declared abstract
- code an abstract method correctly, with no body and ending the declaration with a semi-colon
- describe how all abstract methods must be implemented in the first concrete class in the inheritance tree
- explain that all classes are subclasses of *object*
- describe how methods can be declared and how method calls can be made
- describe how a reference variable can be cast to a subtype
- understand that Java only supports single inheritance
- understand that languages with multiple inheritance have special rules to disambiguate method calls, and that this makes programming more difficult
- explain that an interface defines only abstract, public methods
- understand that an interface definition uses the keyword `interface` in place of `class`
- describe how a class may implement one or more interfaces by using the keyword `implements`
- describe how interface implementation means that the class must provide method bodies for all the interface methods
- know how to use `super` to invoke a superclass method of an overridden method.

Chapter 10

Object lifetime

Essential reading

HFJ Chapter 9.

10.1 Introduction

Objects are born, they live for a while, and eventually they are abandoned. They are born by invoking the class constructor. They have a useful life only insofar as they receive messages (methods are invoked) and send messages (invoke methods on other objects). Java itself does not impose any limits on how many objects your programme might create – but the operating system does, because RAM is a finite resource. However, when objects are no longer useful to your programme, they are cleared from RAM by the *garbage collector*. This frees up RAM for further object creation.

10.2 The stack and the heap

Reading: pp. 236–239 of *HFJ*.

The JVM requests a chunk of RAM from the operating system especially for your Java programme. This memory is managed by the JVM; as you have seen, the JVM divides RAM into a 'stack' and a 'heap'. Method invocations and local variables are placed in the orderly stack, all objects and instance variables live in the unordered, garbage collectible, heap.

Instance variables are the fields that each object has. They are the object state and are declared outside the object's methods. Instance variables live (inside the object, on the heap) as long as the object does. Local variables, which include method parameters, are declared inside methods, and only exist while the method is on the stack.

Methods are stacked on the call stack in *stack frames*. The stack frame contains the method code, values of local variables and a pointer to which line of code is executing. When methods are returned, they are removed from the stack and the method immediately below continues execution. Study the example on page 237.

Local variables might reference an object. These variables exist in the stack, just like all local variables, but the objects that they reference will live on the heap, along with all the other objects, regardless of whether the reference (or references) to them are local or instance variables.

But why should we need to know all of this?

We must understand the stack and the heap if we are to grasp issues such as *scope*, *object creation*, *memory management*, *threads* and *exception handling*. And without this understanding, these programming issues will remain obscure – they will become recipes, and we cannot become good programmers if we can only follow recipes.

10.3 Object creation

Reading: pp. 240–249 of *HFJ*.

Look at the three steps of object declaration, creation and assignment on page 240 of *HFJ*. The constructor is a special block of code that instantiates the object. It looks like a method and executes when the JVM hits the `new` keyword.

If you don't write a constructor, the compiler will put a default constructor in the class file for you.

A major use of the constructor is for initialisation of instance variables. Otherwise we might have to rely on the programmer remembering to set object state using the setter methods. In other words, the constructor should make sure that the object is ready for use. The constructor can be parameterised, and these parameters can be used to set the object state. Look at the Duck example on page 244 of *HFJ*.

A default, no argument, constructor can prepare the object with a standard state. An *overloaded* constructor can prepare the object in a particular state, as needed by the creating object.

The compiler only makes a default constructor if you don't write any constructors for your class.

10.4 Superclass constructors

Reading: pp. 250–255 of *HFJ*.

If an object is a subclass, then all its inherited parts – variables and methods – are contained in the object. This is accomplished by constructor chaining. Each subclass invokes its super class constructor. The constructors themselves are stacked – see the diagram on page 252. This will happen automatically, or you can call a superclass constructor explicitly by placing a call to `super()` as the first statement of your constructor. Parents must exist before their children.

The value of including an explicit call to `super` is really when you wish to build the superclass object in a particular state, using a parameterised superclass constructor. Study the Animal-Hippo example on page 255 of *HFJ*.

10.5 `this()`

Reading: pp. 256–257 of *HFJ*.

You will find that overloaded constructors may use some identical code, and you know that we should place common code in a single place. Happily, Java allows you to call one overloaded constructor from another using `this(parameter-list)`. Once more, `this` must be the first statement in a constructor – which incidentally implies that a constructor can call `this` or `super`, but never both.

10.6 Object lifespan

Reading: pp. 258–265 of *HFJ*.

We already know that an object's life depends on its references. However a reference variable is either an instance or a local variable, and this brings us to the tricky concept of *scope*.

A local variable is alive only as long as its stack frame is on the stack, i.e. until the method completes and returns. However a variable can only be used when it is in scope, in which case it will be on top of the stack. Look at page 259 of *HFJ*.

Any reference variable or primitive can only be used when in scope. An object is only alive while some other object has a live variable that refers to it, whether or not this variable is currently in scope. Otherwise it is ripe for destruction by the garbage collector.

There are three ways to abandon an object:

1. The reference goes out of scope permanently.
2. The reference is reassigned.
3. The reference is set to `null`.

10.7 Summary

- The JVM maintains two important areas of memory, the stack and the heap.
- Instance variables are declared inside a class but outside any method.
- Local variables are declared inside a method or method parameter.
- All local (primitive or reference type) variables live on the stack, in the frame corresponding to the method where the variables are declared.
- All objects live on the heap, regardless of whether the reference is a local variable or an instance variable.
- Instance variables live within their object on the heap.
- If the instance variable is a reference to an object, both the reference and the object it refers to are on the heap.
- A constructor has the same name as the class, but has no return type.
- The constructor can be used to initialise the state of the object.

- The compiler will supply a default constructor if you don't define any constructor.
- But if you do define any constructor, the compiler will not build the default constructor.
- It is good practice to provide a default constructor and default values for the instance variables.
- Constructors may be overloaded and the normal rules for overloaded method argument lists apply.
- Instance variables are initialised to default values 0/0.0/false/null

10.8 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- describe how the JVM maintains two important areas of memory, the stack and the heap
- describe how instance variables are declared inside a class but outside any method
- describe how local variables are declared
- explain where all local variables, instance variables and objects live – whether on the stack or on the heap
- describe what constructors are and how they work
- explain how to overload constructors
- explain that instance variables are initialised to default values
- explain how a constructor can be used to initialise the state of an object
- describe how the compiler supplies a default constructor if one is not defined explicitly by the program.

Chapter 11

Events

Essential reading

HFJ Chapter 12.

11.1 Introduction

Although we have been running our applications from the command line, this is not always an ideal way of running and interacting with programs. GUIs allow us to change programme state by pressing buttons, moving sliders, choosing items from a menu, etc. In this chapter you will find out about the Java Swing package which allows us to draw to a window, and how to add 'widgets' such as buttons and sliders, to this window.

11.2 Putting a widget on a window

Reading: pp. 353–355 of *HFJ*.

A `JFrame` represents a window on the screen. It can have a menu bar, menu items, clickable buttons, sliders and all the usual means of graphical interaction that you are familiar with. The statement:

```
JFrame frame = new JFrame();
```

creates a JFrame. Widgets (buttons, text fields, etc.), or more technically, components, can be added to the JFrame's `contentPane`:

```
frame.getContentPane().add(button);
```

The JFrame is displayed when it knows the required size by calling `setVisible`

```
frame.setSize(500, 500);
frame.setVisible(true);
```

11.3 Event handling

Reading: pp. 356–362 of *HFJ*.

Your application must listen for a GUI event such as a clicked button. This means that the application registers its interest with an event source (button, check box, etc.). When an event is fired, the event source notifies all the registered listeners. Registration takes the form `add<EventType>Listener` e.g.

```
button.addActionListener(this);
```

Your application must implement the listener interface. The event source will then know which method to call. You place the event-handling code in the listener call-back method. For action events the method is:

```
public void actionPerformed(ActionEvent e){

    button.setText("Clicked!"); // for example
}
```

The event object carries information about the event, including the source of the event.

11.4 A simple layout manager

Reading: pp. 369–372 of *HFJ*.

`BorderLayout` is a simple Swing layout manager. `JPanels` and other components can be added to one of four regions of the `JFrame` using a two parameter `add()` method. The first parameter specifies the region. The single parameter `add()` method defaults to the centre region.

11.5 Action events from more than one source

Reading: pp. 373–381 of *HFJ*.

The problem is that we have (say) two buttons. Each button should do different things on our application. But we can't have two `actionPerformed` methods; after all which one would be called by the event source? You could have a single `actionPerformed` and query the event to find out where it came from. This **will** work but is clumsy and makes the method difficult to write and understand. As a third option, we could write two separate ActionListener classes – but they wouldn't have access to the variables they need to act on.

The accepted solution is to use inner classes. An inner class can access all its containing classes variables and methods, **even the private ones**. An inner class must be tied to an outer class instance. The two live side by side on the heap, both having access to each other's variables and methods.

The *HFJ* example `TwoButton.java` shows how two inner classes provide `actionListeners` for each button of this simple application.

11.6 Summary

- A JFrame represents a window on the screen.

  ```
  JFrame frame = new JFrame();
  ```

- Widgets (buttons, text fields etc.), or more technically, components, are added to the JFrame's content pane:

  ```
  frame.getContentPane().add(button);
  ```

- The JFrame is displayed when it knows the required size is made visible:

  ```
  frame.setSize(500, 500);
  frame.setVisible(true);
  ```

- The application must listen for a GUI event such as a clicked button.
- The application must register its interest in events with the event source (button, check box, etc.) that fires the event. Registration takes the form `add<EventType>Listener` e.g.

  ```
  button.addActionListener(this);
  ```

- Your GUI must implement the listener interface. The event source will then know which method to call on your GUI. The event-handling code is placed in the listener call-back method. For action events the method is:

  ```
  public void actionPerformed(ActionEvent e){

      button.setText("Clicked!");
  }
  ```

- The event object carries information about the event, including the source of the event.

11.7 Programming

Learning activity

The design team have placed a class specification on your desk. They want you to write an abstract class, `GooComponent`, which is to represent a widget on a Goo animation. Concrete subclasses of GooComponent will include `GooButton` and `GooSlider`, which are graphics user interfaces for a toggle switch and a sliding control.

GooComponent should implement Drawable,

```
package goo;

import java.awt.Graphics;

public interface Drawable {

  public abstract void draw(Graphics g);
}
```

A `GooComponent` is situated at *(*x, y) (top left corner) and has a width and a height. A GooComponent should maintain a list of objects of type `GooListener`. Apart from the constructors, the specification lists these methods:

- `addListener(GooListener gl)`. Adds a listener.
- `draw(Graphics g)` method.
- `abstract void fireActionPerformed(GooEvent)`. This method is called by Goo whenever a mouse is moved, dragged, clicked or pressed. Goo passes a reference to a `GooEvent` object which represents the event itself.
- `isInside(in x, int y, int width, int height, int a, int b)` returns true if (a, b) is within the rectangle of dimensions *width x height* situated at *(x, y)*.

A GooListener type is defined by the interface

```
package goo;

public interface GooListener {

    public abstract void gooActionPerformed(GooEvent e);
}
```

Use the class specification to write a class skeleton for `GooComponent`. (A class skeleton contains the important instance and class variables, and instance and class methods, except that the method code blocks are not implemented.)

11.8 Skeleton GooComponent

```
public abstract class GooComponent implements Drawable{

    private ArrayList<GooListener> listeners;
    private x, y, width, height;

    public GooComonent(){
    }

    public GooComponent(int x, int y, int w, int h) {
    }

    public void addListener(GooListener gl){
    }

    public void isInside(int x, int y, int width, int height, int a, int b){
```

```
    }

    public abstract void fireActionEvent(GooEvent e){
    }
}
```

Notice that GooComponent does not define draw, even though it implements Drawable. This is because it is abstract; however, the first concrete subclass in the inheritance tree **must** provide a draw method.

Learning activity

Complete the class by providing implementation.

11.9 GooComponent

```
package goo;

import java.util.ArrayList;

public abstract class GooComponent implements Drawable {

  private ArrayList<GooListener> listeners;

  private int x, y, width, height;

  public GooComponent() {

    this(0, 0, 0, 0);
  }

  public GooComponent(int x, int y, int w, int h) {

    setListeners(new ArrayList<GooListener>());

    this.setX(x);
    this.setY(y);
    setWidth(w);
    setHeight(h);
  }

  public void addListener(GooListener gl) {

    getListeners().add(gl);
  }

  public boolean isInside(int x, int y, int w, int h, int a, int
      b){

    return(a > x && a < x + w && b > y && b < y + h);
  }

  public abstract void fireActionPerformed(GooEvent e);
```

```
  private void setListeners(ArrayList<GooListener> listeners) {
    this.listeners = listeners;
  }

  public ArrayList<GooListener> getListeners() {
    return listeners;
  }

  public void setWidth(int width) {
    this.width = width;
  }

  public int getWidth() {
    return width;
  }

  public void setHeight(int height) {
    this.height = height;
  }

  public int getHeight() {
    return height;
  }

  public void setY(int y) {
    this.y = y;
  }

  public int getY() {
    return y;
  }

  public void setX(int x) {
    this.x = x;
  }

  public int getX() {
    return x;
  }
}
```

The code for addListener is self-evident, except perhaps the `isInside` computation, which can be performed in a single Boolean expression.

Learning activity

Write a GooEvent class to represent a user event. You can assume that the event happens at *(*x, y) and has an associated integer value (for example value = 0 for 'OFF', and value = 1 for 'ON'). A GooEvent should also contain a reference to the triggering component.

11.10 GooEvent

```
package goo;

public class GooEvent {

  private GooComponent source;

  private int x, y, value;

  final static int UNSPECIFIED = -1, ON = 1, OFF = 0;

  public GooEvent(GooComponent c, int x, int y, int val) {

    source = c;
    this.x = x;
    this.y = y;
    value = val;
  }

  public GooComponent getSource() {
    return source;
  }

  public void setSource(GooComponent c) {
    source = c;
  }

  public int getValue() {
    return value;
  }

  public void setValue(int v) {
    value = v;
  }

  public int getX() {
    return x;
  }

  public void setX(int i) {
    x = i;
  }

  public int getY() {
    return y;
  }

  public void setY(int i) {
    y = i;
  }
}
```

The GooEvent class is just a bundle of data from the event, along with getters and setters. The integer variable `value` specifies the current state of the component (e.g. a button may be ON/OFF, a slider has a value between 0 and 100). Some constants

(declared as static and final) have also been defined. `static` means that they are class variables i.e. not associated with any instance of the class, and are accessed as GooEvent.ON, etc. `final` ensures that the values cannot be changed.

Learning activity

Write a concrete subclass of GooComponent, `GooButton`. The button should know how to draw itself, and it should respond to a call to fireActionperformed by passing the GooEvent on to all registered listeners. Consider if any of the data in the GooEvent object needs to be changed by the button.

11.11 GooButton

```
package goo;

import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;

public class GooButton extends GooComponent {

  private boolean isOn = false;
  private Color onColor = new Color(255, 255, 255);
  private Color offColor = new Color(200, 200, 200);
  private Color buttonBackground = Color.white;
  private int margin, pressX, pressY, pressWidth, pressHeight;

  public GooButton() {

    super();
  }

  public GooButton(int x, int y, int w, int h) {

    super(x, y, w, h);
    margin = 2;
    pressX = x + margin;
    pressY = y + margin;
    pressWidth = getWidth() - 2 * margin;
    pressHeight = getHeight() - 2 * margin;
  }

  public void fireActionPerformed(GooEvent e) {

    if (isInside(pressX, pressY, pressWidth, pressHeight - 2 *
        margin, e
         .getX(), e.getY())) {

      isOn = !isOn;

      e.setSource(this);
      e.setValue(isOn ? GooEvent.ON : GooEvent.OFF);

      ArrayList<GooListener> listeners = getListeners();
      for (GooListener l : listeners) {
```

```
        l.gooActionPerformed(e);
      }
    }
  }

  public void draw(Graphics g) {

    Color currColor = g.getColor();

    g.setColor(buttonBackground);
    g.fillRect(getX(), getY(), getWidth(), getHeight());

    g.setColor(isOn ? onColor : offColor);
    g.fillRect(pressX, pressY, pressWidth, pressHeight);
    g.setColor(currColor);
  }
}
```

`fireActionPerformed` only need notify listeners if the event is inside the button. If this is so, the state of the button reverses and the event source is set to the button itself using the `this` variable. The event value is set according to the value of `isOn`. The complicated `booleanExpression ? trueValue : falseValue` syntax is known as the conditional operator. It evaluates to `trueValue` if `booleanExpression` is true, otherwise evaluating to `falseValue`.

Learning activity

Write another concrete subclass of GooComponent, `GooSlider`. A slider consists of a moveable knob which can be dragged across a linear track by the mouse. The slider notifies all listeners of the current value i.e. on the position of the knob along the track, scaled to lie between 0 and 100.

The slider should respond to draw and fireActionPerformed messages; you may want to enhance the functionality of the slider with additional methods.

11.12 GooSlider

```
package goo;

import java.awt.Color;
import java.awt.Graphics;
import java.util.ArrayList;

public class GooSlider extends GooComponent {

  private Color background = Color.WHITE;

  // margin between background and foreground
  private int margin;

  // foreground
  private Color foreground = new Color(200, 200, 200);
  private int foregroundX, foregroundY, foregroundWidth,
      foregroundHeight;
```

```
// knob
private int knobWidth, knobX;
private Color knobColor = Color.BLACK;

// value of slider between 0 and maxValue
private int value, maxValue;

// value 0 at position scaleOrigin
private float scaleOrigin;

// value maxValue at scaleOrigin + scaleMax
private int scaleMax;

// the x coordinate of the input event
private int eventX;

public GooSlider() {

  super();
}

public GooSlider(int x, int y, int w, int h) {

  super(x, y, w, h);

  // parameters
  margin = 2;
  knobWidth = 21;
  maxValue = 100;

  // foreground rectangle
  foregroundX = x + margin;
  foregroundY = y + margin;
  foregroundWidth = getWidth() - 2 * margin;
  foregroundHeight = getHeight() - 2 * margin;

  // set scale
  scaleOrigin = foregroundX + knobWidth / 2.0f;
  scaleMax = foregroundWidth - knobWidth - 1;

  // set initial value
  setValue(Math.round(maxValue / 2.0f));

}

public void setValue(int v) {

  if (0 <= v && v <= maxValue) {

    value = v;
    eventX = posX(v);
    knobX = Math.round(eventX - knobWidth / 2.0f);
  }
}

// returns a value when given a position
private int value(int evX) {

  return Math.round(maxValue * 1.0f * (evX - scaleOrigin) /
```

```
        scaleMax);
  }

  // returns a position when given a value
  private int posX(int val) {

    return Math.round(scaleOrigin + 1.0f * val * scaleMax /
        maxValue);
  }

  public void fireActionPerformed(GooEvent e) {

    int evX = e.getX();
    int evY = e.getY();

    int evValue = value(evX);
    if (evY >= foregroundY && evY <= foregroundY +
        foregroundHeight
        && evValue >= 0 && evValue <= maxValue) {

      eventX = evX;
      knobX = Math.round(eventX - knobWidth / 2.0f);
      value = evValue;

      e.setValue(value);
      e.setSource(this);

      ArrayList<GooListener> listeners = getListeners();
      for (GooListener l : listeners) {

        l.gooActionPerformed(e);
      }
    }
  }

  public void draw(Graphics g) {

    Color currColor = g.getColor();

    g.setColor(background);
    g.fillRect(getX(), getY(), getWidth(), getHeight());

    g.setColor(foreground);
    g.fillRect(foregroundX, foregroundY, foregroundWidth,
        foregroundHeight);

    g.setColor(knobColor);
    g.fillRect(knobX, foregroundY, knobWidth, foregroundHeight);

    g.setColor(background);
    g.drawLine(eventX, foregroundY, eventX, foregroundY +
        foregroundHeight);

    g.setColor(currColor);
  }
}
```

Learning activity

We wish to test our event components with an application. We decide to work with the GooDrops animations from the `referencetypes` package.

Write a subclass of `referencetypes.Colourdrop`, `ControlDrop`, with an overridden `move()`. move should allow the drop to fall at any speed between 0 and xVel, according to the value of a float instance variable.

11.13 Controlled Goo Drop

```
package simpleevents;

import java.awt.Color;

import referencetypes.ColourDrop;

public class ControlDrop extends ColourDrop {

  private float modifier = 1.0f;

  public ControlDrop(int x, int y, int vx, int vy, int sz, Color
      c) {
    super(x, y, vx, vy, sz, c);
  }

  public void move(int width, int height){

    xpos = Math.round(xpos + getModifier() * xvel);
    ypos = Math.round(ypos + getModifier() * yvel);

    if (ypos > height) {
      ypos = 0;
      xpos = (int)(Math.random() * width);
    }
  }

  public void setModifier(float modifier) {
    this.modifier = modifier;
  }

  public float getModifier() {
    return modifier;
  }
}
```

Learning activity

You should now subclass GooDropsInColour, overriding makeDrop so that the drops array is populated with ControlDrops, and include a new method public void slow(float s) which sets `modifier` in each `ControlDrop` to s. Also override draw so that the drops do not move at all if a Boolean instance variable, stop, is set to true.

11.14 Goo Drops with Controls

```
package simpleevents;

import java.awt.Color;
import java.awt.Graphics;
import java.util.Random;

import referencetypes.GooDropsInColour;
import referencetypes.Drop;

public class ControlledGooDrops extends GooDropsInColour {

  private boolean stop = false;

  public ControlledGooDrops(int w, int h, int nd) {
    super(w, h, nd);
  }

  public Drop makeDrop(int xpos, int ypos,
                       int xvel, int yvel, int size) {

    Random random = getRandom();
    Color color = new Color(random.nextInt(256), random.nextInt
        (256),
        random.nextInt(256));
    return new ControlDrop(xpos, ypos, xvel, yvel, size, color);
  }

  public void slow(float s) {

    Drop[] drops = getDrops();
    for (Drop drop : drops) {
      ((ControlDrop) drop).setModifier(s);
    }
  }

  public void draw(Graphics g) {

    if (!stop)
      super.draw(g);
    else {
      Drop[] drops = getDrops();
      for (Drop d : drops) {
        d.draw(g);
      }
    }
  }

  public void setStop(boolean stop) {
    this.stop = stop;
  }

  public boolean getStop() {
    return stop;
  }
}
```

Learning activity

Finally, write an application to test ControlledGooDrops. Add a GooButton and a GooSlider so that the user can control the speed of the drops, and can stop and start the animation.

11.15 Controlled GooDrops App

```
package simpleevents;

import goo.GooButton;
import goo.GooEvent;
import goo.GooListener;
import goo.GooSlider;

public class ControlledGooDropsApp {

  ControlledGooDrops gooDrops;

  class SliderListener implements GooListener{

    public void gooActionPerformed(GooEvent e) {

      gooDrops.slow(e.getValue() / 100.0f);
    }
  }

  class ButtonListener implements GooListener{

    public void gooActionPerformed(GooEvent e) {

      gooDrops.setStop(!gooDrops.getStop());
    }
  }

  public ControlledGooDropsApp() {

    int width = 800;
    int height = 500;
    int numDrops = 200;

    gooDrops = new ControlledGooDrops(width, height, numDrops);

    GooButton button = new GooButton(10, 20, 50, 50);
    gooDrops.addComponent(button);
    button.addListener(new ButtonListener());

    GooSlider slider = new GooSlider(10, 90, 200, 50);
    gooDrops.addComponent(slider);
    slider.addListener(new SliderListener());
    slider.setValue(100);

    gooDrops.background(0); // black background
    gooDrops.frameRate(25); // 25 frames per sec
    gooDrops.smooth();
    gooDrops.go();
```

```
  }

  public static void main(String[] args) {

    new ControlledGooDropsApp();
  }
}
```

11.16 Learning outcomes

By the end of this chapter, the relevant reading and activities, you should be able to:

- explain how a `JFrame` represents a window on the screen
- describe how widgets or components are added to the `JFrame`'s content pane
- explain how the application must listen for a GUI event such as a clicked button
- explain how the application registers its interest in events, and the form of registration
- describe how the GUI must implement the listener interface
- explain that event object carries information about the event, including the source of the event.

Chapter 12

Graphics

Essential reading

HFJ Chapter 12.

12.1 Introduction

Reading: pp. 363–368 of *HFJ*.

Swing enables the programmer to draw 2D graphics and a picture – such as .gif or .jpg – directly to a widget. The Swing application must subclass `JPanel` and override `paintComponent()` which is called by the JVM whenever the window needs refreshing. **Your code must never call this method.**

The argument to paintComponent() is a Graphics object that represents a drawing surface. Only the JVM can construct this object. You draw on the surface by calling methods on the Graphics object, e.g.

```
g.setColor(Color.BLUE);
g.fillRect(10, 10, 100, 200);
```

An image is drawn from an Image object:

```
Image image = new ImageIcon("cat.jpg");
g.drawImage(image, 10,20, this);
```

The object referenced by the Graphics parameter is actually an instance of Graphics2D. Graphics2D methods can be invoked after casting the object reference to Graphics2D:

```
Graphics2d g2d = (Graphics2D) g;
```

12.2 Animations

Reading: pp. 382–385 of *HFJ*.

A succession of images can give the appearance of motion if the images are displayed many times per second and if each image differs slightly from the previous one. For example, an oval might be drawn at (x, y) and then redrawn at (x+1, y+1) a fraction of a second later. The oval will appear to move in a diagonal line across the screen.

However each image must be erased before the next one is drawn. Simply filling the window with a single colour will erase all images from the previous 'frame'.

We can draw 20 frames per second if we can ask the JVM to halt execution for $\frac{1}{20}$ second (50 ms) between frames. This is accomplished by putting the current 'thread' to sleep:

```
try{
    Thread.sleep(50);
}
catch(Exception e){}
```

The call to `Thread.sleep` is made inside a try/catch block. Every JVM manages one or more 'threads' or streams of execution. Every thread has a separate stack. The try/catch block allows the programmer to specify what happens if something goes wrong. We shall study threads and try/catches in Part 2 of this course (see Volume 2 of the study guide).

12.3 Summary

- You can draw 2D graphics directly on a widget.
- You can draw a .gif or a .jpeg directly on a widget.
- To draw graphics, subclass [Panel and override `paintComponent()`
- `paintComponent()` is called by the JVM whenever the window needs refreshing. Your code must never call this method.
- The argument to paintComponent() is a Graphics object that represents a drawing surface. Only the JVM can construct this object.
- You draw on the surface by calling methods on the Graphics object, e.g.

  ```
  g.setColor(Color.BLUE);
  g.fillRect(10, 10, 100, 200);
  ```

- An image is drawn from an Image object:

  ```
  Image image = new ImageIcon("cat.jpg");
  g.drawImage(image, 10,20, this);
  ```

- The object referenced by the Graphics parameter is actually an instance of Graphics2D.
- Graphics2D methods can be invoked after casting the object reference to Graphics2D:

  ```
  Graphics2d g2d = (Graphics2D) g;
  ```

- An animation is a succession of images, each one redrawn at a different place.
- The current execution thread can be halted with a call to `Thread.sleep(timeMillis)` where timeMillis is the required pause.
- The call to sleep must be put inside a try/catch block. The purpose of this block is to provide emergency code if the call fails.

12.4 Programming

Learning activity

Write a Java graphics program to display a green disc with a diameter of 40 pixels in the middle of a brown background. The disc must be redrawn at the centre of the window whenever the window is resized.

12.5 GooPanel

```
package simplegraphics;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class GPanel extends JPanel {

  private static final long serialVersionUID = 1L;
  private int diameter = 40;

  public void paintComponent(Graphics g) {

    int width = this.getWidth();
    int height = this.getHeight();

    g.setColor(Color.WHITE);
    g.fillRect(0, 0, width, height);

    int x = (width - diameter) / 2;
    int y = (height - diameter) / 2;

    g.setColor(Color.GREEN); // J
    g.fillOval(x, y, diameter, diameter);
  }

  public static void main(String[] args) {

    GPanel myPanel = new GPanel();

    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(myPanel);
    frame.setSize(300, 300);
    frame.setVisible(true);
  }
}
```

Let us now follow the program in the order of execution, starting with the first statement in `main`.

A `GPanel` object is instantiated. GPanel subclasses JFrame and overrides paintComponent.

A `JFrame` object is created. A `JFrame` represents an actual window on the screen. The JVM will use the `JFrame` object that you have created to set up and draw the window. In fact you should think of the JFrame as the edge (frame) of the window. The window – imagine a painting – is completed by placing a frame around a **panel**. Your draw on the panel; this is clipped into a frame, and displayed.

`setDefaultCloseOperation` ensures that the program terminates when the user closes the window.

The panel is added to the frame. This is an unusual line of code, because it contains two method calls. The compiler works from left to right, so this line is equivalent to `(frame.getContentPane()).add(myPanel);` The call `getContentPane()` asks `frame` for a pane. The pane is a blank space inside the frame. The code asks the `ContentPane` object to add our panel to the frame's pane.

The window's initial dimensions are set to 300 x 300 pixels.

The window is displayed by sending a `setVisible` message to the frame and then main terminates. Normally the program would terminate at this point because main leaves the stack and nothing replaces it. However the Java graphics system is running a separate graphics thread. Your GPanel object is referenced by a variable in this thread and paintComponent is called on your object whenever the window needs refreshing or redrawing.

Notice that the width and height of the JPanel are not stored as instance variables; paintComponent asks its superclass for the current width and height. This means that the window can be resized by the user and the oval is still drawn in the centre.

Learning activity

Refactor (i.e. rework) GPanel so that paintComponent becomes a method of an inner class `GooPanel extends JPanel`. Add an instance method `go()` which causes the frame to be displayed.

12.6 Drawing

```
package simplegraphics;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

class Drawing {

  class GooPanel extends JPanel {

    private static final long serialVersionUID = 1L;

    public void paintComponent(Graphics g) {
```

```
      int width = this.getWidth();
      int height = this.getHeight();

      g.setColor(Color.WHITE);
      g.fillRect(0, 0, width, height);

      int x = (width - diameter) / 2;
      int y = (height - diameter) / 2;

      g.setColor(Color.GREEN); // J
      g.fillOval(x, y, diameter, diameter);
    }
  }

  private int diameter = 40;
  private JFrame frame;
  private GooPanel gooPanel;

  public Drawing() {

    gooPanel = new GooPanel();

    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(gooPanel);
    frame.setSize(300, 300);
  }

  void go() {

    frame.setVisible(true);
  }

  public static void main(String[] args) {

    Drawing drawing = new Drawing();
    drawing.go();
  }
}
```

The code for `GooPanel extends JPanel` has been placed as an inner class within Drawing. An inner class is just like any (top) level class, except that it is intimately tied to its outer class. This means that an inner class object can use its outer class variables (and, in fact, vice versa). The Jframe and GooPanel have been declared as instance variables, rather than local variables within a method. Other solutions are possible; for example all the statements form the constructor could be placed in go() and frame could then be defined locally.

Learning activity

Take `Drawing` and modify the code so that the desired drawing is specified by a concrete subclass. In order to do this, include a single abstract method `public abstract void draw(Graphics g)`. Remove the code that draws the green oval and replace by a call to draw. Include getters for the frame width and height.

12.7 GooDrawing

```
package simplegraphics;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

abstract class GooDrawing {

  private JFrame frame;
  private GooPanel gooPanel;

  class GooPanel extends JPanel {

    private static final long serialVersionUID = 1L;

    public void paintComponent(Graphics g) {

      g.setColor(Color.WHITE);
      g.fillRect(0, 0, this.getWidth(), this.getHeight());

      draw(g);
    }
  }

  public GooDrawing(int w, int h) {

    gooPanel = new GooPanel();

    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(gooPanel);
    frame.setSize(w, h);
  }

  public abstract void draw(Graphics g);

  public int getWidth() {

    return gooPanel.getWidth();
  }

  public int getHeight() {

    return gooPanel.getHeight();
  }

  void go() {

    frame.setVisible(true);
  }
}
```

Learning activity

Write a concrete subclass of GooDrawing to demonstrate how programmers might implement simple drawings without worrying about frames and panels.

12.8 GooDrawingApp

```
package simplegraphics;

import java.awt.Color;
import java.awt.Graphics;

public class GooDrawingApp extends GooDrawing {

  private int diameter = 40;

    public GooDrawingApp(int w, int h) {
    super(w, h);
  }

  public void draw(Graphics g) {

    // place your drawing instructions here

    int x = (getWidth() - diameter) / 2;
    int y = (getHeight() - diameter) / 2;

    g.setColor(Color.GREEN); // J
    g.fillOval(x, y, diameter, diameter);
  }

  public static void main(String[] args){

    GooDrawing app = new GooDrawingApp(300, 300);
    app.go();
  }
}
```

By taking the common elements of all drawings and placing them in a superclass, any drawing can be accomplished by merely subclassing and implementing draw (Graphics g).

Learning activity

HFJ pg. 384 shows a simple animation class. Take this class and apply the same procedure that led from Drawing to GooDrawing and GooDrawingApp. You should end up with an abstract class SimpleGoo and an application SimpleGooApp which runs the same animation as *HFJ*'s SimpleAnimation.

12.9 Simple Animation

```
package simplegraphics;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class SimpleAnimation {

  int x = 70;
  int y = 70;

  JFrame frame;
  GooPanel gooPanel;

  class GooPanel extends JPanel {

    private static final long serialVersionUID = 1L;

    public void paintComponent(Graphics g) {

      g.setColor(Color.white);
      g.fillRect(0, 0, this.getWidth(), this.getHeight());
      g.setColor(Color.green);
      g.fillOval(x, y, 40, 40);
    }
  }

  public SimpleAnimation() {

    gooPanel = new GooPanel();

    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(gooPanel);
    frame.setSize(300, 300);
  }

  public void go() {

    frame.setVisible(true);

    for (int i = 0; i < 130; i++) {

      x++;
      y++;

      gooPanel.repaint();

      try {
        Thread.sleep(50);
      } catch (InterruptedException e) {
      }
    }
```

```
  }

  public static void main(String[] args) {

    SimpleAnimation sa = new SimpleAnimation();
    sa.go();
  }
}
```

Our SimpleAnimation differs slightly from HFJ's Simple Animation since some of the code from go has been moved to the constructor and `MyDrawPanel` has been renamed `GooPanel`.

12.10 SimpleGoo

```
package simplegraphics;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public abstract class SimpleGoo {

  private JFrame frame;
  private GooPanel gooPanel;

  private boolean loop = false;

  class GooPanel extends JPanel {

    private static final long serialVersionUID = 1L;

    public void paintComponent(Graphics g) {

      g.setColor(Color.white);
      g.fillRect(0, 0, this.getWidth(), this.getHeight());
      draw(g);
    }
  }

  public SimpleGoo(int w, int h) {

    gooPanel = new GooPanel();

    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(gooPanel);
    frame.setSize(w, h);
  }

  public abstract void draw(Graphics g);

  public void go() {

    frame.setVisible(true);
    loop = true;
```

```
    while (loop) {

      gooPanel.repaint();

      try {
        Thread.sleep(50);
      } catch (InterruptedException e) {
      }
    }
  }
}
```

We arrive at SimpleGoo by removing the animation code and adding an abstract draw method. The intention is that subclasses of SimpleGoo provide their own animation code, thereby hiding the GUI and loop details.

12.11 SimpleGooApp

```
package simplegraphics;

import java.awt.Color;
import java.awt.Graphics;

public class SimpleGooApp extends SimpleGoo {

  int x = 70, y = 70;
  int numFrames = 0;

  public SimpleGooApp(int w, int h) {
    super(w, h);
  }

  public void draw(Graphics g) {

    // write your animation here

    if (numFrames < 130) {
      x++;
      y++;
    }

    g.setColor(Color.green);
    g.fillOval(x, y, 40, 40);

    numFrames++;
  }

  public static void main(String[] args) {

    SimpleGooApp sg = new SimpleGooApp(300, 300);
    sg.go();
  }
}
```

The application subclasses `Goo` and provides an implementation of `draw`. Since the original animation runs for 130 frames, a counter, `numFrames`, has been introduced to ensure that the disc stops moving.

Learning activity

SimpleGoo enables us to program an animation without worrying too much about GUI details. We just take the template which follows, and write our animation in `draw`.

```
package mypackage;

import java.awt.Color;
import java.awt.Graphics;

public class MyGooApp extends SimpleGoo {

    public SimpleGooApp(int w, int h) {
        super(w, h);
    }

    public void draw(Graphics g) {

        // write your animation here
    }

    public static void main(String[] args) {

        SimpleGooApp sg = new SimpleGooApp(300, 300);
        sg.go();
    }
}
```

Of course we may also define any other instance variables and methods that we may need.

However `SimpleGoo` is limited: we cannot choose a different background colour, or alter the animation speed. We would also like to make use of anti-aliasing by calling a `smooth` on `SimpleGoo`. Other refinements might include a `noloop` method which cancels the animation loop and converts `SimpleGoo` into a drawing, and creates getters for the frame height and width.

Learning activity

Implement the refinements suggested above.

You will need to refer to Graphics2D.setRenderingHints() in the API for information on how to use Java's anti-aliasing capability.

12.12 Refined SimpleGoo

```
package simplegraphics;

import java.awt.Color;
```

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;

import javax.swing.JFrame;
import javax.swing.JPanel;

public abstract class SimpleGoo1 {

  private JFrame frame;
  private GooPanel gooPanel;

  private boolean loop = true;
  private boolean smooth = false;

  private int width, height;

  private Color backgroundColor = Color.white;

  private int frameTime = 50;

  private RenderingHints renderingHints = new RenderingHints(
      RenderingHints.KEY_ANTIALIASING, RenderingHints.
          VALUE_ANTIALIAS_ON);

  class GooPanel extends JPanel {

    private static final long serialVersionUID = 1L;

    public void paintComponent(Graphics g) {

      Graphics2D g2d = (Graphics2D) g;
      if (smooth)
        g2d.setRenderingHints(renderingHints);
      g2d.setColor(backgroundColor);
      g2d.fillRect(0, 0, this.getWidth(), this.getHeight());
      draw(g2d);
    }
  }

  public SimpleGoo1() {
    this(300, 300);
  }

  public SimpleGoo1(int w, int h) {

    width = w;
    height = h;

    gooPanel = new GooPanel();

    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(gooPanel);
    frame.setSize(width, height);
  }

  public abstract void draw(Graphics g);
```

```
  public void smooth() {

    smooth = true;
  }

  public void noSmooth() {

    smooth = false;
  }

  public void frameRate(double framesPerSec) {

    frameTime = (int) (1000 / framesPerSec);
  }

  public void background(int gscale) {

    background(gscale, gscale, gscale);
  }

  public void background(int red, int green, int blue) {

    backgroundColor = new Color(red, green, blue);
  }

  public int getWidth() {

    return width;
  }

  public int getHeight() {

    return height;
  }

  public void noloop() {

    loop = false;
  }

  public void loop() {

    loop = true;
  }

  public void go() {

    frame.setVisible(true);
    while (loop) {

      gooPanel.repaint();

      try {
        Thread.sleep(frameTime);
      } catch (InterruptedException e) {
      }
    }
  }
}
```

You have effectively built Goo, the animation package that you have been using since you programmed your first GooDrop in Chapter 2. Take a look at the code for Goo in the goo package. Much of it should look familiar.

12.13 Part I Summary

Congratulations, you have finished Part I. You should now be able to:

- program drawings and animations
- add homemade buttons and sliders to your graphics applications
- use Swing's own components
- understand polymorphism, listeners, interfaces, getters and setters ...

You have come a long way.

See you in Part II.

Goo-d Bye!!

12.14 Learning outcomes

By then of this chapter, the relevant reading and activities, you should be able to:

- draw 2D graphics, a .gif or a .jpeg on a widget by subclassing Panel and overriding paintComponent()
- understand how the JVM refreshes the window
- explain what the argument to paintComponent() is and know that only the JVM can construct this object
- cast the Graphics object to a Graphics2D object in order to invoke the methods from the Graphics2D library
- understand how a computer constructs an animation
- program simple animations by using Thread.sleep to halt the program for a given time interval between frames.

Chapter 13

Revision

13.1 Overview

13.1.1 Java without objects

Summary

- Source code, compiled code and the JVM
- Statements end with a semicolon (;)
- Code blocks are defined by a pair of curly braces
- Assignment operator =
- Equals operator ==
- A while loop runs through its block as long as the conditional test is true
- If the conditional test is false, the while loop code block won't run and execution passes to the code immediately after the loop block
- Conditional branching: `if` and `if/else`
- String arrays.

Programs

- Eliza

13.1.2 Objects

Summary

- Class boxes
- Object programming lets you extend a program without having to touch previously-tested code.
- All Java code is defined in a class.
- A class describes how to make an object of that class time. A class is like a blueprint.
- An object knows about things and does things.
- Things an object knows are called instance variables. They represent the state of that object.
- Things an object does are called methods. They represent the behaviour of an object.

- When you create a class, you may also wish to create a separate test class which you'll use to create objects of your new class type.
- `main` can be used as a launcher for your application, and as a class tester.
- A class can inherit instance variables and methods from a more abstract superclass.
- At runtime a Java program is nothing more than objects 'talking' to other objects.
- The garbage-collectible heap and the garbage collector.

Programs

- Drop
- GooDrop
- GooDrop2
- GooDropApp
- RedDrop
- SimpleDrop
- WobblyDrop

13.1.3 Reference types

Summary

- There are two flavours of variables: primitive and reference.
- Variables must always be declared with a name and a type.
- The value of a primitive variable is the value of the bits representing the value (e.g. 5,, 'a', true, 3.1416, etc.)
- A reference variable is like a remote control. The dot operator (.) is like pressing a button on the remote control to access a method or instance variable.
- A reference variable has the value `null` when it is not referencing an object.
- An array is always an object, even if the array is declared to hold primitives. There is no such thing as a primitive array, only an array that holds primitives.
- Memory diagrams illustrate object life on the heap.
- References may be active or null; objects may be reachable or abandoned.

Programs

- ColourDrop
- Drop
- GooDrops
- GooDropsInColour

13.1.4 Object behaviour

Summary

- Classes define what an object knows and what it does.
- Things an object knows about are its instance variables (State).
- Things that an object does are its instance methods (behaviour).
- A method can have parameters, which means that you can pass one or more values in to the method.
- The number and type of values you pass in must match the order and type of the parameters declared by the method.
- Java uses pass-by-value.
- Values passed in and out of the methods can be implicitly promoted to a larger type, or explicitly cast to a smaller type.
- You can pass a literal (e.g. 5, 'a', true, 3.1416, etc.) or a variable of the declared parameter type (e.g. `x` where `x` has been declared as an `int` variable.) This is not quite the whole story.
- A method must declare a return type. A `void` return type means that the method does not return anything.
- If a method declares a non-void return type, it must return a value compatible with the return type.
- According to the principle of encapsulation, instance variables should be declared private, and setters/getters methods declared public.
- Instance variables are automatically initialised by the compiler; but local variables must be initialised explicitly.
- Variables can be compared using the equals operator `==` and by calling `equals()`.

Programs

- Hoop
- HoopApp
- HoopWithEquals
- MovingHoop
- MovingHoopApp
- SolidEllipse
- SolidEllipseApp

13.1.5 Program development

Summary

- Start with a high level design.
- Write pseudocode and then test code
- Then write the actual code.
- `for` loops are preferred when you know how many loops will be performed.

- Post- and pre-decrement/increment operators.
- `Integer.parseInt()` converts, if possible, a String into an integer.
- `break` forces an exit from a loop.
- The enhanced `for` loop.
- Casting primitives.

Programs

- GooByStarlight
- GooMoon
- GooStar
- Moon
- Sky
- Star
- StarlightApp

13.1.6 The Java library

Summary

- The `ArrayList` is a class in the Java API.
- You can consult the API by looking at the javadocs, or a summary such as *Java in a nutshell*.
- Useful `ArrayList` methods are `add(), remove, indexOf(), isEmpty(), contains()`. The number of elements in an array is available as the `length` variable; the length of an `ArrayList` is returned by the `size()` method.
- An `ArrayList` resizes to whatever size is appropriate.
- `ArrayLists` can be paramaterised using a type parameter in angle brackets ¡ ¿.
- An `ArrayList` can only hold objects; primitives are automatically wrapped into objects, and unwrapped back to primitives on insertion and retrieval.
- Classes are grouped into packages.
- The full name of a class is of the form *packagename.classname*.
- A class from the API must either be imported, or specified by its full name (with the exception of `java.lang` which contains the most fundamental class of the API).
- `&&` and `||` are short-circuit operators.
- `&` and `|` are not short-circuit operators and are typically used for manipulating bits.
- Use `expressionA != expressionB` or `!(expressionA = expressionB)` to test if two expressions do not evaluate to the same result.
- You can either learn the order of precedence or use parentheses to specify the order of evaluation of long Boolean expressions.

Programs

- LinesAndPoints
- Point
- MouseAndKey

13.1.7 Inheritance

Summary

- A subclass extends a superclass.
- A subclass inherits all public members of the superclass.
- Inherited methods can be overridden.
- Instance variables cannot be overridden, although they can be redefined in a subclass (not recommended).
- Use the IS-A test to verify that inheritance is valid.
- The lowest overridden method is invoked at runtime (dynamic binding).
- Overridden methods must have the same arguments, return types must be compatible and the method cannot be less accessible.
- Overloaded methods must have a different argument list.

Programs

- Shape
- Polygon
- CurvyShape
- MovingPolygon
- MovingCurvyShape

13.1.8 Abstraction

Summary

- An abstract class cannot be instantiated.
- An abstract class can have abstract and non-abstract methods.
- If a class has one or more abstract methods, it must be declared abstract.
- An abstract method has no body, and the declaration ends in a semicolon.
- All abstract methods must be implemented in the first concrete class in the inheritance tree.
- All classes are subclasses of `Object`.
- Methods can be declared with Object arguments and/or return types.
- Method calls can only be made if the methods are in the class (or interface) of the reference variable type, regardless of the object type.

- A reference variable can be cast to a subtype in order to make a call to a subtype method, but at runtime the cast will fail if the object on the heap is not compatible with the cast. E.g.

```
Animal pet  = new Cat();
((Cat)pet).miaow();
```

- Java only supports single inheritance.
- Languages with multiple inheritance have special rules to disambiguate method calls. This makes programming more difficult.
- An interface defines only abstract, public methods.
- An interface definition uses the keyword `interface` in place of `class`.
- A class may implement one or more interfaces by using the keyword `implements`.
- Interface implementation means that the class must provide method bodies for all the interface methods.
- Use `super` to invoke a superclass method of an overridden method.

Programs

- Drawable
- DrawableMoveable
- Message
- Moveable
- MovingPolygon
- Polygon
- Shape

13.1.9 Object lifetime

Summary

- The JVM maintains two important areas of memory, the stack and the heap.
- Instance variables are declared inside a class but outside any method.
- Local variables are declared inside a method or method parameter.
- All local (primitive or reference type) variables live on the stack, in the frame corresponding to the method where the variables are declared.
- All objects live on the heap, regardless of whether the reference is a local variable or an instance variable.
- Instance variables live within their object on the heap.
- If the instance variable is a reference to an object, both the reference and the object it refers to are on the heap.
- A constructor has the same name as the class, but has no return type.
- The constructor can be used to initialise the state of the object.
- The compiler will supply a default constructor if you don't define any constructor.

- But if you do define any constructor, the compiler will not build the default constructor.
- It is good practice to provide a default constructor and default values for the instance variables.
- Constructors may be overloaded and the normal rules for overloaded method argument lists apply.
- Instance variables are initialised to default values 0/0.0/false/null

13.1.10 Events

Summary

- A JFrame represents a window on the screen.

  ```
  JFrame frame = new JFrame();
  ```

- Widgets (buttons, text fields etc.), or more technically, components, are added to the `JFrames`'s content pane:

  ```
  frame.getContentPane().add(button);
  ```

- The JFrame is displayed when it knows the required size is made visible:

  ```
  frame.setSize(500, 500);
  frame.setVisible(true);
  ```

- The application must listen for a GUI event such as a clicked button.
- The application must register its interest in events with the event source (button, check box, etc.) that fires the event. Registration takes the form `add<EventType>Listener` e.g.

  ```
  button.addActionListener(this);
  ```

- Your GUI must implement the listener interface. The event source will then know which method to call on your GUI. The event-handling code is placed in the listener call-back method. For action events the method is

  ```
  public void actionPerformed(ActionEvent e){

      button.setText("Clicked!");
  }
  ```

- The event object carries information about the event, including the source of the event.

Programs

- ControlDrop
- ControlledGooDrops
- ControlledGooDropsApp

13.1.11 Graphics

Summary

- You can draw 2D graphics directly on a widget.
- You can draw a .gif or a .jpeg directly on a widget.
- To draw graphics, subclass `Panel` and override `paintComponent()`
- `paintComponent()` is called by the JVM whenever the window needs refreshing. Your code must never call this method.
- The argument to paintComponent() is a Graphics object that represents a drawing surface. Only the JVM can construct this object.
- You draw on the surface by calling methods on the Graphics object, e.g.

```
g.setColor(Color.BLUE);
g.fillRect(10, 10, 100, 200);
```

- An image is drawn from an Image object:

```
Image image = new ImageIcon("cat.jpg");
g.drawImage(image, 10,20, this);
```

- The object referenced by the Graphics parameter is actually an instance of Graphics2D.
- Graphics2D methods can be invoked after casting the object reference to Graphics2D:

```
Graphics2d g2d = (Graphics2D) g;
```

Programs

- Drawing
- GooDrawing
- GooDrawingApp
- GPanel
- SimpleAnimation
- SimpleGoo
- SimpleGoo1
- SimpleGooApp

13.2 Sample examination questions, answers and appendices

Important note. The information and advice given in the following section are based on the examination structure used at the time this guide was written. Please note that subject guides may be used for several years. Because of this we strongly advise you to always check the current Regulations for relevant information about the examination. You should also carefully check the rubric/instructions on the paper you actually sit and follow those instructions.

Using the sample examination material. The examination is in two parts: Part A Graphical programming and Part B Internet programming. These correspond to Part 1 and Part 2 of this course. You will have to answer two questions from Part A and two questions from part B of the examination.

Attached are the four Part A papers and the Appendices from recent exams. The Appendices (found at the end of the examination paper) contain materials needed for the questions such as lengthy class definitions, and some summaries from the Java API. The order is as follows: Papers 1 and 2 Part A, Appendices for 1 and 2, Answers to 1 and 2, Papers 3 and 4 Part A, Appendix for 3 and 4, Answers to 3 and 4.

Paper 1 Part A

QUESTION 1

(a) Write a Java class `SimpleGui`. This class should have an instance method `public void go()` and a static `main` method. You do not need to provide a constructor for this class. The main method should instantiate a `SimpleGui` object and call `go()` on this object. However the `go()` method should be left blank.

[5 Marks]

(b) Add code to `SimpleGui.java` so that the result of calling `go()` is to create and display a window of size 300 x 300. The programme should terminate when a user closes this window.

[5 Marks]

(c) Explain Java's event handling mechanism. You should consider the relationship between the event source (e.g. a button) and an object that should take some action as a result of this event occurring.

[10 Marks]

(d) Add a `JButton` to the window created in `SimpleGui`. This button should display the current date and time when a user clicks it.

[5 Marks]

Paper 1 Part A

QUESTION 2

(a) Inheritance is a key idea in object oriented programming. In a few sentences, explain the essence of this idea.

[5 Marks]

(b) Under what circumstances should you use inheritance in your own coding? When should you guard against unnecessary use of inheritance?

[5 Marks]

(c) In general terms, what are the advantages of inheritance?

[5 Marks]

(d) Appendix 1 shows a number of classes of a Jungle simulation. The class below, `JungleTestRun` has been written to test the simulation. What is the output of `JungleTestRun.main()`?

```
public class JungleTestRun {

    public static void main(String[] args) {

        CreepyCrawly bug = new GiantMillipede();
        bug.scare();
    }
}
```

[5 Marks]

(e) Which (if any) of the following lines will compile if they are added to `JungleTestRun.main()`? Explain your answer.

```
Cat c = new Cat();
Cat d = new Lion();
```

[5 Marks]

Paper 1 Part A

QUESTION 3

(a) The Java Virtual Machine (JVM) organises memory into two parts; the stack and the heap. What is the stack and how is it used by the JVM?

[5 Marks]

(b) What is the heap and how is it used by the JVM?

[5 Marks]

(c) Draw a stack-heap diagram to illustrate the relationships between variables and objects for the code below:

```
class DogPound{

   public static void main(String[] args){

      Dog dog1 = new Dog();
      Dog dog2 = new Dog();
      ...
   }
}
```

[5 Marks]

(d) Draw a stack-heap diagram to illustrate the relationships between variables and objects just after line 3 for the code below. What will happen to the object referenced by `dog1` in line 1?

```
class DogPound{

   public static void main(String[] args){

      Dog dog1 = new Dog(); // line 1
      Dog dog2 = new Dog(); // line 2
      dog1 = dog2; // line 3
      ...
   }
}
```

[5 Marks]

(e) An object is eligible for garbage collection when its last live reference disappears. Demonstrate with code excerpts how this might happen.

[5 Marks]

Paper 2 Part A

QUESTION 1

(a) Appendix 1 shows a number of classes of a Jungle simulation. The class below, `JungleTestRun`, has been written to test the simulation. What is the output of `JungleTestRun.main()`?

[8 Marks]

```
import java.util.ArrayList;

public class JungleTestRun {

   public static void main(String[] args) {

      ArrayList<Scary> list = new ArrayList<Scary>();
      list.add(new Lion());
      list.add(new GiantMillipede());
      for (Scary s : list)
         s.scare();
   }
}
```

(b) Which (if any) of the following lines will compile if they are added to `JungleTestRun.main()`? Explain your answer.

```
Animal a = new Cat();
Animal a = new Animal();
```

[4 Marks]

(c) `Animal, Cat and CreepyCrawly` have been declared as `abstract` classes in the Jungle simulation. Why is this?

[3 Marks]

(d) The Jungle simulation defines an Interface `Scary`. What is the purpose of interfaces in the Java language? You should illustrate your answer by referring to the Jungle simulation.

[10 Marks]

Paper 2 Part A

QUESTION 2

(a) The code below outlines the class `SimpleAnimation`.

```
import java.awt.Graphics;

public class SimpleAnimation{

    Graphics graphics;
    MyDrawPanel panel;

    int x, y;

    public static void main(String[] args) {
        new SimpleAnimation().setup();
    }

    public void setup() {

    }

    public void draw() {

    }
}
```

Add code to `setup()` so that the result of calling `setup()` on a SimpleAnimation object is to create and display a `JFrame` of size 300 x 300. The programme should terminate when a user closes this window.

[5 Marks]

(b) Add an inner class `MyDrawPanel extends JPanel` to `SimpleAnimation`. This class should override `public void paintComponent(Graphics g)`. `paintComponent` should assign `g` to the instance variable `graphics`, and call `draw`.

[5 Marks]

(c) Include two lines of code in `setup()` to instantiate a `MyDrawPanel` object and add it to your `JFrame`.

[5 Marks]

Paper 2 Part A

(d) Add code to `draw` so that the result of calling this method is to increment x and y by one pixel, clear the panel by painting it white, and then to draw a filled green circle at coordinates (x, y). (Your attention is drawn to the class summaries given in Appendix 2 of this paper)

[5 Marks]

(e) Include a block of code in `setup()` that will repaint the panel at a rate of 20 frames per second, and hence produce an animation of a slowly moving ball.

[5 Marks]

Paper 2 Part A

QUESTION 3

(a) Explain where instance variables, static variables and local variables are declared in a Java programme.

[3 Marks]

(b) Variables are considered to be alive during programme execution for as long as they can potentially be used. With reference to their containing entities, how long do instance, static and local variables live for?

[3 Marks]

(c) The Java Virtual Machine (JVM) organises memory into the stack and the heap. Explain, with reference to the stack and the heap, where instance variables, local variables and objects are organised by the JVM.

[3 Marks]

(d) Explain how methods are stacked by the JVM.

[5 Marks]

(e) What is the difference between life and scope?

[2 Marks]

(f) What is the relationship between the lifetime of an object and the lifetime of the variables that reference it?

[4 Marks]

(g) Write a Java class to demonstrate the creation and life-time of an object. Comment your code to show when the object is created and when it effectively dies.

[5 Marks]

Appendices to Paper 1 and Paper 2

Appendix 1: Jungle Simulation

```
public abstract class Animal {

    int numberOfLegs = 4;

    public Animal(int l){
        numberOfLegs = l;
        System.out.println("In Animal.Animal()");
    }

    public int getNumberOfLegs() {
        return numberOfLegs;
    }
}


public abstract class CreepyCrawly extends Animal implements Scary {

    public CreepyCrawly(int l) {
        super(l);
        System.out.println("In CreepyCrawly.CreepyCrawly()");
    }

    public void scare(){
        System.out.println("Crawling up your leg!!");
        crawl();
    }

    public abstract void crawl();
}


public class GiantMillipede extends CreepyCrawly{

    public GiantMillipede(){
        this(1000);
        System.out.println("In GiantMillipede.GiantMillipede()");
    }
    public GiantMillipede(int l){
        super(l);
    }

    public void crawl(){
        System.out.println("In GiantMillipede.crawl()");
    }
}
```

Appendices to Paper 1 and Paper 2

```
public abstract class Cat extends Animal {

    public Cat(){
        super(4);
    }
    public abstract void purr();
}


public class Lion extends Cat implements Scary {

    public Lion(){
        System.out.println("In Lion.Lion()");
    }

    public void purr(){
        System.out.println("In Lion.purr()");
    }

    public void scare() {
        System.out.println("ROAR!!!");
    }

}


public interface Scary {

    public void scare();
}
```

Appendices to Paper 1 and Paper 2

Appendix 2: Class summaries

```
class java.awt.Graphics
abstract void dispose()
abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
abstract void drawLine(int x1, int y1, int x2, int y2)
abstract void drawOval(int x, int y, int width, int height)
abstract void drawString(String str, int x, int y)
abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
abstract void fillOval(int x, int y, int width, int height)
abstract void fillRect(int x, int y, int width, int height)
abstract void setColor(Color c)

class java.awt.color
static final black
static final BLACK
static final white
static final WHITE
static final red
static final RED
static final green
static final GREEN
static final blue
static final BLUE

class java.util.Date
public Date()
public boolean after(Date when)
public boolean equals(Object obj)
public boolean before(Date when)
public String toString();

class java.net.InetAddress
public static InetAddress getByName(String host) throws UnknownHostException
public String getHostName()
public String getHostAddress()
public static InetAddress getLocalHost() throws UnknownHostException

class java.net.URL
public URL(String spec) throws MalformedURLException
public final InputStream openStream() throws IOException
public String getHost()

class java.io.InputStream
public abstract int read() throws IOException
public int read(byte[] b) throws IOException
public int read(byte[] b, int off, int len) throws IOException
```

Appendices to Paper 1 and Paper 2

```
public void close() throws IOException

class java.io.OutputStream
public abstract int write() throws IOException
public int write(byte[] b) throws IOException
public int write(byte[] b, int off, int len) throws IOException
public int write(int b) throws IOException
public void close() throws IOException

java.net.Socket
public Socket(InetAddress address, int port) throws IOException
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
public void close() throws IOException

java.net.ServerSocket
public ServerSocket(int port) throws IOException
public void close() throws IOException
public Socket accept() throws IOException

java.applet.Applet
public URL getCodebase()
public AudioClip getAudioClip(URL u)

java.applet.AudioClip
pubic void play()
pubic void stop()
```

Answers to Paper 1 Part A

QUESTION 1

(a) Write a Java class `SimpleGui`. This class should have an instance method `public void go()` and a static `main` method. You do not need to provide a constructor for this class. The main method should instantiate a `SimpleGui` object and call `go()` on this object. However the `go()` method should be left blank.

[5 Marks]

```
public class SimpleGui {
   public void go(){

   }
   public static void main(String[] args){
      SimpleGui gui = new SimpleGui();
      gui.go();
   }
}
```

(b) Add code to `SimpleGui.java` so that the result of calling `go()` is to create and display a window of size 300 x 300. The programme should terminate when a user closes this window.

[5 Marks]

```
public void go(){
   JFrame frame = new JFrame();
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setSize(300, 300);
   frame.setVisible(true);
}
```

(c) Explain Java's event handling mechanism. You should consider the relationship between the event source (e.g. a button) and an object that should take some action as a result of this event occurring.

[10 Marks]

An application must listen for a GUI event in order to know when the user takes some action (such as clicking the button)

Listeners must be registered with an event source (e.g. button)

The listener interface provides a call-back mechanism

The interface defines a method that the event source will call when the event happens

Objects are registered for events with the source by calling the source's registration method

This is of the form `add<EventType>Listener`

Answers to Paper 1 Part A

For example, `addActionListener(this)` registers the object referenced by `this` for the ActionEvents fired by a button

The object must implement the interface's event-handling methods

The desired action is placed in the listener call-back method e.g. within `public void ActionPerformed(ActionEvent e` for a button-clicked event

An event object is passed to the event-handler method - this object contains information about the type and the source of the event

(d) Add a `JButton` to the window created in `SimpleGui`. This button should display the current date and time when a user clicks it.

[5 Marks]

```
// imports...
public class SimpleGui implements ActionListener{
   JButton button;

   public void go(){
      JFrame frame = new JFrame();
      button = new JButton();

      button.addActionListener(this);
      frame.getContentPane().add(button);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(300, 300);
      frame.setVisible(true);
   }

   public static void main(String[] args){
      SimpleGui gui = new SimpleGui();
      gui.go();
   }

   public void actionPerformed(ActionEvent arg0) {
      Date now = new Date();
      button.setText( now.toString() );
   }
}
```

Answers to Paper 1 Part A

QUESTION 2

(a) Inheritance is a key idea in object oriented programming. In a few sentences, explain the essence of this idea.

[5 Marks]

(NB non-static methods only) A subclass inherits all the non-private methods and instance variables of its superclass

Inherited methods can be overridden

The subclass can add extra, more-specialised methods and instance variables

A superclass reference can point to a subclass object

In this case, if the subclass overrides a superclass method, the overridden version of the method is invoked (dynamic binding)

(b) Under what circumstances should you use inheritance in your own coding? When should you guard against unnecessary use of inheritance?

[5 Marks]

Use inheritance when one class is more specific e.g. Dog IS-A animal

Use inheritance when you have some behaviour that should be shared amongst multiple classes of the same type.

Do not use just because you wish to re-use code from another class if the relationship between these classes violates the above two rules

Do not use if the subclass and superclass do not pass the IS-A test

Do not construct a deep inheritance tree because this separates code by too many steps. Keep the tree broad and shallow i.e. inheritance increases dependency and coupling

(c) In general terms, what are the advantages of inheritance?

[5 Marks]

Common code is in one place

This aids maintenance and development because updates need only be made in one place

Inheritance defines a common protocol for a group of classes

This allows other developers to add subtypes to your system by ensuring that they conform to the supertype

Polymorphism: a supertype reference can point to a subtype object. This means that you don't have to change code when you introduce new subtypes

(d) Appendix 1 shows a number of classes of a Jungle simulation. The class below, `JungleTestRun` has been written to test the simulation. What is the output of `JungleTestRun.main()`?

Answers to Paper 1 Part A

```
public class JungleTestRun {

    public static void main(String[] args) {

        CreepyCrawly bug = new GiantMillipede();
        bug.scare();
    }
}
```

[5 Marks]

```
In Animal.Animal()
In CreepyCrawly.CreepyCrawly()
In GiantMillipede.GiantMillipede()
Crawling up your leg!!
In GiantMillipede.crawl()
```

(e) Which (if any) of the following lines will compile if they are added to `JungleTestRun.main()`? Explain your answer.

```
Cat c = new Cat();
Cat d = new Lion();
```

[5 Marks]

`Cat c = new Cat()` will not compile because Cat is an abstract class. (2 marks)

However `Cat d = new Lion()` WILL compile because Lion is concrete and you can have an abstract reference to a concrete class (3 marks)

Answers to Paper 1 Part A

QUESTION 3

(a) The Java Virtual Machine (JVM) organises memory into two parts; the stack and the heap. What is the stack and how is it used by the JVM?

[5 Marks]

The stack is an ordered block of memory

The JVM pushes method invocations and values of local variables onto the stack

Technically this is called a stack frame

The method at the top of the stack is the currently executing method

When the method ends, the stack frame is removed and the method below resumes

(b) What is the heap and how is it used by the JVM?

[5 Marks]

The heap has no defined order

All objects are placed in the heap

This is true if the reference to this object is local or if it is an instance variable

The heap is garbage collectible

this means that the garbage collector destroys any objects that are no longer referenced

(c) Draw a stack-heap diagram to illustrate the relationships between variables and objects for the code below:

```
class DogPound{

   public static void main(String[] args){

      Dog dog1 = new Dog();
      Dog dog2 = new Dog();
      ...
   }
}
```

[5 Marks]

Answers to Paper 1 Part A

(d) Draw a stack-heap diagram to illustrate the relationships between variables and objects just after line 3 for the code below. What will happen to the object referenced by `dog1` in line 1?

```
class DogPound{

   public static void main(String[] args){

      Dog dog1 = new Dog(); // line 1
      Dog dog2 = new Dog(); // line 2
      dog1 = dog2; // line 3
      ...
   }
}
```

[5 Marks]

The Dog object in line 1 is eligible for garbage collection.

(e) An object is eligible for garbage collection when its last live reference disappears. Demonstrate with code excerpts how this might happen.

Answers to Paper 1 Part A

[5 Marks]

This might happen in one of three ways

1. The reference is permanently out of scope

```
void go(){
   Dog bongo = new Dog();
} // end scoope
```

2. The reference is re-assigned

```
Dog bongo = new Dog();
bongo = new Dog(); // re-assign
```

3. The reference is set to null

```
Dog bongo = new Dog();
bongo = null;
```

Award two marks for each circumstance and code snippet up to a total of 5

Answers to Paper 2 Part A

QUESTION 1

(a) Appendix 1 shows a number of classes of a Jungle simulation. The class below, `JungleTestRun`, has been written to test the simulation. What is the output of `JungleTestRun.main()`?

[8 Marks]

```
import java.util.ArrayList;

public class JungleTestRun {

   public static void main(String[] args) {

      ArrayList<Scary> list = new ArrayList<Scary>();
      list.add(new Lion());
      list.add(new GiantMillipede());
      for (Scary s : list)
         s.scare();
   }
}
```

```
In Animal.Animal()
In Lion.Lion()
In Animal.Animal()
In CreepyCrawly.CreepyCrawly()
In GiantMillipede.GiantMillipede()
ROAR!!!
Crawling up your leg!!
In GiantMillipede.crawl()
```

(b) Which (if any) of the following lines will compile if they are added to `JungleTestRun.main()`? Explain your answer.

```
Animal a = new Cat();
Animal a = new Animal();
```

[4 Marks]

`Animal a = new Cat()` will not compile because Cat is abstract and abstract classes cannot be instantiated. `Animal a = new Animal()` will not compile because the constructor Animal() is undefined. This is because a constructor with an argument has already been defined in `Animal`.

(c) `Animal, Cat and CreepyCrawly` have been declared as `abstract` classes in the Jungle simulation. Why is this?

[3 Marks]

Answers to Paper 2 Part A

These are categories of things (concepts). Object programming advises that actual things are instantiated as objects. e.g. an 'animal' has no physical form, but a Lion does

(d) The Jungle simulation defines an Interface `Scary`. What is the purpose of interfaces in the Java language? You should illustrate your answer by referring to the Jungle simulation.

[10 Marks]

Sometimes you wish to share code between classes in different branches of an inheritance tree.

This might be the case when different classes share common behaviour.

For example, in the Jungle simulation, CreepyCrawlys and Lions are scary.

In languages with multiple inheritance, CreepyCrawly and Lion could both extend an additional class.

But this would lead to ambiguity if both superclasses override a method of a higher class (i.e. both superclasses have a method of the same name).

The Java solution is to define a common interface of entirely abstract methods

Implementing classes must therefore provide code for these methods and the ambiguity problem cannot arise

Interfaces define a type, so that e.g. in Jungle, a Scary reference can point to either a Lion or a CreepyCrawly

This is an example of polymorphism

A key idea in object programming, allowing for greater flexibility of design and maintenance

Answers to Paper 2 Part A

QUESTION 2

(a) The code below outlines the class `SimpleAnimation`.

```
import java.awt.Graphics;

public class SimpleAnimation{

    Graphics graphics;
    MyDrawPanel panel;

    int x, y;

    public static void main(String[] args) {
        new SimpleAnimation().setup();
    }

    public void setup() {

    }

    public void draw() {

    }
}
```

Add code to `setup()` so that the result of calling `setup()` on a SimpleAnimation object is to create and display a `JFrame` of size 300 x 300. The programme should terminate when a user closes this window.

[5 Marks]

Award 1 mark per correct line up to a max of 5

```
public void setup() {

   JFrame frame = new JFrame();
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setSize(300, 300);
   frame.setVisible(true);
}
```

(b) Add an inner class `MyDrawPanel extends JPanel` to `SimpleAnimation`. This class should override `public void paintComponent(Graphics g)`. `paintComponent` should assign `g` to the instance variable `graphics`, and call `draw`.

[5 Marks]

Answers to Paper 2 Part A

```
class SimpleAnimation{
   ...
   class MyDrawPanel extends JPanel {
      public void paintComponent(Graphics g) {
         graphics = g;
         draw();
       }
   }
}
```

(c) Include two lines of code in `setup()` to instantiate a `MyDrawPanel` object and add it to your `JFrame`.

[5 Marks]

Award 1 mark per correct line up to a max of 5

```
class SimpleAnimation{

   public void setup() {
      JFrame frame = new JFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      panel = new MyDrawPanel();
      frame.getContentPane().add(panel); // can be combined in one line

      frame.setSize(300, 300);
      frame.setVisible(true);
   }
}
```

(d) Add code to `draw` so that the result of calling this method is to increment x and y by one pixel, clear the panel by painting it white, and then to draw a filled green circle at coordinates (x, y). (Your attention is drawn to the class summaries given in Appendix 2 of this paper)

[5 Marks]

Award 1 mark per correct line up to a max of 5

```
public void draw() {
   x++;
   y++;
   graphics.setColor(Color.white);
   graphics.fillRect(0, 0, 300, 300);
   graphics.setColor(Color.green);
   graphics.fillOval(x, y, 40, 40);
}
```

(e) Include a block of code in `setup()` that will repaint the panel at a rate of 20 frames per second, and hence produce an animation of a slowly moving ball.

Answers to Paper 2 Part A

[5 Marks]

Award 1 mark per correct line up to a max of 5

```
public void setup(){
...
   while (true) {
      panel.repaint();
      try {
         Thread.sleep(50);
      } catch (Exception e) {
      }
   }
}
```

Paper 3 Part A

QUESTION 1

(a) How does the *value* of a primitive variable differ from the *value* of a reference variable? Write a couple of assignment statements to illustrate your answer.

[5 Marks]

(b) Draw a memory diagram to illustrate your answer to the above question.

[5 Marks]

(c) Explain how methods are stacked by the JVM.

[5 Marks]

(d) Explain, in a few lines, the concept of local and instance variable. Include in your answer an account of how the JVM allocates memory for these two types of variables.

Is there any difference in the way the JVM handles the memory allocation of primitive and reference variables?

[5 Marks]

(e) Compare the lifetimes of local and instance variables. What is the difference between life and scope for local variables?

[5 Marks]

Paper 3 Part A

QUESTION 2

(a) Write a class `Clock` with a single instance variable `public long time` and a method `protected long now()` which returns the system time in millisecond units. The class should have a default constructor that sets the value of `time` to 0.

[5 Marks]

(b) Add methods `public long start()` and `public long stop()` to `Clock` so that a call to `start()` and then to `stop()` returns the duration in milliseconds of the interval between calling these two methods. The variable `time` should be reset to this duration.

[5 Marks]

(c) Write a static test method of `Clock` that times the execution of a computationally intensive block of code. You can invent any block of code to time that you wish, and you should include this code block within the test method.

[5 Marks]

(d) What is *encapsulation*? `Clock` is not a correctly encapsulated class. Rewrite `Clock` so that it conforms to this principle.

[5 Marks]

(e) Extend `Clock` by writing a class `NanoClock`. You should override `now()` so that it returns the system time in nanoseconds, using the library method `public static long nanoTime()` of `System`. Include a main method for testing your new class.

[5 Marks]

Paper 3 Part A

QUESTION 3

(a) What are the advantages of using class inheritance in a Java project?

[5 Marks]

(b) (i) What is an abstract method?
(ii) What is an abstract class?
(iii) Can an abstract class be instantiated?
(iv) Under what circumstances would you wish to use abstract classes?
(v) Give an example of such a circumstance.

[5 Marks]

(c) Your software team has been asked to design and implement an animation of the night sky. One of your team has produced outline classes for meteorites and satellites, reproduced below:

```
public class Meteorite{

    private float xPos, yPos;
    private float xVel, yVel;
    private final float vMax = 2.0f;

    public Meteorite(float x, float y) {
        xPos = x;
        yPos = y;
        xVel = (float) Math.random() * vMax;
        yVel = (float) Math.random() * vMax;

    }

    public void move() {
        xPos += xVel;
        yPos += yVel;
    }

   // getters for xPos, yPos, xVel and yVel

    public void draw() {
        // code to draw a meteorite
    }
}
```

Paper 3 Part A

```
public class Satellite{

    private float xPos, yPos;
    private float xVel, yVel;
    private final float vMax = 2.0f;

    public Satellite1(float x, float y) {
        xPos = x;
        yPos = y;
        xVel = (float) Math.random() * vMax;
        yVel = (float) Math.random() * vMax;

    }

    public void move() {
        xPos += xVel;
        yPos += yVel;
    }

    // getters for xPos, yPos, xVel and yVel

    public void draw() {
        // code to draw a satellite
    }
}
```

You immediately realise that `Meteorite` ad `Satellite` have code in common, and that this code can be refactored into a higher level `abstract` class, `SkyObject`. Write a class outline for `SkyObject`.

[5 Marks]

(d) Write a class outline for the new version of `Satellite`, which should subclass `SkyObject`

[5 Marks]

```
public class Satellite2 extends SkyObject {

    public Satellite2(float x, float y){
        super(x, y);
    }

    public void draw(){
        // code to draw a satellite
    }
}
```

Paper 3 Part A

(e) Write a class `SkyTest` which can be used to test your classes. You should write a test to make sure that the satellite and meteorite objects have been correctly initialised and that the method(s) `move` function correctly.

[5 Marks]

Paper 4 Part A

QUESTION 1

The purpose of this question to develop a Swing animation of a falling ball.

(a) Write a class `MyDrawPanel extends JPanel` with a single method `public void paintComponent(Graphics g)`.
This method should draw a white rectangle whose width and height are equal to the width and height of the parent `JPanel`.

[5 Marks]

(b) Write a class skeleton `SimpleAnimation` with an instance method `public void go()` and a `main` method. The body of `go` should be left blank. The `main` method should instantiate a `SimpleAnimation` object and then call `go` on this object. Include `MyDrawPanel` as an inner class in `SimpleAnimation`.

[5 Marks]

(c) Now include code in `go` in order to display a frame of width and height 300 pixels. A `MyDrawPanel` object should be inserted inside the frame.

[5 Marks]

```
public void go(){
        JFrame frame = new JFrame();

        MyDrawPanel drawPanel = new MyDrawPanel();

        frame.getContentPane().add(drawPanel);
        frame.setSize(300, 300);
        frame.setVisible(true);
 }
```

(d) Add code to `MyDrawPanel` to draw a filled green ball of height and width 40 pixels at co-ordinates (x, y) where x and y are integer instance variables of `SimpleAnimation`.

[5 Marks]

(e) Finally complete the `go` method to make the ball fall at a speed of one pixel every 50 milliseconds.

[5 Marks]

Paper 4 Part A

QUESTION 2

(a) Explain in a few lines the concept of multiple inheritance, the danger inherent in this practice, and how Java protects the programmer from this danger.

[5 Marks]

(b) The Java language introduces an 'interface' as a second reference type alongside class reference types.

(i) Can a class implement multiple interfaces?

(ii) Can a class extend one class and implement an interface as well?

(iii) Interfaces may have constructors. True or false?

(iv) Interface methods are implicitly modified by which two modifiers?

[5 Marks]

(c) Consider these three classes:

```
public class Animal {
    public Animal(){
        System.out.println("Making an Animal");
    }
}

public class Hippo extends Animal {
    public Hippo(){
        System.out.println("Making a Hippo");
    }
}

public class HippoTest {
    public static void main(String[] args){
        System.out.println("Starting...");
        Hippo h = new Hippo();
    }
}
```

What is the output when `HippoTest` is run?

[5 Marks]

(d) Draw a representation of a Hippo object on the heap.

[5 Marks]

Paper 4 Part A

(e) Draw diagrams to show how stack frames are added and removed from the stack in order to create a `Hippo` object.

[5 Marks]

Paper 4 Part A

QUESTION 3

Your development team has been asked to devise a Java graphical package that provides classes for drawing 'doodles'. A doodle is a small sketch formed of one or several lines. Examples include boxes, spirals and scribbles.

(a) Your team decides that all doodles shall inherit from an abstract superclass `Doodle`. A method `public void draw(Graphics g)` must be callable on any `Doodle` oblect so that it can be drawn in a Swing application.

Write class outlines for the `Doodle` superclass and for a `Box`.

[10 Marks]

(b) Some of your doodles are *shapes* in the sense that they have an inside and an outside. For example boxes are shapes, but spirals are not. Shapes should respond to messages such as `contains(x, y)` (returning true if (x, y) is inside the shape) and `getBounds()` which returns the smallest `Rectangle` that lies outside the shape.

One solution is to include `contains` and `getBounds()` methods in those doodles that happen to be shapes. What are the problems with this solution?

[5 Marks]

(c) What would be a better way of structuring the package? Provide code to illustrate your answer and include an implementation of `contains(x, y)`.

[10 Marks]

Appendix to Paper 3 and Paper 4

Appendix: Class summaries

```
class java.awt.Graphics
abstract void dispose()
abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
abstract void drawLine(int x1, int y1, int x2, int y2)
abstract void drawOval(int x, int y, int width, int height)
abstract void drawString(String str, int x, int y)
abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
abstract void fillOval(int x, int y, int width, int height)
abstract void fillRect(int x, int y, int width, int height)
abstract void setColor(Color c)

class java.awt.color
static final black
static final BLACK
static final white
static final WHITE
static final red
static final RED
static final green
static final GREEN
static final blue
static final BLUE

class java.util.Date
public Date()
public boolean after(Date when)
public boolean equals(Object obj)
public boolean before(Date when)
public String toString();

class java.lang.StringBuffer
public StringBuffer()
public StringBuffer(String str)
public StringBuffer append(char c)
public StringBuffer append(String str)
public StringBuffer delete(int start, int end)
public int indexOff(String str)
public insert(int offset, Srring str)
public int length()
public String toString()

class java.net.InetAddress
public static InetAddress getByName(String host) throws UnknownHostException
public String getHostName()
public String getHostAddress()
```

Appendix to Paper 3 and Paper 4

```
public static InetAddress getLocalHost() throws UnknownHostException

class java.net.URL
public URL(String spec) throws MalformedURLException
public final InputStream openStream() throws IOException
public String getHost()

class java.io.InputStream
public abstract int read() throws IOException
public int read(byte[] b) throws IOException
public int read(byte[] b, int off, int len) throws IOException
public void close() throws IOException

class java.io.OutputStream
public abstract int write() throws IOException
public int write(byte[] b) throws IOException
public int write(byte[] b, int off, int len) throws IOException
public int write(int b) throws IOException
public void close() throws IOException

java.net.Socket
public Socket(InetAddress address, int port) throws IOException
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
public void close() throws IOException

java.net.ServerSocket
public ServerSocket(int port) throws IOException
public void close() throws IOException
public Socket accept() throws IOException

java.applet.Applet
public URL getCodebase()
public AudioClip getAudioClip(URL u)

java.applet.AudioClip
pubic void play()
pubic void stop()
```

Answers to Paper 3 Part A

QUESTION 1

(a) How does the *value* of a primitive variable differ from the *value* of a reference variable? Write a couple of assignment statements to illustrate your answer.

[5 Marks]

The value of a primitive variable is a literal value

but the value of a reference variable is the address of the object on the heap

```
int i = 3;
Box b = new Box();
```

(b) Draw a memory diagram to illustrate your answer to the above question.

[5 Marks]

Simple stack heap diagram showing i and b on the stack, a Box object on the heap.

See HFJ p 55

(c) Explain how methods are stacked by the JVM.

[5 Marks]

A stack frame is placed on the stack for every called method.

The stack frame holds the state of the method,

i.e. which line of code is executing and the values of all local variables.

The method on the top of the stack is currently running.

The top method is removed from the stack when the final left brace has been reached and execution resumes with the method immediately below.

(d) Explain, in a few lines, the concept of local and instance variable. Include in your answer an account of how the JVM allocates memory for these two types of variables.

Is there any difference in the way the JVM handles the memory allocation of primitive and reference variables?

[5 Marks]

Local variables are declared inside a method.

Instance methods are declared in a class, but outside any method.

Local variables are placed on the stack when the method is called.

Instance variables are stored with the object on the heap.

Primitive and reference variables are treated in the same way.

Answers to Paper 3 Part A

(e) Compare the lifetimes of local and instance variables. What is the difference between life and scope for local variables?

[5 Marks]

A local variable lives only whilst its enclosing method is on the stack.

A local variable is in scope from its point of declaration within the enclosing method, to the end of the method.

An instance variable lives as long as the object is on the heap.

A local method may enter and leave scope.

It is in scope whilst its enclosing method is on top of the stack, but goes out of scope if another method is stacked on top.

Answers to Paper 3 Part A

QUESTION 2

(a) Write a class `Clock` with a single instance variable `public long time` and a method `protected long now()` which returns the system time in millisecond units. The class should have a default constructor that sets the value of `time` to 0.

[5 Marks]

```
public class Clock {

    public long time;

    public Clock() {
        time = 0;
    }

    protected long now(){
        return System.currentTimeMillis();
    }
}
```

(b) Add methods `public long start()` and `public long stop()` to `Clock` so that a call to `start()` and then to `stop()` returns the duration in milliseconds of the interval between calling these two methods. The variable `time` should be reset to this duration.

[5 Marks]

```
public class Clock {

    public long time;

    public Clock() {
        time = 0;
    }

    protected long now(){
        return System.currentTimeMillis();
    }

    public void start() {
        time = now();
    }

    public long stop() {
        time = now() - time;
        return time;
    }
 }
```

Answers to Paper 3 Part A

(c) Write a static test method of `Clock` that times the execution of a computationally intensive block of code. You can invent any block of code to time that you wish, and you should include this code block within the test method.

[5 Marks]

```
public static void main(String[] args){
        Clock clock = new Clock();
        clock.start();
        // How long does it take to create 10 million strings?
        for (int i = 0; i < 10000000; ++i)
            new String("tick-tock");
        System.out.println(clock.stop());
    }
```

(d) What is *encapsulation*? `Clock` is not a correctly encapsulated class. Rewrite `Clock` so that it conforms to this principle.

[5 Marks]

Encapsulation means that data is hidden by marking instance variables `private`.

The data is accessed through `public` getters and setters

```
public class Clock {

    private long time;

    public long getTime(){
        return time;
    }
     // rest of class as before

}
```

(e) Extend `Clock` by writing a class `NanoClock`. You should override `now()` so that it returns the system time in nanoseconds, using the library method `public static long nanoTime()` of `System`. Include a main method for testing your new class.

[5 Marks]

```
public class NanoClock extends Clock{

    protected long now(){
        return System.nanoTime();
    }

    public static void main(String[] args){
        Clock clock = new NanoClock();
        clock.start();
        // How long does it take to create 10 milliion strings?
```

Answers to Paper 3 Part A

```
        for (int i = 0; i < 10000000; ++i)
            new String("tick-tock");
        System.out.println(clock.stop());
    }
}
```

Answers to Paper 3 Part A

QUESTION 3

(a) What are the advantages of using class inheritance in a Java project?

[5 Marks]

Inheritance means that code common to a number of classes can be put in one place.
Subclasses inherit this code from a superclass
and modifications only need to be made in one place.
A superclass defines a common protocol for a group of classes.
This keeps the design tidy and helps designers to add more classes at a later date.

(b) (i) What is an abstract method?
(ii) What is an abstract class?
(iii) Can an abstract class be instantiated?
(iv) Under what circumstances would you wish to use abstract classes?
(v) Give an example of such a circumstance.

[5 Marks]

(i) An abstract method has no body and the declaration ends in a semicolon
(ii) An abstract class has at least one abstract method
(iii) No
(iv) Use an abstract class in order to define a protocol for a group of classes, but when it doesn't make sense to instantiate the class
(v) `Animal` (abstract superclass) `Dog` (concrete subclass)

(c) Your software team has been asked to design and implement an animation of the night sky. One of your team has produced outline classes for meteorites and satellites, reproduced below:

```
public class Meteorite{

    private float xPos, yPos;
    private float xVel, yVel;
    private final float vMax = 2.0f;

    public Meteorite(float x, float y) {
        xPos = x;
        yPos = y;
        xVel = (float) Math.random() * vMax;
        yVel = (float) Math.random() * vMax;
```

Answers to Paper 3 Part A

```
    }

    public void move() {
        xPos += xVel;
        yPos += yVel;
    }

   // getters for xPos, yPos, xVel and yVel

    public void draw() {
        // code to draw a meteorite
    }
}

public class Satellite{

    private float xPos, yPos;
    private float xVel, yVel;
    private final float vMax = 2.0f;

    public Satellite1(float x, float y) {
        xPos = x;
        yPos = y;
        xVel = (float) Math.random() * vMax;
        yVel = (float) Math.random() * vMax;

    }

    public void move() {
        xPos += xVel;
        yPos += yVel;
    }

    // getters for xPos, yPos, xVel and yVel

    public void draw() {
        // code to draw a satellite
    }
}
```

You immediately realise that `Meteorite` ad `Satellite` have code in common, and that this code can be refactored into a higher level `abstract` class, `SkyObject`. Write a class outline for `SkyObject`.

[5 Marks]

```
public abstract class SkyObject {
```

Answers to Paper 3 Part A

```
    private float xPos, yPos;
    private float xVel, yVel;
    private final float vMax = 2.0f;
    public SkyObject(float x, float y){
        xPos = x;
        yPos = y;
        xVel = (float)Math.random() * vMax;
        yVel = (float)Math.random() * vMax;

    }
    public void move(){
        xPos += xVel;
        yPos += yVel;
    }
    public abstract void draw();

    // getters for xPos, yPos, xVel and yVel
}
```

(d) Write a class outline for the new version of `Satellite`, which should subclass `SkyObject`

[5 Marks]

```
public class Satellite2 extends SkyObject {

    public Satellite2(float x, float y){
        super(x, y);
    }

    public void draw(){
        // code to draw a satellite
    }
}
```

(e) Write a class `SkyTest` which can be used to test your classes. You should write a test to make sure that the satellite and meteorite objects have been correctly initialised and that the method(s) `move` function correctly.

[5 Marks]

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

        SkyObject sat = new Satellite2(50, 75);
        SkyObject met = new Meteorite2(100, 25);
        System.out.println(sat.getX() + ", " + sat.getY());
        System.out.println(met.getX() + ", " + met.getY());
        sat.move();
        met.move();
        System.out.println(sat.getX() + ", " + sat.getY());
        System.out.println(met.getX() + ", " + met.getY());
    }
}
```

Answers to Paper 4 Part A

QUESTION 1

The purpose of this question to develop a Swing animation of a falling ball.

(a) Write a class `MyDrawPanel extends JPanel` with a single method `public void paintComponent(Graphics g)`.
This method should draw a white rectangle whose width and height are equal to the width and height of the parent `JPanel`.

[5 Marks]

```
class MyDrawPanel extends JPanel {

   public void paintComponent(Graphics g) {
         g.setColor(Color.white);
         g.fillRect(0, 0, this.getWidth(), this.getHeight());
   }
}
```

(b) Write a class skeleton `SimpleAnimation` with an instance method `public void go()` and a `main` method. The body of `go` should be left blank. The `main` method should instantiate a `SimpleAnimation` object and then call `go` on this object. Include `MyDrawPanel` as an inner class in `SimpleAnimation`.

[5 Marks]

```
public class SimpleAnimation {


    public static void main(String[] args) {
       SimpleAnimation gui = new SimpleAnimation();
       gui.go();
    }

    public void go() {
     }

    class MyDrawPanel extends JPanel {

       // as before
    }
}
```

(c) Now include code in `go` in order to display a frame of width and height 300 pixels. A `MyDrawPanel` object should be inserted inside the frame.

[5 Marks]

Answers to Paper 4 Part A

```
public void go(){
        JFrame frame = new JFrame();

        MyDrawPanel drawPanel = new MyDrawPanel();

        frame.getContentPane().add(drawPanel);
        frame.setSize(300, 300);
        frame.setVisible(true);
 }
```

(d) Add code to `MyDrawPanel` to draw a filled green ball of height and width 40 pixels at co-ordinates (x, y) where x and y are integer instance variables of `SimpleAnimation`.

[5 Marks]

```
public class SimpleAnimation {

    int x = 70;
    int y = 70;

   // as before

    class MyDrawPanel extends JPanel {

        public void paintComponent(Graphics g) {

            // as before

            g.setColor(Color.green);
            g.fillOval(x, y, 40, 40);
        }
    }
}
```

(e) Finally complete the `go` method to make the ball fall at a speed of one pixel every 50 milliseconds.

```
    public void go() {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        MyDrawPanel drawPanel = new MyDrawPanel();

        frame.getContentPane().add(drawPanel);
        frame.setSize(300, 300);
        frame.setVisible(true);

        for (int i = 0; i < 130; i++) { // 130 is arbitrary

            y++;

           drawPanel.repaint();

            try {
```

Answers to Paper 4 Part A

```
                Thread.sleep(50);
            } catch (Exception ex) {
            }
        }

    }
```

Answers to Paper 4 Part A

QUESTION 2

(a) Explain in a few lines the concept of multiple inheritance, the danger inherent in this practice, and how Java protects the programmer from this danger.

[5 Marks]

Multiple inheritance is when a class can inherit from more than one parent class.
It is dangerous because of possible ambiguities
for example, a given method name could exist in more than one inheritance tree
so there is no way of the knowing which would actually be called (in the absence of further rules)
Java escapes this deadly diamond of death by prohibiting multiple inheritance all together.

(b) The Java language introduces an 'interface' as a second reference type alongside class reference types.

(i) Can a class implement multiple interfaces?
(ii) Can a class extend one class and implement an interface as well?
(iii) Interfaces may have constructors. True or false?
(iv) Interface methods are implicitly modified by which two modifiers?

[5 Marks]

(i) Yes
(ii) Yes
(iii) False
(iv) `public` and `abstract`

(c) Consider these three classes:

```
public class Animal {
    public Animal(){
        System.out.println("Making an Animal");
    }
}

public class Hippo extends Animal {
    public Hippo(){
        System.out.println("Making a Hippo");
    }
}

public class HippoTest {
```

Answers to Paper 4 Part A

```
    public static void main(String[] args){
        System.out.println("Starting...");
        Hippo h = new Hippo();
    }
}
```

What is the output when `HippoTest` is run?

[5 Marks]

```
Starting...
Making an Animal
Making a Hippo
```

(d) Draw a representation of a Hippo object on the heap.

[5 Marks]

HFJ p 251. 2 marks for the general idea, one mark each for Hippo, Animal and Object blobs.

(e) Draw diagrams to show how stack frames are added and removed from the stack in order to create a `Hippo` object.

[5 Marks]

See HFJ page 252. Two marks for the general idea, then 0.5 mark for each stack in p252, plus one more mark to show the removal of Animal and then Hippo blobs.

Answers to Paper 4 Part A

QUESTION 3

Your development team has been asked to devise a Java graphical package that provides classes for drawing 'doodles'. A doodle is a small sketch formed of one or several lines. Examples include boxes, spirals and scribbles.

(a) Your team decides that all doodles shall inherit from an abstract superclass `Doodle`. A method `public void draw(Graphics g)` must be callable on any `Doodle` oblect so that it can be drawn in a Swing application.

Write class outlines for the `Doodle` superclass and for a `Box`.

[10 Marks]

```
public abstract class Doodle {
    public abstract void draw(Graphics g);
}

public class Box extends Doodle{
    private int x, y, w ,h;
    public Box(int x, int y, int w, int h){
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }
    public void draw(Graphics g){
        g.drawRect(x, y, w, h);
    }
}
```

(b) Some of your doodles are *shapes* in the sense that they have an inside and an outside. For example boxes are shapes, but spirals are not. Shapes should respond to messages such as `contains(x, y)` (returning true if (x, y) is inside the shape) and `getBounds()` which returns the smallest `Rectangle` that lies outside the shape.

One solution is to include `contains` and `getBounds()` methods in those doodles that happen to be shapes. What are the problems with this solution?

[5 Marks]

The problem is that if we decide later that *shapes* have additional properties

such as `intersects`

we have to dig out all Doodles that are also shapes and make the alterations one by one.

Furthermore if another development team adds other doodles which are also shapes, there is no obvious way of ensuring that these shapes have the correct methods.

(c) What would be a better way of structuring the package? Provide code to illustrate your answer and include an implementation of `contains(x, y)`.

[10 Marks]

Answers to Paper 4 Part A

A better solution is to write a `Shape` interface.
Then any Doodle might become a 'shape' by implementing this interface,
and the contract for Shape would be documented as a set of abstract methods.

```
public interface Shape {
    public abstract boolean contains(int x, int y);

}
class Box extends Doodle implements Shape{
 ...
 public boolean contains(int x, int y){
        return x >= this.x && x <= this.x + w && y >= this.y && y <= this.y + h;
    }
```

Comment form

We welcome any comments you may have on the materials which are sent to you as part of your study pack. Such feedback from students helps us in our effort to improve the materials produced for the International Programmes.

If you have any comments about this guide, either general or specific (including corrections, nonavailability of essential texts, etc.), please take the time to complete and return this form.

Title of this **subject guide** .....

.....

Name .....

.....

Address .....

.....

.....

.....

Email .....

Student number .....

For which qualification are you studying?.....

.....

Comments

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Please continue on additional sheets if necessary.

Date .....

Please send your comments on this form (or a photocopy of it) to:

Publishing Manager, International Programmes, University of London, Stewart House
32 Russell Square, London WC1B 5DN, UK.