UNIVERSITY OF LONDON | INTERNATIONAL PROGRAMMES

Graphical object-oriented and internet programming in Java Volume 1

T. Blackwell

CO2220

2009

Undergraduate study in
Computing and related programmes

This is an extract from a subject guide for an undergraduate course offered as part of the University of London International Programmes in Computing. Materials for these programmes are developed by academics at Goldsmiths.
For more information, see: www.londoninternational.ac.uk

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London International Programmes 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.

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

www.londoninternational.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

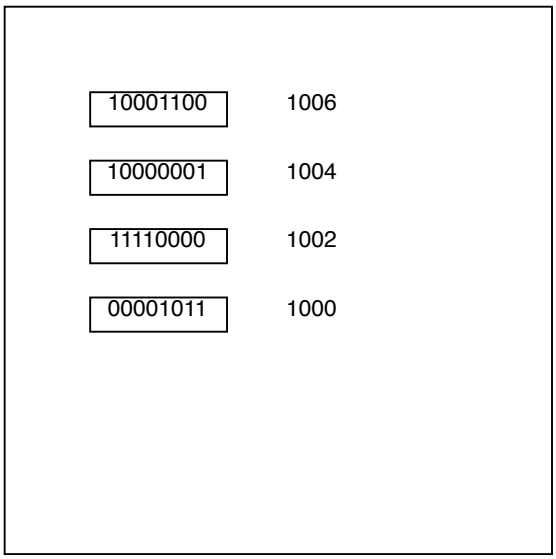

Figure 4.1: A portion of RAM showing four consecutive words of memory at addresses 1000, 1002, 1004 and 1006

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.

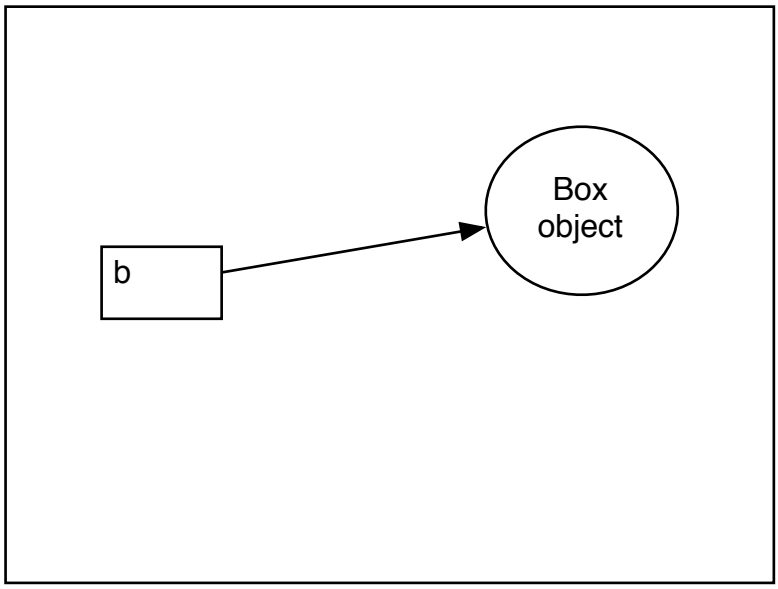

Figure 4.2: Memory diagram showing a reference variable `b` pointing at a `Box` object.

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.

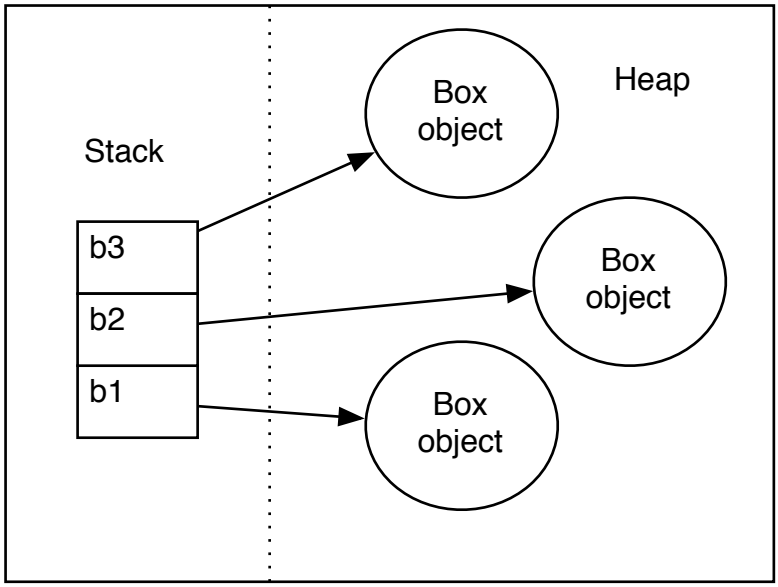

Figure 4.3: Memory diagram showing three `Box` reference variables and their objects

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.

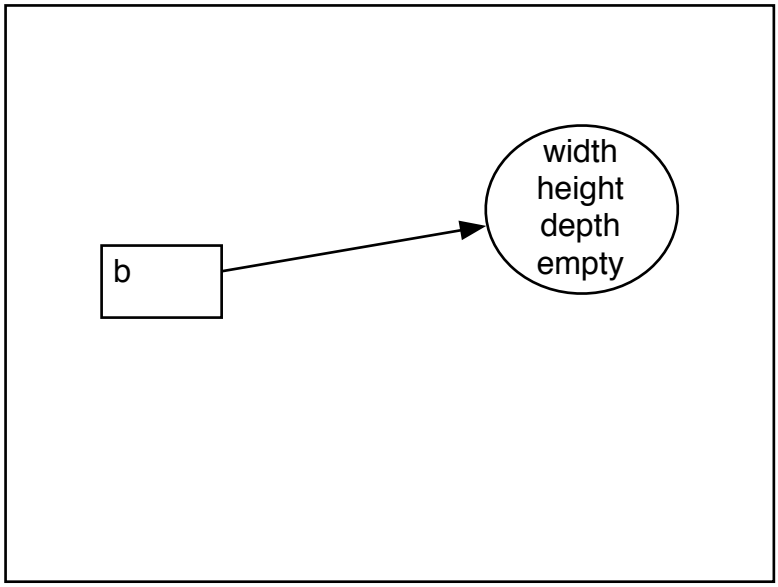

Figure 4.4: Instance variables live with their containing object, on the heap

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.