M255 Unit 3
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Variables, objects and representations

Unit 3

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Copyright © 2006, 2008 The Open University.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher or a licence from the Copyright Licensing Agency Ltd. Details of such licences (for reprographic reproduction) may be obtained from the Copyright Licensing Agency Ltd of 90 Tottenham Court Road, London, W1T 4LP.

Open University course materials may also be made available in electronic formats for use by students of the University. All rights, including copyright and related rights and database rights, in electronic course materials and their contents are owned by or licensed to The Open University, or otherwise used by The Open University as permitted by applicable law.

In using electronic course materials and their contents you agree that your use will be solely for the purposes of following an Open University course of study or otherwise as licensed by The Open University or its assigns.

Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of The Open University or in accordance with the Copyright, Designs and Patents Act 1988.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 5495 7

2.1

CONTENTS

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

In *Unit 2* you did a lot of practical work in self-contained microworlds, sending messages to different objects in order to discover the nature of objects – that they have state, have a protocol (the messages that they respond to) and that they are organised into classes.

In this unit we begin to lift the lid on how the behaviour and state of objects might be implemented. You will learn that there are two categories of data types in Java – primitive data types and classes. Values of primitive types are the nuts and bolts that can be represented at the machine level – numbers (`1234` or `564.33`), or Boolean values like `true` and `false`, or characters like `?`, `G` and `b`. Instances of classes are, as you have already learnt, objects. Ultimately, all objects (in Java) are represented at the machine level using primitive values. For example, `Account` objects have the attributes `balance` and `holder`. The attribute `balance` has a value of the primitive type `int`, and the attribute `holder` has a value which is a `String` object. However this `String` object is made up of values from the primitive type `char`. So, as will be demonstrated, ultimately everything in Java is composed from these primitive types.

This unit will also introduce you to the concept of expressions, chunks of program code that evaluate to a single value, and how that value can then be assigned to a variable so that a program can remember the value for future use.

You will also be introduced to BlueJ, which is an example of an IDE – an **integrated development environment**. Later on in the course you will use this IDE to modify classes of objects and to create entirely new classes. In this unit, however, you will be using BlueJ to access the 'OUWorkspace', a programming tool that the M255 course team have integrated into BlueJ. This tool is designed to enable you to get used to the syntax of Java and enable you to write and test snippets of Java code in a quick and convenient way.

1 Data types and variables

All computer programs deal with data. Data comes in all shapes and sizes. For example, it could be simple numbers such as temperature readings, or strings of characters such as the names of account holders, or complex objects such as bank accounts or frogs. If a program running on a computer is going to be able to make use of this data, the data needs to be stored in the computer's memory, and in order to access the stored data, the program needs some way of keeping track of where the data is stored.

To help it to do this a program can make use of variables. A **variable** is simply a named 'chunk' or block of the computer's memory where data (or a reference to the data) is stored. By using the name of the variable in program code we are able to access the data stored in the corresponding block of memory and also to change what is stored there. It is because the value that is stored in a memory location can change that we use the word variable for these named memory locations. For example, as you will see in the next unit, the position of a `Frog` object is stored in a variable called `position`. The value that is stored in `position` changes every time the frog moves to the right or the left. In other words the value of `position` varies; `position` is a variable.

In this section we explain that Java has two categories of data, and how each category is stored differently in a variable.

1.1 Primitive data types

Java distinguishes two sorts of data: values of primitive data types and instances of classes (objects). You have already met examples of both. Thus numbers such as `5` and `56.35` are examples of values of primitive data types, whereas the frogs, hoverfrogs and toads you interacted with in the microworlds in *Unit 2* are objects – instances of the `Frog`, `HoverFrog` and `Toad` classes.

A **primitive data type** is defined as a set of values together with operations that can be performed on them. The primitive data types in Java provide a set of basic building blocks from which all the more complex types of data can be built. There are three categories of primitive data type:

- numbers;
- characters;
- Booleans.

There are various kinds of **number types**. First there are the **integer types**, `byte`, `short`, `int` and `long`. The values of each of these types are integers (positive and negative whole numbers and zero); the differences are in the range of values they include and the amount of memory each requires. Thus a value of the `byte` type requires only 8 bits of memory but can only be an integer in the range –128 to 127. The size and range of the integer data types are summarised in the following table.

Type	Size	Range
`byte`	8 bits	–128 to 127
`short`	16 bits	–32768 to 32767
`int`	32 bits	–2147483648 to 2147483647
`long`	64 bits	–9223372036854775808 to 9223372036854775807

In this course we use mainly the `int` data type for integer values.

However, not all numbers are whole numbers; in order to store **decimal numbers** such as 17.34 we use **floating-point types**. There are two of these types in Java – `float` and `double`. A value of the `float` type takes up 32 bits of memory while one of the `double` type requires 64 bits. The `double` type not only covers a wider range of numbers (up to over 17 followed by 307 zeros, i.e. 17×10^{307}, compared with the much smaller 3 followed by 38 zeros, i.e. 3×10^{38}, for `float`) but, more importantly, has greater accuracy (more significant figures of the number are stored).

In order to represent particular values of any of these primitive types in a program we make use of what are called literals. A **literal** is just a textual representation of a particular value of some type. Some of these are pretty obvious. Thus `17` is the literal that represents the `int` 17. If you want to represent the same number 17 as a `long` you need to use the literal `17L`.

There are no special literals for representing the smaller types of integer; to store an integer as a `byte` or a `short` you need to assign the integer to a variable of the appropriate type.

For floating-point numbers the default type is `double`. Thus the literal `1.7` represents the `double` floating-point number 1.7. In order to specify a `float` you append an `F`; thus `1.7F` represents a value 1.7 of type `float`. Note that these appended letters can be in either upper case or lower case.

There is another way of representing floating-point numbers that is useful for very large or very small numbers. This is known as exponential notation. For example,

```
3.17E15
```

represents the number 3 170 000 000 000 000 (stored as a `double`). The `E15` means that the point should be moved 15 places to the right, requiring 13 extra 0s to be added.

Similarly,

```
3.17E-15
```

represents the number 0.000 000 000 000 003 17, as the `E-15` means the point has to be moved 15 places to the left.

We said above that a primitive data type is a set of values together with operations that can be performed on them. The operations on numbers include the familiar ones of addition, subtraction, multiplication and division. We will look at these and other operations in more detail in Section 2.

For characters Java uses just one primitive data type, `char`. The values of the type `char` represent both printable (e.g. A, B, x, and %) and non-printable characters (sometimes called control characters). In the Introduction you came across values of type `char` as the individual letters in `String` objects. So for example the string `"Java"` is made up of the characters `'J'`, `'a'`, `'v'` and `'a'`. Historically, characters were stored in 8 bits of memory because they were represented at machine level by a seven-bit **ASCII** code, which allowed for 128 characters. (ASCII stands for American Standard Code for Information Interchange and was originally devised as a standard coding system to allow text to be transferred from one device to another; the spare eighth bit was often used to help check whether the code had been corrupted during transfer between the

An example of a non-printable character would be a tab or a carriage return.

devices.) However, to code all the different characters that are used in all of the world's written languages requires many more characters than are available with a single byte. As a result a new character representation, **Unicode**, has gradually superseded ASCII since the late 1980s. Unicode characters are stored in 16 bits, and this is what Java uses for the primitive type `char`.

There are two ways of representing particular values of the data type `char`. One, which we used above, is to enclose the character in single quotes, for example `'c'` or `'%'`. We call this a character literal. The other method is to use the actual Unicode code for the character. This is done by writing `\u` followed by the code in hexadecimal. Thus `\u0063` represents the character whose code is 63 in hexadecimal (which is 99 in decimal); this is the character `'c'`. In this course we will normally use character literals such as `'c'` and not concern ourselves with Unicode codes.

Hexadecimal is the base-16 number system, which consists of 16 unique symbols: the digits 0 to 9 and the letters A to F. For example, the decimal number 15 is represented as F in the hexadecimal numbering system. One use of the hexadecimal system is that it can represent every byte (8 bits) as two consecutive hexadecimal digits.

Finally, Java uses the type `boolean` for Boolean data. There are only two values of this type and they are represented by the literals `true` and `false`.

The operations available on all these types will be dealt with in Section 2.

1.2 Variables of primitive data types

Now that we have introduced the primitive data types we need to consider how Java uses variables to hold values of these types in a program. Recall that a variable is a named 'chunk' or block of memory. Before you use a variable in a Java program you need to declare it; this will ensure that the variable is allocated a chunk of memory of a suitable size to store the required type of value. To declare a variable in Java you simply give the type of the variable and its name in a **statement** as shown in Figure 1. Note that a semicolon (`;`) must be used at the end of every statement to indicate to the Java interpreter that it has come to the end of the statement.

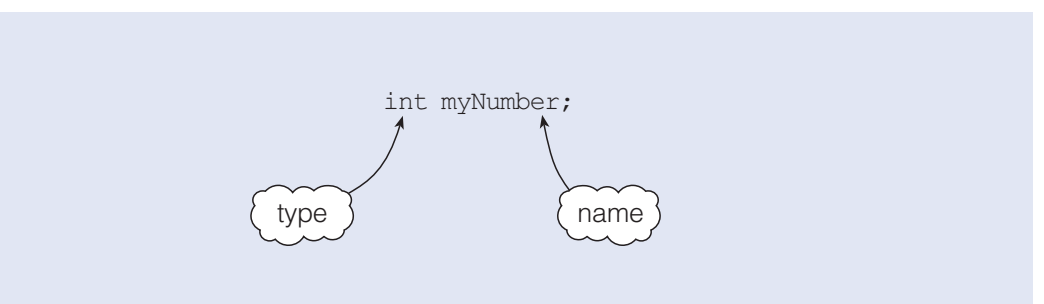

Figure 1 Declaring a variable

When this statement is executed it reserves a 32-bit block of the computer's memory (enough to store a value of type `int`) and records the identifier `myNumber` as the name of this block of memory. We can illustrate this as follows:

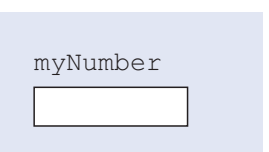

Figure 2 The result of variable declaration

In general it is sensible to assume that when a variable is first declared it is undefined (it has no value). However, you will learn later that there are some contexts in which variables are automatically provided with default values of the appropriate types when they are declared.

This shows a 'blank' block of memory (the empty rectangle) with the name `myNumber` and is a visual representation of what has happened as a result of the declaration of the variable `myNumber`. In particular the variable apparently has no value. This is because we have not yet **assigned** a value to it.

To assign a value to a variable you use the assignment operator =. For example, after the variable `myNumber` has been declared as above you can assign the value `17` to it by executing the statement

```
myNumber = 17;
```

This type of statement is called an **assignment statement**, and this example will cause the value `17` to be stored in the variable `myNumber`. We can illustrate this as follows.

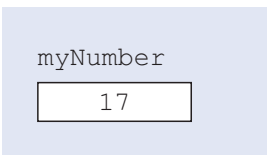

Figure 3 The result of an assignment statement

Statements are the building blocks of Java programs. Each statement represents a single instruction to the Java interpreter.

Note that the number is stored in the actual named block of memory.

Variables declared to hold values of some primitive data type are termed **value type variables**. This is because the 'chunk' of memory they label holds a value, as is illustrated in Figure 3.

Exercise 1

With pencil and paper, write two Java statements, one which will declare a character variable called `letter` and then another that will assign the letter `'D'` to the variable.

Solution

```
char letter;
letter = 'D';
```

1.3 BlueJ and the OUWorkspace

BlueJ encompasses a *programming language*, a *library of classes* and a *development environment*. All commercial (object-oriented) programming systems contain these three parts, with the major variation being in the environment. BlueJ is an example of a Java environment developed for teaching to which we (the course team) have added a general-purpose programming tool called the OUWorkspace.

The only part of the BlueJ system that you will use in this unit is our addition, the OUWorkspace, so details of other elements of the BlueJ system will be left until *Unit 4*.

Once you have launched BlueJ, you can open the OUWorkspace by selecting OUWorkspace from the Tools menu. The **OUWorkspace** is an important part of this course; it is a tool that enables you to write and test arbitrary Java **statements** in a quick and convenient way. You can think of a statement as a single *instruction* that has to be carried out by the computer when the statement is executed. You saw in Subsection 1.2 some examples of simple Java statements for declaring variables and assigning values to them. Remember that simple Java statements are all terminated with a semicolon.

ACTIVITY 1

Launch BlueJ by double-clicking the BlueJ icon, which is on your Windows desktop. Then select OUWorkspace from the BlueJ Tools menu.

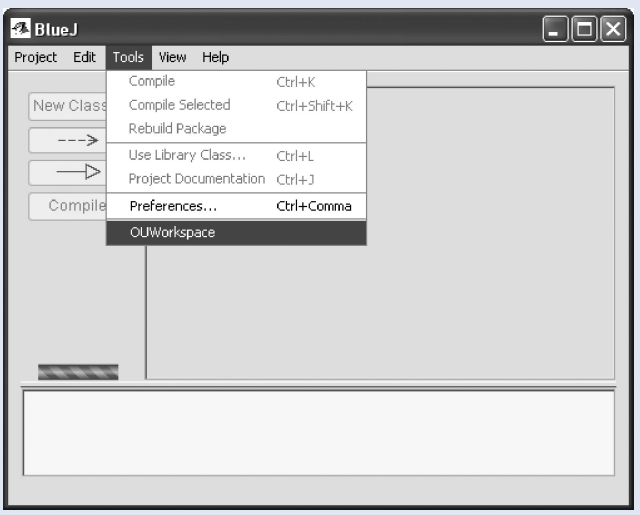

Figure 4 Opening the OUWorkspace

After selecting OUWorkspace from the Tools menu, the window shown in Figure 5 should open.

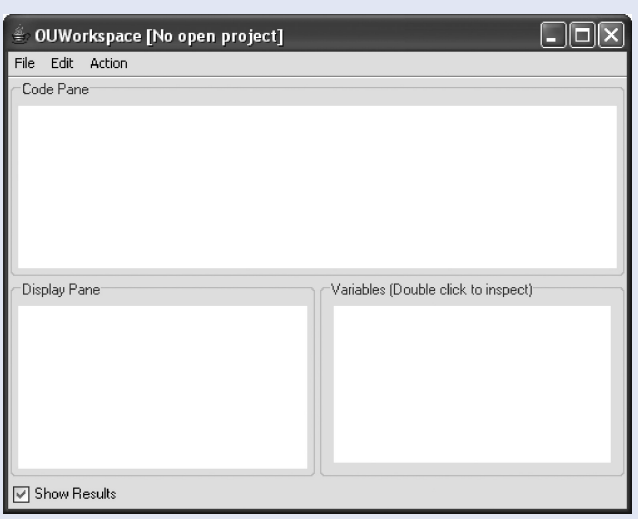

Figure 5 The OUWorkspace

The OUWorkspace has many similarities to the Accounts World used in *Unit 2*; however, it is more flexible and powerful, and will be used throughout the rest of the course.

The top pane in the OUWorkspace is called the Code Pane. This is where you type statements that you want executed. Statements are not executed until they have been highlighted (selected) and their execution requested by choosing Execute Selected from the Action menu.

Below the Code Pane there are two further panes – on the left there is the Display Pane. This has two functions. First, it is where the OUWorkspace's Java interpreter will write any error messages if you make a mistake in any statement(s) you are testing in the

Code Pane. For example, if you attempt to assign a value to a variable (`x`) before the variable has been declared, the Java interpreter that is used to parse and execute code in the Code Pane will write `Semantic error: Assignment to undeclared variable: x` in the Display Pane. The second function of the Display Pane is to display the value of the last expression evaluated in a statement (or series of statements). This will occur only if you check the Show Results check box.

To the right of the Display Pane is the Variables Pane. This pane will display any variables declared in the Code Pane.

1 Type the following code into the Code Pane (remembering the semicolons).

```
int myNumber;
myNumber = 17;
```

Ensure that the Show Results check box is checked.

Now highlight both lines of code and execute them by selecting Execute Selected from the Action menu.

What do you see in the Display Pane (bottom left) and in the Variables Pane (bottom right)?

What happens when you double-click the variable name in the Variables Pane?

2 Close the Inspector window and type in the following statement.

```
myNumber = 45;
```

Select and execute the statement. Inspect the variable `myNumber`.

3 Clear the Code Pane and the Display Pane by choosing the appropriate items from the Action menu.

Now enter the following statement into the Code Pane, highlight it and then execute it.

```
yourNumber = 34;
```

What happens? Try to explain it.

4 Enter and execute the following statement.

```
int yourNumber;
```

Describe what happens.

5 Enter and execute statements that will declare a character variable called `choice` and give it the value `'Q'`.

DISCUSSION OF ACTIVITY 1

1 After executing the code

```
int myNumber;
myNumber = 17;
```

the Variables Pane displays the name of the variable you have just declared (`myNumber`) and the Display Pane shows the number 17 – the result of executing `myNumber = 17;`. This is shown in Figure 6.

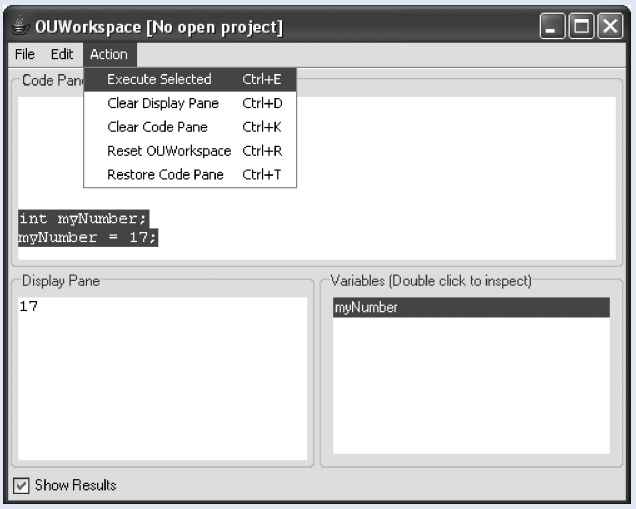

Figure 6 Executing code in the OUWorkspace

Although code is normally entered in the top left corner of the Code Pane, we have shown the snippet starting a little lower in Figure 6 to allow the Action menu to be displayed.

When you double-click `myNumber` in the Variables Pane, an Inspector window opens, as shown in Figure 7.

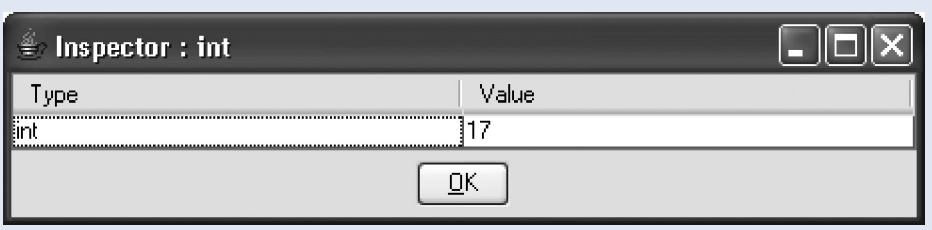

Figure 7 An Inspector window showing the value of the variable `myNumber`

2 The Display Pane should show the result `45` and the inspector should also show that `myNumber` has the value `45`. Both of these confirm that the value of the variable `myNumber` has changed from its previous value of `17`.

3 Executing the code

```
yourNumber = 34;
```

results in the Display Pane showing the following error message:

```
Semantic error: Assignment to undeclared variable: yourNumber
```

This error message is informing you that the variable `yourNumber` has not been declared, and therefore you cannot assign a value to it.

4 When you execute the statement

```
int yourNumber;
```

`yourNumber` appears in the Variables Pane.

5 The statements you need to declare a character variable called `choice` and give it the value `'Q'` are:

```
char choice;
choice = 'Q';
```

1.4 Reference type variables and objects

We noted at the end of Subsection 1.2 that variables declared to hold values of some primitive data type are termed value type variables. In this section we introduce you to another type of variable, **reference type variables**.

Reference type variables are variables which are declared to hold objects. As you saw in *Unit 2*, objects are instances of classes. Each class defines a reference type. For example the class `Frog` defines a reference type called `Frog`. The values that can be assigned to variables declared to be of the reference type `Frog` are simply the objects that are instances of the class `Frog`, or subclasses of that class.

There are a number of important differences between objects and values of primitive data types:

- The values of primitive data types are predefined; they already exist. The programmer never has to *create* primitive values like `42` or `3.14592` or `'X'`. They are *built into the language*. You just use them. In contrast, objects have to be created as required.
- You can create as many different instances (objects) of a class as you want. For example, there is no limit to the number of `Frog` objects that you can create. (You will see how to create a `Frog` object shortly.) Thus the number of instances of a class is potentially infinite. This is in contrast to a primitive data type which only has a finite number of values.
- A programmer can invent new classes of objects. You will find out how to make new classes in *Unit 4*. It is not possible to make new primitive data types in Java.
- Objects usually take up much more memory than values of primitive data types. This is because an object will normally have a number of attributes each of which will be another object or a value of some primitive data type. For example, a `Frog` object has two attributes: `position`, which holds a value of the primitive type `int`, and `colour`, which *references* an `OUColour` object.
- The space needed to store an object is also unpredictable because we cannot know at compile-time what class of object will actually be dynamically assigned to a variable at run-time. If a variable has been declared to hold, for example, `Frog` objects, it is legal to assign to it an instance of the class `HoverFrog`; instances of a subclass can be assigned to variables declared for its superclass. However, `HoverFrog` objects take up more storage than `Frog` objects (they have an extra attribute `height`). Indeed the class of an object held by a particular reference type variable may change at various times during the course of execution.

The last point is an important one. It means that we cannot always know in advance how much storage space a reference type variable may require. For this reason the variables that we use with objects work in a completely different way from variables that hold values of primitive data types. A reference type variable does not hold the actual object; instead the reference type variable contains a reference (the address) to where the object is stored in memory. We can illustrate this by using a **variable reference diagram**. Figure 8 shows a variable `gribbit` referencing a particular instance of the class `Frog`. The diagram shows that the variable holds a reference rather than the actual `Frog` by having an arrow coming out of the `gribbit` variable box pointing to a representation of the `Frog` object. Similarly while the attribute `position` of the `Frog` object holds the `int` value `5`, the attribute `colour` references an `OUColour` object (which in this case represents the colour brown).

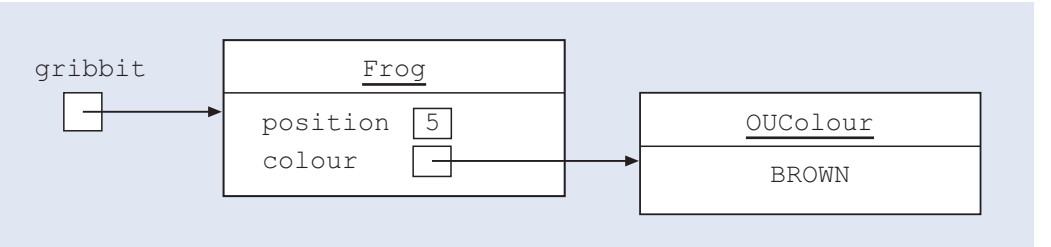

Figure 8 A variable reference diagram

Exercise 2

Why do you think that the position of the frog in Figure 8 is simply shown as a number in a box, whereas the colour is shown using an arrow pointing to another box?

Solution

The position of a `Frog` object is represented by an `int` value; this value is stored in a box labelled `position`. However, the colour of a `Frog` object is represented by an instance of the class `OUColour`; so by using the arrow we are able show the `colour` box *referencing* the `OUColour` object rather than containing it.

Reference type variables are declared in the same way as value type variables. First you give the name of the type (the class) and then the name of the variable. Thus

```
Frog kermit;
```

declares a reference type variable called `kermit` that can be used to reference instances of the class `Frog`. When a reference variable is first declared it does not reference anything; it simply holds the special value `null`.

Figure 9 The result of reference variable declaration

In order to give the variable `kermit` a sensible value, you first need to create a `Frog` object for it to reference. You do this by using a special operator `new` together with something called a constructor. As its name suggests, the operator `new` is used for creating new objects. A **constructor** is a piece of code for initialising the state of an object of a class when it is created. The constructors for a class are written by the programmer who implements the class; every class must have at least one constructor.

The constructor for the `Frog` class is `Frog()`. It is used in conjunction with the operator `new` as follows.

```
new Frog();
```

The operator `new` first creates a new `Frog` object. It knows that it has to create a `Frog` object from the name of the constructor that follows; the name of a constructor is always the same as the name of the class whose objects it initialises. The code of the constructor `Frog()` is then invoked and this code sets the attributes of the new `Frog` object to their initial values (`1` and `OUColour.GREEN`).

However, if you want to use this new `Frog` object in a program you need to have some way of keeping track of it. You do this by assigning it to a variable as soon as it has been

created. To assign the new `Frog` object to the variable `kermit` you need to execute the following statement.

```
kermit = new Frog();
```

Note that you need to do the assignment and creation together in one statement. If we used a separate statement to create the `Frog` object, then we would have no way of getting hold of it afterwards and would not be able to assign it to any variable.

The end result of executing the two statements

```
Frog kermit;
kermit = new Frog();
```

is to produce the situation illustrated in Figure 10.

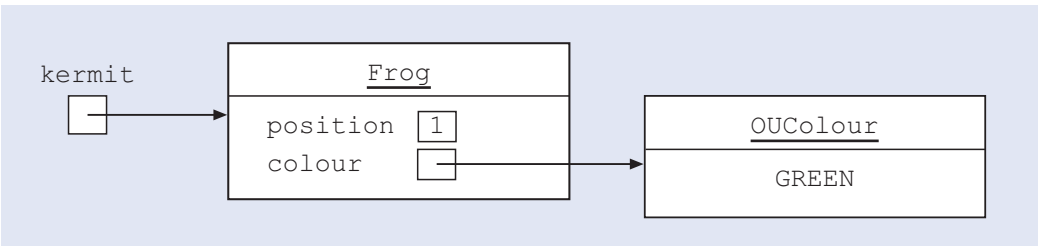

Figure 10 A variable reference diagram showing a new **`Frog`** object, **`kermit`**

SAQ 1

Describe the three steps that take place when the statement

```
kermit = new Frog();
```

is executed.

ANSWER.....

First, the operator `new` causes a new instance of `Frog` to be created.

Second, the constructor `Frog()` causes the new instance of `Frog` to have its `position` attribute initialised to `1` and its `colour` attribute initialised to `OUColour.GREEN`.

Finally, the assignment operator = causes the new, initialised instance of `Frog` to be assigned to the variable `kermit`.

ACTIVITY 2

In this activity, once you have launched BlueJ you will need to open a project from the Project menu. Do not worry too much about this for now – all this does in the context of this unit is to allow you to create instances of classes that have been defined by the course team and that are not part of the standard Java libraries.

If you already have BlueJ running with an open OUWorkspace window, close the OUWorkspace before continuing.

Launch BlueJ and select Open Project from the Project menu. Navigate to the folder called M255 projects. Double-click on the Block1 folder and then on the project named Unit3_Project_1. Opening the project displays a number of icons in the main BlueJ window: one document icon and three rectangular class icons labelled HoverFrog, Frog and Toad. The inheritance relationship between the `HoverFrog` and `Frog` classes is indicated by an open arrowhead (see Figure 11).

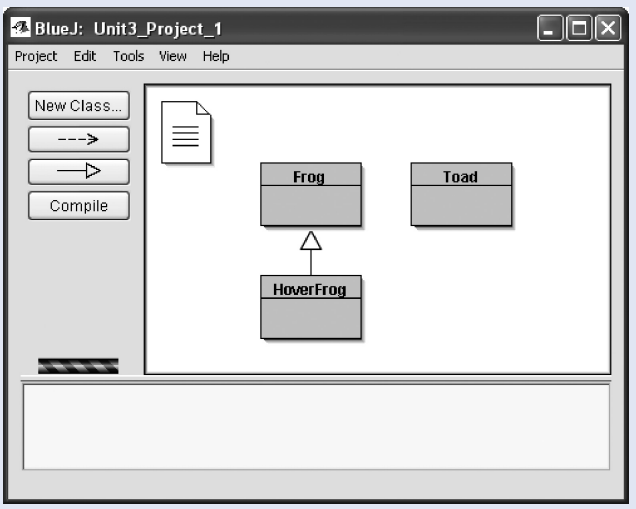

Figure 11 BlueJ project window

The rectangular class icons indicate that in this project the classes `HoverFrog`, `Frog` and `Toad` are available and that you will be able to create instances of them in the OUWorkspace.

From BlueJ's Tools menu select OUWorkspace. The OUWorkspace window that opens has a menu bar with four menus: File, Edit, Action and Graphical Display.

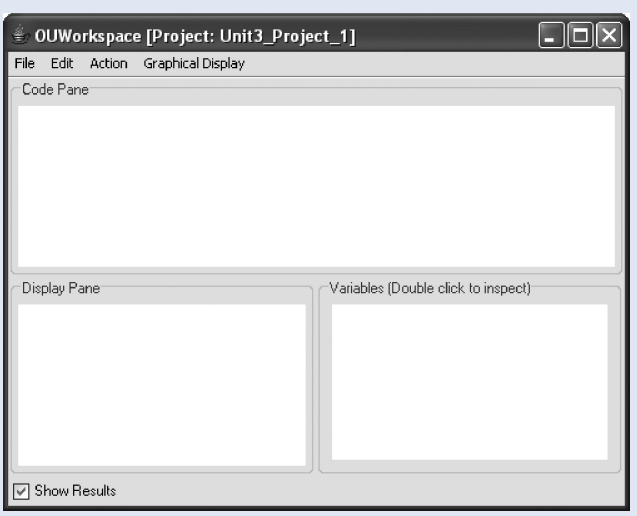

Figure 12 The OUWorkspace window associated with projects containing classes whose objects can be displayed graphically

1 Enter the following statement in the Code Pane and execute it.

```
Frog kermit;
```

Inspect the variable `kermit`.

2 Now enter and execute the statement

```
kermit = new Frog();
```

Inspect the variable `kermit`.

3 Now enter and execute the statements

```
kermit.right();
kermit.right();
kermit.setColour(OUColour.BROWN);
```

Inspect `kermit`.

4 Next enter and execute the statement

```
kermit = new Frog();
```

and inspect `kermit` once again.

5 From the Graphical Display menu of the OUWorkspace choose Open. Once you have done this, a new window labelled Amphibians should appear behind the OUWorkspace. What can you see in this window? Arrange the windows conveniently then, using the OUWorkspace, try sending some messages to `kermit` by typing and executing them in the Code Pane. Try things like:

```
kermit.right();
kermit.setColour(OUColour.PURPLE);
kermit.left();
```

What do you observe in the Amphibians graphics window?

6 In the OUWorkspace declare a variable called `gribbit` and assign to it a new `Frog` object. What do you see in the graphical display?

DISCUSSION OF ACTIVITY 2

1 The variable `kermit` appears in the Variables Pane of the OUWorkspace. Inspecting `kermit` shows that its value is `null`.

2 Inspecting `kermit` now shows that it references an instance of the `Frog` class, that its attribute `position` has the value `1`, and that its attribute `colour` has the value `OUColour.GREEN`. These attributes are implemented as special variables called **instance variables** that are declared by the programmer of the `Frog` class to hold a `Frog` object's state.

3 Inspecting `kermit` now shows that its attribute `position` has the value `3` and that its attribute `colour` has the value `OUColour.BROWN`. In programming terms we say that its *instance variable* `position` has been set to `3` and that its *instance variable* `colour` has been set to `OUColour.BROWN`.

4 Inspecting the object referenced by `kermit` now shows that its instance variable `position` has been set to `1` and that its instance variable `colour` has been set to `OUColour.GREEN`. The reason for this sudden change in `colour` and `position` is that the code

```
kermit = new Frog();
```

has made `kermit` reference a new `Frog` object. The frog that `kermit` referenced initially, in parts 2 and 3, is no longer accessible as no variable now references it. The object that `kermit` referenced in parts 2 and 3 is now available for **garbage collection**.

5 When you opened the Amphibians window from the Graphical Display menu it displayed a graphical representation of the `Frog` object referenced by `kermit`. Sending state changing messages in the OUWorkspace to the `Frog` object referenced by `kermit` resulted in the Amphibians window graphically displaying those state changes.

6 Assignment of a new `Frog` object to `gribbit` is accompanied by the display of a second frog in the Amphibians window.

The Amphibians window that you encountered in Activity 2 models the (possibly empty) collection of variables shown in the Variables Pane of the OUWorkspace. Hence, as soon as you created a new `Frog` object and assigned it to the variable `gribbit`, it was displayed in the Amphibians window. However, the Amphibians window only knows how to display `Frog`, `HoverFrog` and `Toad` objects graphically. It ignores any variables that hold values of primitive types or any variables that reference non-amphibian-like objects. Note also that, since this window only monitors the variables in the Variables Pane of the OUWorkspace, it has no buttons or text boxes. You will learn more about how all this happens in *Unit 4*.

Please note that you can open an Amphibians window from the OUWorkspace to view graphical representations of `Frog`, `HoverFrog` and `Toad` objects in any project that includes those classes.

1.5 Variable names

Variables should be given names (more technically called **identifiers**) that tell a reader what their purpose is. There is not a totally free choice, because Java imposes rules, but these will seldom prevent you from using a suitable name. The rules listed below are a little tighter than what the Java language specifies, but we believe following them will help to make your code more readable.

M255 conventions for identifiers are as follows.

- The first character should be an upper- or lower-case letter (that is, A...Z, a...z).
- Subsequent characters may be an upper- or lower-case letter or a digit (0, 1, 2...9).
- Identifiers are not allowed to have any spaces in them.
- You may not use a **keyword**, which is a word that already has a special meaning in Java. The following table includes all the keywords; these cannot be used as identifiers (please note the American spelling used for the keyword `synchronized`).

`abstract`	`double`	`int`	`strictfp`
`assert`	`else`	`interface`	`super`
`boolean`	`enum`	`long`	`switch`
`break`	`extends`	`native`	`synchronized`
`byte`	`final`	`new`	`this`
`case`	`finally`	`package`	`throw`
`catch`	`float`	`private`	`throws`
`char`	`for`	`protected`	`transient`
`class`	`goto`	`public`	`try`
`const`	`if`	`return`	`void`
`continue`	`implements`	`short`	`volatile`
`default`	`import`	`static`	`while`
`do`	`instanceof`		

- Finally, you cannot use `true`, `false` or `null` as variable names because they are reserved words that are literals used to indicate values.

Note that the above conventions mean that you cannot have an identifier that starts with a digit, and an identifier cannot contain single or double quotes. Although Java does allow two other characters (`$` and `_`) in identifiers there are many characters that must not be used so it is simplest just to use ordinary alphabetic letters.

In M255 we would also encourage you to respect the following style guidelines when choosing identifiers for your variables. Keeping to these guidelines should make your programs much easier to read and (if necessary) debug.

- Although there is no lower or upper limit on the number of characters that can be used in an identifier, we would urge you to avoid very short identifiers, such as `a`, `b`, `x`, `ch` because they are not very informative.
- Identifiers composed entirely of letters are usually the most meaningful to human readers, though it might sometimes be appropriate to use digits in identifiers for a number of similar values, for example `frog1`, `frog2`, `frog3`. Always choose meaningful names; that is, identifiers which give some indication as to the role played by the variable.
- Start your identifiers with lower-case letters. This helps to distinguish them from classes, which start with an upper-case letter. Where an identifier is composed of two or more English words or abbreviations, then we suggest the use of a single upper-case letter to mark the start of each word after the first. We have used this style of identifier (known as a camel-backed identifier) for `myVar` and `yourVar`. Here are a few more examples: `totalCost`, `dateToday` and `myFamilyName`.

SAQ 2

Which of the following are valid variable identifiers in Java? Give a reason in each of the cases where you consider that the text is not an identifier.

(a) `kermit`

(b) `hover frog`

(c) `hoverFrog1`

(d) `my+Frog`

(e) `myAcc`

(f) `3.2`

(g) `"iAmAVariable"`

(h) `MyNumber`

ANSWER.....

(a) `kermit` is a valid variable identifier.

(b) `hover frog` is not valid; it contains a space character.

(c) `hoverFrog1` is a valid variable identifier.

(d) `my+Frog` is not a valid identifier since it contains a `+` character which is not a letter or a digit.

(e) `myAcc` is a valid variable identifier.

(f) `3.2` is not valid; it does not start with a letter; in fact it is a number literal.

(g) `"iAmAVariable"` is not valid; it does not start with a letter; in fact it is a string literal.

(h) `MyNumber` is valid. However, it breaks an almost universal programming convention that variable names should start with a lower-case letter.

In this subsection you have seen that Java deals with two kinds of data – values of primitive data types and instances of classes (objects). You have learnt how to declare variables for both of these data types (value type variables and reference type variables) and how to assign values of the appropriate type to these variables.

You have also learnt that the attributes of objects are implemented by instance variables.

An assignment statement is always of the form

```
aVariable = something;
```

where '`something`' represents a simple literal value or a newly created object. You will see in Section 2 that more complex expressions can be used on the right-hand side of an assignment statement.

Although assignment statements always have the same form (we say they all have the same **syntax**), what an actual assignment statement means (its **semantics**) depends on the type of the variable (and the value that is being assigned to it).

When a value of some primitive data type is assigned to a value type variable then the assignment causes the actual value to be stored in the variable. We call this **value semantics**. However, if the value is an object being assigned to a reference type variable then the object is not stored in the variable; instead the variable holds a reference to where the object is stored. This is called **reference semantics**.

2 Expressions

An **expression** is something that can be evaluated to produce a single value. A helpful way to think of an expression is that it is anything that *could* appear on the right-hand side of an assignment statement. An assignment statement always looks like the following.

```
someVariable = someExpression;
```

The value that an expression evaluates to may be of a primitive data type or it may be an object. **Evaluate** just means 'find the value of'. The simplest examples of expressions in Java are literals such as `17` or `'c'`. The literal `17` evaluates to the `int` value `17`; the character literal `'c'` evaluates to the character `c` (a value of type `char`). An example of an expression that evaluates to an object is `new Frog()`.

A variable that has been assigned a value is also an example of a simple expression. Thus, if `myNumber` has been assigned the value `34`, then, when the expression `myNumber` is evaluated, it returns the value `34`. Similarly if a `Frog` object has been assigned to the variable `kermit`, then, when `kermit` is evaluated, it will return as its value the `Frog` object that it references.

2.1 Expressions involving primitive data types

You can build more complex expressions involving values of primitive data types by combining literals and variables using **operators**. The operators most people are familiar with are the arithmetical operators that are used with numbers. For example,

```
17 + 12
```

is an expression built from the two literals `17` and `12` using the operator `+` (addition). The value it returns when evaluated is `29`. In this expression we call `17` and `12` the **operands** and `+` the operator. We call `+` a **binary operator** because it has two operands.

In the above expression both operands are integers (in fact they are of type `int`); however, you can also use addition with values of type `float` and `double` and, indeed, with a mixture of different types of numbers. The type of the value returned by the evaluation of such expressions will depend on the type of the operands. For example,

`1.7 + 3.4` evaluates to a value of type `double`.

`5 + 1.7` also evaluates to a value of type `double` as the `5` gets converted by the Java interpreter from an `int` to a `double` before the addition is done.

`5 + 17` evaluates to a value of type `int` as both the operands are of type `int`.

The other arithmetic operators are similar. These are – (subtraction), `*` (multiplication) and `/` (division). Examples of expressions using them are the following.

`17 – 5` evaluates to the `int` value `12`.

`7.1 – 5` evaluates to the `double` value `2.1`.

`3 * 2` evaluates to the `int` value `6`.

`1.5 * 2` evaluates to the `double` value `3.0`.

`6 / 2` evaluates to the `int` value `3`.

`7 / 2` evaluates to the `int` value `3` (note this is not `3.5`; when you divide an `int` by an `int` the answer produced is an `int` and any remainder is simply ignored – this is called **integer division**).

`7.0 / 2` evaluates to the `double` value `3.5`.

The only surprising thing in the above examples is the integer division that takes place when `7 / 2` is evaluated. Java provides another operator, the remainder or modulus operator, `%`, which gives the value of the remainder when one integer is divided by another. For example,

`7 % 2` evaluates to the `int` value `1` (the remainder when `7` is divided by `2`),

`17 % 3` evaluates to the `int` value `2`,

`5 % 17` evaluates to the `int` value `5` (do not confuse this with `5 / 17` which gives `0`).

One other arithmetical operator that is sometimes useful is the negation operator. This is just a minus sign (as used for subtraction) but instead of being put between two numbers it is put in front of single number and produces the negative of that number when the expression is evaluated. This is an example of a **unary operator** as it has only one operand. For example,

`- 7` evaluates to the `int` value `-7`,

`- (-7)` evaluates to the `int` value `7`.

All these operators have numbers as operands and when evaluated return numbers as their values. However, there is another group of operators that can have numerical operands but return Boolean values when the expression is evaluated. These are:

`==` (equal to),

`!=` (not equal to),

`<` (less than),

`<=` (less than or equal to),

`>` (greater than),

`>=` (greater than or equal to).

Here are some examples of using them.

Expression	Value
`3 == 3`	`true`
`7.1 == 2.5`	`false`
`7 == 7.0`	`true`
`3 != 3`	`false`
`7.1 != 2.5`	`true`
`2 < 3`	`true`
`2 <= 3`	`true`
`3 <= 3`	`true`
`4 < 3`	`false`
`4 > 3`	`true`

Note that the operators consisting of two symbols, such as `>=`, must not have a space between the symbols. Although not necessary, it is good practice to have space on each side of a binary operator.

The **equality operators** (== and !=) can actually be used with operands of any type. When used with objects as operands, == will answer `true` if both sides evaluate to the same object.

The **relational operators** (<, <=, > and >=) can also be used with operands of type `char`. Thus an expression using < evaluates to `true` if the Unicode code of the character on the left is less than that of the character on the right. For our purposes it is sufficient to know that Unicode codes are arranged in the following order `'0' '1' '2' ... '9' ...` (other characters) `... 'A' 'B' ... 'Z' ...` (other characters) `... 'a' 'b' ... 'z' ...` .

There are three important operators that can be used with Booleans only. These are `&&` (logical and), `||` (logical or), and `!` (not). These **logical operators** work as follows.

Note that `!` is a unary operator (like the negation operator –).

Expression	Value
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`
`false || false`	`false`
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`! true`	`false`
`! false`	`true`

In all the examples above, the operands in the expressions are literals. However, the operands do not need to be literals; an operand can be a variable or indeed any expression that evaluates to a value of an appropriate type.

Before you can use a variable in an expression you need to ensure that it has been declared *and* given a value. In order to ensure that a variable has a value it is often sensible to give it an initial value when it is declared. You can combine declaration and assignment in a single statement. Two examples are

```
int myNumber = 15;
Frog kermit = new Frog();
```

Examples of expressions involving `myNumber` declared and initialised as above are

```
myNumber + 12
myNumber < 17
```

Exercise 3

Assume that `myNumber` and `yourNumber` have been declared and initialised by executing the following statements.

```
int myNumber = 17;
int yourNumber = 5;
```

What are the values of the following expressions?

(a) `myNumber – 15`

(b) `myNumber * yourNumber`

(c) `myNumber / yourNumber`

(d) `yourNumber > myNumber`

(e) `yourNumber != myNumber`

Solution

(a) `2`

(b) `85`

(c) `3`

(d) `false`

(e) `true`

When you use one expression as an operand in another expression, it is sensible to enclose the expression being used as an operand in parentheses (round brackets) unless it is a simple literal or variable. This will avoid any misunderstanding about which part of the expression should be evaluated first. For example, to evaluate the expression

```
(17 – 5) – 2
```

we first evaluate `17 – 5` to get `12`; this is then the value of the first operand for the right-hand subtraction, which becomes `12 – 2` giving the value `10`.

On the other hand, in the expression

```
17 – (5 – 2)
```

it is the `5 – 2` that gets evaluated first giving the value `3` for the second operand of the left-hand subtraction, which becomes `17 – 3` giving the value `14`.

If you simply write `17 – 5 – 2` then it might not be clear which of the evaluations is meant. In fact, there are rules (called **precedence rules**) that determine the order in which things are worked out when there are no parentheses to help determine the order. According to these rules `17 – 5 – 2` would be worked out as if it were `(17 – 5) – 2`.

Expressions involving different operators can also be misinterpreted if parentheses are not used. For example, the correct way to evaluate `17 + 3 * 2` is to treat it as `17 + (3 * 2)` which gives `23` and not as `(17 + 3) * 2` which would give `40`. This is because `*` (multiplication) has a higher precedence than `+` (addition) and so, in the absence of parentheses, multiplication is done before addition.

The following table summarises the order of precedence for the operators you have met so far in this subsection. However, it is always better to use parentheses in a complex expression to make the order of evaluation clear. This helps the readability of the code that you write as well as making it less likely that you will make mistakes with the precedence rules. In the table the operators in any row all have higher precedence than those in the rows below. Apart from the assignment operator, if binary operators of equal precedence appear (without parentheses) in succession in an expression they are evaluated from left to right.

creation	`new`
multiplicative	`* /`
additive	`+ -`
relational	`< > <= >=`
equality	`== !=`
logical AND	`&&`
logical OR	`||`
assignment	`=`

You may be surprised to find assignment (=) included in the above table. In fact although we usually think of assignment as a statement, in other words an instruction to do something, an assignment statement does in fact return a value when it is executed, so it is also an expression. The value it returns is simply the value that it assigns to the variable on the left-hand side of the =.

You can find a precedence table showing all the operators in Java in the *Java Handbook*. You may wish to consult this when additional operators are introduced later in the course.

When an expression is built up using other expressions as its operands we say that the operand expressions are sub-expressions of a **compound expression**. For example, in the compound expression

```
(3 + 2) * 6
```

the left-hand operand to the `*` operator `(3 + 2)` is an example of a **sub-expression**. Such sub-expressions can themselves be made up of further sub-expressions, a little like Russian dolls; for example, in the compound expression

```
4 * (5 + (6 / 2))
```

we say that `(6 / 2)` is **nested** within the sub-expression `(5 + (6 / 2))`.

Such nested expressions should be read from the inside out. So, in this example, `6` is first divided by `2`. Then `5` is added to that result which is finally multiplied by `4`.

More generally, as each nested expression is evaluated, the resulting value is passed up to the next level, so that by the time the outer level is reached the entire compound expression will evaluate to a single value.

There is no limit to the level of nesting; however, it is sensible to limit the amount of nesting since if too much nesting is used it can become very difficult for a human to interpret the expression. As code becomes less readable it is easier to make mistakes.

Exercise 4

In your head, or using pencil and paper, evaluate each of the following expressions.

(a) `3 * (7 + 2)`

(b) `(3 * 7) + 2`

(c) `(7 – (5 – 2)) > (8 / 2)`

(d) `(3 <= 7) && (7 < 9)`

(e) `(6 + 4) / (6 – 4)`

Solution

(a) The expression `(7 + 2)` is evaluated first and then multiplied with `3` to give the value `27`.

(b) The expression `(3 * 7)` is evaluated first and then added to `2` to give `23`.

(c) The expression `(5 – 2)` is evaluated first to give the value `3`. This value is then subtracted from `7` to give a final value for the left-hand operand of `4`. Next `(8 / 2)` is evaluated to give a value for the right-hand operand as `4`. So the operator > has final values of `4` on both the left- and right-hand sides and `(4 > 4)` evaluates to `false`.

(d) The expression `(3 <= 7)` is evaluated first (`3` less than or equal to `7`) which evaluates to `true`. Then `(7 < 9)` is evaluated which also evaluates to `true`. As the final values of the operands of the `&&` operator both evaluate to `true`, the result of the operation is also `true`.

(e) The expression `(6 + 4)` is evaluated first to give the value `10`. Next `(6 – 4)` is evaluated to give the value `2`. The operand `/` then works with the value `10` on the left-hand side and `2` on the right-hand side to return the value `5`.

Exercise 5

Although we recommend that you use parentheses to make the order of evaluation clear, there are some forms of expression where the parentheses are very often omitted. Again, in your head or using pencil and paper, evaluate each of the following.

(a) `3 * 2 + 11`

(b) `3 + 2 * 7`

(c) `25 – 6 – 2`

Solution

(a) As `*` has a higher precedence than `+`, `3 * 2` is evaluated first to give `6` which is then added to `11` to give `17`.

(b) As `*` has a higher precedence than `+`, `2 * 7` is evaluated first to give `14` which is then added to `3` to give `17`.

(c) This expression is evaluated strictly from left to right as both operators have the same precedence (they are both –). Therefore `25 – 6` is evaluated first to give `19`. Then `2` is subtracted from `19` to give `17`.

ACTIVITY 3

Launch BlueJ, if not already open, and then from the Tools menu select OUWorkspace.

Enter, select and execute the following variable **declarations**, one at a time.

```
int anInt;
double aDouble;
boolean aBool;
int myNumber = 17;
int yourNumber = 5;
```

In each of the following statements an expression is evaluated and then assigned to a variable of an appropriate type. When an assignment statement is executed, the value assigned is returned as the answer. Make sure that Show Results is checked; you will then be able to read the value of the expression that was evaluated from its textual representation in the Display Pane. Enter and execute each of the following statements in turn – make sure that the parentheses are in pairs.

```
1  anInt = myNumber – 15;
2  anInt = myNumber * yourNumber;
3  anInt = myNumber / yourNumber;
4  aDouble = myNumber / yourNumber;
5  aBool = yourNumber > myNumber;
6  aBool = yourNumber != myNumber;
7  anInt = myNumber % yourNumber;
8  aDouble = 7 / 2;
9  aDouble = 7.0 / 2;
10 anInt = 3 * (7 + 2);
11 anInt = (3 * 7) + 2;
12 aBool = (7 – (5 – 2)) > (8 / 2);
13 aBool = (3 <= 7) && (7 < 9);
14 anInt = (6 + 4) / (6 – 4);
15 anInt = 3 * (2 + 11);
16 anInt = (3 + 2) * 7;
17 anInt = 25 – (6 – 2);
```

In the next two statements we have deliberately introduced errors. In each case a bracket (parenthesis) is missing. Type in the statements just as you see them and then execute them, one at a time. Can you understand the error messages?

```
18 anInt = 22 / 11) + 4;
19 aBool = (22 / 11 + 4;
```

DISCUSSION OF ACTIVITY 3

1 `anInt` is assigned the value `2`.

2 `anInt` is assigned the value `85`.

3 `anInt` is assigned the value `3`.

4 `myNumber / yourNumber` evaluates to `3` which is converted to `3.0` before being assigned to `aDouble`.

5 `aBool` is assigned the value `false`.

6 `aBool` is assigned the value `true`.

7 `anInt` is assigned the value `2`.

8 `7 / 2` evaluates to `3` (integer division) which is converted to `3.0` before being assigned to `aDouble`.

When you execute

```
aDouble = 7 / 2;
```

the Display Pane will display `3`, yet inspecting `aDouble` will show that it has the value `3.0`. The reason for this is that the Display Pane shows the result of integer division, which is an integer. Java converts this `int` value into a `double` value (`3.0`) before assigning that value to the variable `aDouble`.

9 `aDouble` is assigned the value `3.5`.

10 `anInt` is assigned the value `27`.

11 `anInt` is assigned the value `23`.

12 `aBool` is assigned the value `false`.

13 `aBool` is assigned the value `true`.

14 `anInt` is assigned the value `5`.

15 `anInt` is assigned the value `39`.

16 `anInt` is assigned the value `35`.

17 `anInt` is assigned the value `21`.

18 When the left parenthesis of a pair is omitted, the interpreter reports an error which is written in the Display Pane:

```
Syntax error: column 14. Encountered: )
```

What this error is reporting is that the interpreter has read the line of code from left to right and has come across a right parenthesis at column 14, i.e. the 14th character in the line, without a matching left parenthesis.

19 When the right parenthesis of a pair is omitted, the following error report is written in the Display Pane:

```
Syntax error: column 19. Encountered: ;
```

What this error is reporting is that the interpreter has reached the end of the statement (delimited by the `;` at column 19) and has encountered some error – not very informative!

Finally, it is worth noting that there are two commonly used **postfix** operators `++` and `– –` that can be used with numeric variables. They work as follows.

Suppose `myNumber` is a variable of any number type (`int`, `long`, `float`, `double` or whatever). Then the statement

```
myNumber++;
```

is an instruction to increase the value stored in `myNumber` by `1`.

```
myNumber– –;
```

decreases the value stored in `myNumber` by `1`.

These operators have a higher precedence than the unary minus (negation).

SAQ 3

What is the value of `myNumber` after the following statements have been executed?

```
int myNumber = 17;
myNumber = myNumber – 11;
myNumber++;
```

ANSWER.....

`myNumber` holds the value `7`.

In this course we tend to avoid the use of `++` and `– –` because of the confusion they can cause.

It is tempting to say that the expression `myNumber++` is equivalent to the expression `myNumber + 1` but this is not strictly true. This is because evaluating `myNumber++` returns the value of `myNumber` *before* it has the `1` added to it; whereas the expression `myNumber + 1` returns the value *after* the `1` has been added. So, for example, in the following code:

```
int myNumber, result;
myNumber = 17;
result = myNumber++;
```

`result` is assigned the value 17 *and* `myNumber` is incremented to `18`.

Compare that with this code:

```
int myNumber, result;
myNumber = 17;
result = myNumber + 1;
```

`result` is assigned the value `18` and `myNumber` remains holding the value `17`.

2.2 Expressions involving objects

You have seen how operators are used with values of primitive data types to build up complex expressions from simple ones. However, with objects the only kinds of expression we have mentioned explicitly are ones of the form

```
kermit
```

and

```
new Frog()
```

In the first a variable referencing an object is used; it evaluates to the object that it references. In the second an expression consisting of the operator `new` and the constructor `Frog()` is used as the operand; this expression evaluates to a newly created and initialised `Frog` object. Notice that the `new` operator has a constructor of a class as its operand and returns a value that is an instance of that class.

Unlike values of primitive data types, objects are not normally used with operators. The only common operators generally available for use with objects are the equality operators `==` and `!=`. For example, if `kermit` and `gribbit` are reference type variables then

```
kermit == gribbit
```

will evaluate to `true` if the operands reference or evaluate to the same object; otherwise it will evaluate to `false`. The operator `!=` is the opposite of `==`.

However, there is another type of expression, one that does not involve operators, that is used with objects. As you will recall from *Unit 2*, when a message is sent to an object an answer *may* be returned. For example, if `kermit` is a `Frog` object then the message-send

```
kermit.getColour()
```

returns an answer which is the value of the colour attribute of `kermit`. Similarly if `myAccount` is an instance of `Account` then the message-send

```
myAccount.debit(100)
```

returns an answer which is either `true` or `false` depending on the balance and overdraft limit of the receiver.

On the other hand, message-sends such as

```
kermit.right()
```

and

```
myAccount.credit(100)
```

do not return answers; they simply change the state of the receiver.

SAQ 4

Some message-sends return answers, some change the state of the receiver and some do both. Give an example of message-send that can do both.

ANSWER

`myAccount.debit(100)` will do both, assuming that `myAccount` is an instance of `Account` with a big enough balance to allow `100` to be taken from it. It will return the answer `true` and change the state of `myAccount` by reducing its attribute `balance` by `100`.

A message-send that returns an answer is called a **message expression**.

SAQ 5

Which of the following are message expressions? You can assume that `kermit` references a `Frog` object; `myAccount` references an `Account` object; and `myNumber` references an `int`.

(a) `kermit.left()`

(b) `myAccount.getBalance()`

(c) `myAccount.credit(50)`

(d) `kermit.getPosition()`

(e) `myNumber > myAccount.getBalance()`

(f) `myAccount.debit(kermit.getPosition())`

(g) `myAccount.debit(70) == true`

(h) `kermit.getPosition() + 2`

(i) `myAccount.getBalance() * (kermit.getPosition() + 2)`

ANSWER

(a) No, the message `left()` does not return a message answer.

(b) Yes, the message `getBalance()` returns the balance of an `Account` object.

(c) No, the message `credit()` does not return a message answer.

(d) Yes, the message `getPosition()` returns the position of a `Frog` object.

(e) No, the whole expression is a Boolean expression. However, the right-hand operand of this expression *is* a message expression, which evaluates to an integer.

(f) Yes, the message `debit()` returns either `true` or `false`. In this example the argument to the `debit()` message is also a message expression which evaluates to an integer.

(g) No, the whole expression is a Boolean expression. However, the left-hand operand of this expression *is* a message expression, which evaluates to a Boolean value.

(h) No, the whole expression is an arithmetic expression. However, the left-hand operand to the + operator *is* a message expression which evaluates to an integer.

(i) No, the whole expression is an arithmetic expression. The left-hand operand to the `*` operator is a message expression which evaluates to an integer. The right-hand operand to the `*` operator is a sub-expression, `(kermit.getPosition() + 2)`, which also evaluates to an integer. Inside the sub-expression, `kermit.getPosition()` is a message expression which evaluates to an integer.

A very important point to note is that you cannot tell simply by looking at a message-send whether or not it is a message expression. In other words, unless you know what the message-send 'means' or 'does' you cannot work out whether or not it returns an answer. Often the name might give a clue, but sometimes message names can be misleading. Before you can use a message correctly by writing a message-send you need to know what it does when sent to that type of receiver and whether or not it returns an answer.

It is important to realise that a message-send can be used as a statement on its own whether or not it returns a message answer.

For example, both the following are Java statements:

```
kermit.left();
kermit.getPosition();
```

Both are instructions to send a message in the protocol of `Frog` to an instance of `Frog`. However, it would be unusual to find the second one in a genuine program since it *does nothing* useful. The whole purpose of the message-send `kermit.getPosition()` is to return the value of the receiver's position, but in the above statement the message answer that is returned is simply ignored. So the statement achieves nothing. You would only send the message `getPosition()` to a `Frog` object if you were going to:

- assign the message answer to a suitable variable for use later in the program; for example, you might write:

  ```
  int aNumber = kermit.getPosition();
  ```

  or similarly as part of a more complex expression such as:

  ```
  int frogSum = kermit.getPosition() + gribbit.getPosition();
  ```

- use the message answer as the argument to another message such as:

  ```
  gribbit.setPosition(kermit.getPosition());
  ```

2.3 Strings

Although many classes such as `Frog` and `Account` are developed by programmers for particular applications, there are some classes that are used for creating objects that, like primitive data types, are useful in most if not all applications. There are many sorts of these *general-purpose classes* and they are provided in various libraries as part of the Java language. One important general-purpose class is the `String` class.

Instances of the `String` class model sequences of characters. In other words, an instance of `String` is a collection of characters in a particular order. In Java a string can be represented by enclosing its characters in double quotes. For example, `"cat"` is a string literal representing a `String` object whose characters are `'c'`, `'a'` and `'t'` in that order. Other examples of strings are `"Hello Mum"`, `"Whitehall 1212"` and `"013683795"`.

`String` objects are used for many purposes; for example, as file names, names of products for sale, addresses of businesses, descriptions of holiday resorts. You have already met `String` objects in the `Account` class – the value of the attribute `holder` of an `Account` object is a `String` object. Thus executing the statement

```
myAccount.setHolder("Grendel Barty");
```

makes the `holder` attribute of `myAccount` reference a 13-character `String` object whose characters are `'G' 'r' 'e' 'n' 'd' 'e' 'l' ' ' 'B' 'a' 'r' 't' 'y'`.

SAQ 6

In the `String` object `"Grendel Barty"`, write down the following:

(a) the first character

(b) the fourth character

(c) the eighth character

ANSWER.....

(a) `'G'`

(b) `'n'`

(c) `' '` (a space)

Note the difference between the representations of strings and characters: a `String` literal (or string literal) is a sequence of characters enclosed in double quotes, and a `char` literal (or character literal) is an individual character enclosed in single quotes. A more technical way of expressing this is to say that `String` literals are *delimited* by double quotes and individual `char` literals are *delimited* by single quotes. The term **delimit** is more accurate than *enclosed* because the quotes are used to tell the Java interpreter where a string starts and ends so that the interpreter can determine that the series of characters is not a variable name or a keyword. In a similar way the semicolon (`;`) is a delimiter, as it tells the Java interpreter where a single program statement ends.

Note that the OUWorkspace uses a Java interpreter rather than a compiler. In *Unit 4* you will begin to use BlueJ's Java compiler.

If you wish to include the double quotes character in a string literal, a backslash (\) must precede each double quote, as in `"say \"hello\" to me"`, for example. The reason for the backslash is that if a double quote is to be included in a string, then there must be some way of signalling to the Java interpreter that the end of the string has not been encountered yet and that it should consider the double quote as just a character rather than a delimiter.

`String` objects can respond to many messages; these include `toUpperCase()` and `toLowerCase()`. The message answer from each of these messages (when sent to a `String` object) is a `String` object showing the effect suggested by the name of message. For example, sending the string `"Milton Keynes"` the message `toUpperCase()` results in the message returning a `String` object `"MILTON KEYNES"`.

Another message in the protocol of `String` objects **concatenates** (joins) two `String` objects. To join two `String` objects, the message `concat()` is sent to a `String` object receiver with another `String` object as the message argument. This results in a message answer that is a `String` object made up from the combination, in order, of the characters of the receiver and the argument. For example, `"Java".concat(" programming")` results in a `String` object `"Java programming"`. Since concatenation is used frequently when programming with `String` objects, a shorthand alternative is provided by Java. Instead of using the message `concat()`, the operator `+` (plus) placed between two strings will return a new string that is a concatenation of those two strings. For example, `"Java" + " programming"` will return the string `"Java programming"`.

It is important to remember that strings are not primitive data values. This means that it is possible for two distinct strings to end up with exactly the same characters (i.e. having the same state) without the strings actually being the same object. Thus when an expression such as:

```
string1 == string2
```

(where `string1` and `string2` reference `String` objects) evaluates to `false` it does not necessarily mean that the objects referenced by `string1` and `string2` are 'different' in the sense of having different characters (or the same characters but in a different order).

It simply means that the two variables happen to reference distinct objects. It is quite possible that these distinct objects may contain exactly the same characters, in exactly the same order.

In order to find out if two string objects contain exactly the same characters in the same order (rather than whether two variables reference the same `String` object) there is an `equals()` message that you can use with strings. It is used as follows:

The `equals()` message can be used with objects of any class. However, its meaning varies from class to class. You will learn more about `equals()` later in the course.

```
string1.equals(string2)
```

This message expression returns the value `true` if the receiver (`string1`) and the argument (`string2`) both reference `String` objects that have the same characters in the same order (i.e. if the objects have the same state).

For example, if the following code is executed:

```
String string1 = "cat";
String string2 = "cat";
String string3 = "dog";
```

then you will find that

```
string1.equals(string2)
```

evaluates to `true`, as will

```
string1.equals("cat")
```

while

```
string1.equals(string3)
```

evaluates to `false`.

SAQ 7

Suppose that the following statement has been executed.

```
String fred = "bill";
```

What would be returned by the following message expressions?

(a) `"fred".toUpperCase()`

(b) `fred.toUpperCase()`

ANSWER.....

(a) `"FRED"` i.e. a string whose characters are `'F' 'R' 'E' 'D'` in that order.

(b) `"BILL"`

Note the difference between the literal `"fred"` that is a string with characters `'f' 'r' 'e' 'd'` and the variable `fred` that references a `String` `"bill"` with characters `'b' 'i' 'l' 'l'`.

SAQ 8

Explain why `'h'` and `"h"` are literal representations of different types.

ANSWER.....

The expression `'h'` is a literal that represents the single `char` value `h`, and `"h"` is a literal representing a `String` object with just the single character `'h'` in it. So `'h'` represents a primitive data value not an object and `"h"` references an object that contains a single primitive value in it.

The operator +

The operator + has an unusual property. If just *one* of its operands is a `String` object then the other operand is automatically converted to a `String` object and the + is interpreted as meaning concatenation. How the conversion of objects to strings is done will be examined in Section 4. For primitive types, such as numbers, Booleans and characters, the string version is just a string made up of the characters of the literal that represents the value. For example:

`"cat" + 's'` evaluates to `"cats"` (`'s'` is converted to `"s"`)

`"The answer is " + 17` evaluates to `"The answer is 17"` (`17` is converted to `"17"`)

`10 + "66"` evaluates to `"1066"` (the number `10` is converted to the string `"10"`)

`"17 > 8 is " + (17 > 8)` evaluates to the string `"17 > 8 is true"`
(`17 > 8` is in parentheses and therefore is evaluated first to give `true`; then `true` is converted to `"true"` before being concatenated with `"17 > 8 is "`)

`10 + 66` evaluates to the integer value `76` (there are no string operands so no conversion takes place; here the + results in ordinary integer addition).

SAQ 9

What types and values do the following expressions evaluate to?

(a) `"The answer to Life, the Universe and Everything is " + 42`

(b) `76 + " Trombones led the big parade"`

(c) `56 + 23`

(d) `"56" + 23`

ANSWER.....

(a) The string `"The answer to Life, the Universe and Everything is 42"`

(b) The string `"76 Trombones led the big parade"`

(c) The integer `79`

(d) The string `"5623"`

ACTIVITY 4

Remember you can view graphical representations of `Frog` objects in the Amphibians window (accessible via the Graphical Display menu).

Launch BlueJ and select Open Project from the Project menu. Navigate to the folder containing your M255 projects. Find the project named Unit3_Project_1 and open it. Then select OUWorkspace from the Tools menu. Enter, select and execute the following statements, one at a time.

```
int anInt;
String aString;
OUColour aColour;
char aChar;
Frog kermit = new Frog();
kermit.right();
kermit.setColour(OUColour.RED);
```

Make sure Show Results is checked. Enter, select and execute each of the following statements, noting the value that is returned in each case.

```
1  anInt = kermit.getPosition();
2  aColour = kermit.getColour();
3  aString = "Milk".concat("Wood");
4  aString = "Milk ".concat("Wood");
5  aString = "Milk".concat(" Wood");
6  aString = "Under Milk ".concat("Wood");
7  aString = "Milk " + "Wood";
8  aString = "It is " + "raining.";
9  aString = "He wouldn't say \"boo\" to a goose";
10 aString = "The End of the Line".toUpperCase();
11 String name = "Engelbert";
12 aString = name.toLowerCase();
13 aString = name;
```

You can, if you like, save the contents of the Code Pane to a file, by selecting Save from the OUWorkspace's File menu and giving the file a sensible name such as Activity4.txt. The saved code can be reloaded into the Code Pane in a later session by selecting Open from the File menu and navigating to the saved file.

DISCUSSION OF ACTIVITY 4

1 `anInt` is assigned the `int` value `2`.
2 `aColour` is assigned the object `OUColour.RED`.
3 `aString` is assigned the `String` object `"MilkWood"`.
4 `aString` is assigned the `String` object `"Milk Wood"`.
5 `aString` is assigned the `String` object `"Milk Wood"`.
6 `aString` is assigned the `String` object `"Under Milk Wood"`.
7 `aString` is assigned the `String` object `"Milk Wood"`.
8 `aString` is assigned the `String` object `"It is raining."`.
9 `aString` is assigned the `String` object `"He wouldn't say "boo" to a goose"`.
10 `aString` is assigned the `String` object `"THE END OF THE LINE"`.
11 `name` is assigned the `String` object `"Engelbert"`.
12 `aString` is assigned the `String` object `"engelbert"`.
13 `aString` is assigned the `String` object `"Engelbert"`.

3 Variables, typing and assignment

So far in this unit we have introduced the concepts of data types, variables and assignment. You have seen how variables must be declared as being of some particular type, and how values are assigned to them using assignment statements (which themselves might be involve either simple or complex expressions).

We now need to look at the restrictions that declaring a variable as a given data type places on the values that can be assigned to that variable (and what happens if we attempt to break these restrictions).

3.1 Assigning values of primitive types to variables

You might think that if a variable has been declared to be of a certain type, then only values of that type can be assigned to it. However, as you have seen, for operators like + it is not necessary for both operands to be of the same type; for example, you can use + to add two numbers of different types (an `int` and a `double`), and you can also use + to concatenate a `String` object with a value of any type. Similarly with assignment, you will find that it is sometimes possible to assign a value to a variable that is of a different type to the variable's declared type. In the next few activities you will investigate what is possible.

ACTIVITY 5

Launch BlueJ and from the Tools menu select OUWorkspace.

1 Enter and execute both these statements. Note what happens.

```
int anInt;
anInt = 17.5;
```

2 Now enter and execute the following, then inspect `aFloat`.

```
float aFloat;
aFloat = 17;
```

3 Enter and execute the following. Can you explain what happens?

```
anInt = aFloat;
```

4 Enter and execute the following. Can you explain what happens?

```
aFloat = 17.0;
```

5 Enter and execute the following, then inspect `aFloat`.

```
aFloat = 17.0F;
```

6 Enter and execute the following, then inspect `aDouble`.

```
double aDouble;
aDouble = 17;
```

7 Enter and execute the following, then inspect `aDouble`.

```
aDouble = 17.5;
```

You can save the contents of the Code Pane to a file by choosing Save from the File menu, and giving the file a sensible name such as Activity5.txt.

DISCUSSION OF ACTIVITY 5

1 An error message is written in the Display Pane:

```
Semantic error: line 2. Variable: Can't assign double to int
```

The message is telling you that `17.5` cannot be assigned to the variable `anInt` because `17.5` is a `double` and `anInt` has been declared as an `int`.

Note that when you highlight and execute more than one line of code, error messages from the OUWorkspace tell you which line the error was detected in.

2 The `int` value `17` is converted to a `float` by Java and assigned to `aFloat`.

3 An error message is written in the Display Pane:

```
Semantic error: Variable assignment: anInt: Can't assign
float to int
```

The message is telling you that a value of type `float` cannot be assigned to a variable (`anInt`) that has been declared as an `int`.

4 An error message is written in the Display Pane:

```
Semantic error: Variable assignment: aFloat: Can't assign
double to float
```

The message is telling you that a value of type `double` cannot be assigned to a variable (`aFloat`) that has been declared as a `float`. Remember (from Section 1) that the default type of a decimal literal is type `double`. To specify a literal `float` you must append the letter 'F', for example `17.0F`.

5 This time the assignment works because `17.0F` is a literal of type `float`. The inspector confirms that `aFloat` is `17.0`.

6 The `int` value `17` is converted to a `double` by Java and assigned to `aDouble`, which is now `17.0`.

7 The default type of a literal decimal is type `double`. Therefore `17.5` is of type `double` and can be assigned to the variable `aDouble`. This is confirmed by the inspector.

Activity 5 shows that it is sometimes possible to assign a value of one numerical type to a variable of a different numerical type. Thus you can assign an `int` value to a `float` variable and a `float` value to a `double` variable; but you cannot assign a `double` value to a `float` variable (even if it (`17.0`) is equal to an integer, `17`).

If we arrange the most commonly used numerical types in the order

```
double, float, long, int
```

then it is permissible to assign a value of one type to a variable of any type to its left (as well as to a variable of the same type) but you cannot assign a value to a variable of a type to its right (even if the actual value seems to allow it). Thus although `17.0` equals `17` (an integer) you cannot store `17.0` in a variable of type `int`.

Note also that when you assign an `int` to a `float` the value is stored as a `float` and cannot afterwards be assigned back to an `int`.

However, it is sometimes possible to force a value of one type to be converted to a value of another type, by using what is known as a **cast**. A cast comprises a type name enclosed in parentheses. Thus you can use the cast `(int)` to 'convert' a non-integer number into a value of type `int`. For example:

```
double aDouble;
int anInt;
aDouble = 17.6;
anInt = (int) aDouble;
```

The cast `(int)` coverts the value `17.6` stored in `aDouble` into an `int` by simply throwing away everything after the decimal point. Thus it is the integer `17` that gets assigned to the variable `anInt`.

ACTIVITY 6

If the OUWorkspace is still open from Activity 5 skip the following three declarations and start the investigation, otherwise launch BlueJ and open the OUWorkspace. Then enter and execute the following declarations:

```
int anInt;
float aFloat;
double aDouble;
```

Enter and execute the following statements in the OUWorkspace. If no error is reported, inspect the relevant variable; in all cases try to explain what has happened.

1 `anInt = (int) 17.8;`

2 `anInt = (int) 17000000000.6;`

3 `aFloat = (float) 17.8;`

4 `aFloat = (float) 17E40;`

5 `aDouble = 17E40;`

6 `aDouble = 17 / 5;`

7 `anInt = 17;`
`aDouble = anInt / 5;`

8 `aFloat = 17;`
`aDouble = aFloat / 5;`

9 `double anotherDouble = 17;`
`aDouble = anotherDouble / 5;`

10 `aDouble = (double) anInt / 5;`

11 `aDouble = (float) anInt / 5;`

12 `aDouble = 17;`
`aFloat = aDouble / 5;`

13 `aDouble = 17;`
`aFloat = (float) (aDouble / 5);`

You can save the contents of the Code Pane to a file, giving the file a sensible name such as Activity6.txt.

For the remainder of the course we will not necessarily prompt you to save the contents of the Code Pane after an activity (but please use the facility when you judge it to be helpful).

DISCUSSION OF ACTIVITY 6

In this discussion when we write, for example, '`anInt` is `17`' we mean that inspecting `anInt` shows it has the value `17`.

1 `anInt` is `17`.

2 `anInt` is `2147483647`. This is the biggest `int` that is possible – it is not much use to us as an approximation to `17000000000.6`. It would be better to use a variable of type `long` if you wanted to store this value approximately as an integer.

3 `aFloat` is `17.8`. This is just the same as you would get from `aFloat = 17.8F`, so it is a bit pointless.

4 `aFloat` is `Infinity`. `17E40` is too big to be stored as a `float`. (Recall that `E40` means move the point 40 places to the right which in this case means following the `17` with 40 zeros.) As a result the value is stored as `Infinity`.

5 `aDouble` is `1.7E41`. Note that when Java calculates a very large result it shows the result using exponential notation with the number before the `E` lying between 1 and 10. In this case as you have assigned the value to a variable of type `double`, it is stored accurately; `double` can cope with much larger numbers than `float`.

6 `aDouble` is `3.0`. Although `aDouble` is a `double` the expression `17 / 5` involves integer division so it evaluates to `3`. This integer value may be assigned to `aDouble` (of type `double`) without a cast. It is automatically converted to the `double 3.0` before being assigned to `aDouble`.

7 `aDouble` is `3.0`. Because `anInt` is an `int`, the explanation is just the same as for part 6.

8 You should get something similar to `aDouble` is `3.4000000953674316`. Since `aFloat` is a `float` the expression `aFloat / 5` is evaluated using `float`s and the answer is a `float`. However, because of rounding errors, when this is converted to a `double` there are some inaccuracies after the seventh decimal place, and thus your number is unlikely to be identical to the one shown above. (Values of type `float` are not stored as accurately as `double`s.)

9 `aDouble` is `3.4`. This time the calculation is done using doubles and so the rounding error mentioned in part 8 is removed.

10 `aDouble` is `3.4`. The cast `(double)` has higher precedence than division `/` and so the `int` stored in `anInt` is converted to a `double` before the division is done. Thus the answer is the same as in part 9. It would be clearer to write this as `aDouble = ((double) anInt) / 5;`.

11 `aDouble` is `3.4000000953674316`. (You may have quite different figures at the end.) Notice that the rounding errors may not be the same as in part 8.

12 This gives an error message since `aDouble / 5` gives a value of type `double` and this cannot be assigned directly to `aFloat`, which is of type `float`.

13 `aFloat` is `3.4`. The answer to `aDouble / 5` is converted to a `float` by the cast `(float)`.

3.2 Assigning objects to reference type variables

The previous subsection involved only values of primitive data types being assigned to value type variables. In this subsection you will use the `Frog` and `HoverFrog` classes to investigate how assignment is affected when the variables are reference type variables, and the values being assigned to them are objects.

ACTIVITY 7

Launch BlueJ first if necessary.

From the Project menu of BlueJ select Open Project, then navigate to and open the project named Unit3_Project_1. The main BlueJ window now displays three rectangles, one labelled Frog, another labelled HoverFrog and another labelled Toad (see Figure 11). These rectangles tell you that in this project the classes `Frog`, `HoverFrog` and `Toad` are available and that you will be able to create instances of these classes in the OUWorkspace.

From the Tools menu select OUWorkspace. Then enter and execute the following declarations.

```
Frog kermit;
Frog gribbit;
HoverFrog happy;
HoverFrog bouncy;
```

You might find it helpful at this stage to open the OUWorkspace's Graphical Display menu, as this will allow you to see graphical representations of the `Frog` and `HoverFrog` objects that will be created during this activity.

Enter each of the following statements in turn, select it and execute it. After each execution inspect the objects referenced by each of the variables `kermit`, `gribbit`, `happy` and `bouncy` and note any changes. If executing the statement results in an error message being written in the Display Pane, make a note of the problem.

(a) `kermit = new Frog();`

(b) `happy = new HoverFrog();`

(c) `kermit.right();`

(d) `kermit.right();`

(e) `happy.setColour(OUColour.RED);`

(f) `gribbit = kermit;`

(g) `gribbit.setColour(OUColour.BLUE);`

(h) `kermit.right();`

(i) `bouncy = kermit;`

(j) `kermit = happy;`

(k) `happy.up();`

(l) `kermit.up();`

DISCUSSION OF ACTIVITY 7

This discussion simply summarises the changes of state that take place. The next subsection will give a fuller explanation using variable reference diagrams.

The inspector shows the colour, rather than the name of the colour.

(a) `kermit` references a `Frog` with `position` set to `1` and `colour` set to `OUColour.GREEN`. All the other variables hold `null`.

(b) `happy` references a `HoverFrog` with `position` set to `1`, `colour` set to `OUColour.GREEN`, and `height` set to `0`.

(c) The `Frog` object referenced by `kermit` now has `position` set to `2`.

(d) The `Frog` object referenced by `kermit` now has `position` set to `3`.

(e) The `HoverFrog` object referenced by `happy` now has `colour` set to `OUColour.RED`.

(f) `gribbit` now references the same `Frog` object as `kermit`. It has `position` set to `3` and `colour` set to `OUColour.GREEN`.

(g) The `Frog` referenced by both `kermit` and `gribbit` now has `colour` set to `OUColour.BLUE`.

(h) The `Frog` referenced by both `kermit` and `gribbit` now has `position` set to `4`.

(i) This produces an error message. `kermit` references a `Frog` object, and you cannot assign a `Frog` object to a variable of type `HoverFrog`, so `bouncy` still holds `null`.

(j) You can assign a `HoverFrog` object (the one referenced by `happy`) to a variable of type `Frog`. So `kermit = happy` works and results in both `kermit` and `happy` referencing the `HoverFrog` that `happy` was referencing; it has `position` set to `1`, `colour` set to `OUColour.RED`, and `height` set to `0`.

(k) The `HoverFrog` referenced by both `happy` and `kermit` now has `height` set to `1`.

(l) This should have produced an error, but because of a peculiarity of the OUWorkspace the code has worked and the `HoverFrog` object referenced by both `happy` and `kermit` now has `height` set to `2`.

The OUWorkspace's Java interpreter is erroneously using *the class of the object* referenced by the variable `kermit` (which is `HoverFrog`) to determine what messages are valid, rather than *the declared type* of the variable `kermit` (which is `Frog`). The BlueJ Java compiler (which you will encounter in Unit 4) correctly determines what messages are valid based on the type of the variable, therefore the compiler would reject the statement with the error message:

```
cannot find symbol - method up()
```

as the method `up()` is not part of the protocol of `Frog`.

It is hoped that this peculiarity of the OUWorkspace can be fixed in a future release.

The last activity was designed to show the following facts about assignment and reference type variables.

- If a variable is of type `T` (where `T` is a class such as `Frog`), then an instance of a subclass of `T` (such as `HoverFrog`) can be assigned to the variable.
- An instance of a class `T` (such as `Frog`) cannot be assigned to a variable whose type is a subclass of `T` (such as `HoverFrog`).
- If a variable is of type `T` (where `T` is a class such as `Frog`), then only messages in the protocol of `T` can be sent to the object referenced by that variable. Even if the variable actually references an instance of a subclass of `T` (such as `HoverFrog`), the Java interpreter will reject any attempt to send messages in the protocol of the subclass that are not in the protocol of `T`.

Note that if a class `U` is a subclass of the class `T` or of any of `T`'s subclasses then we say that `U` is a **subtype** of the type `T`. The important point to take from the above is:

- An object whose type is a subtype of the declared type of a variable can be assigned to that variable.

3.3 Visualising references to objects

It can often be useful to use variable reference diagrams to visualise what is going on when dealing with reference type variables and assignment. We now repeat the statements that were used in Activity 7 and after each show the state of the relevant objects using variable reference diagrams. Note that in these diagrams we have added a reminder of the type of each variable below its box.

After the variables have been declared, they all initially hold the value `null`.

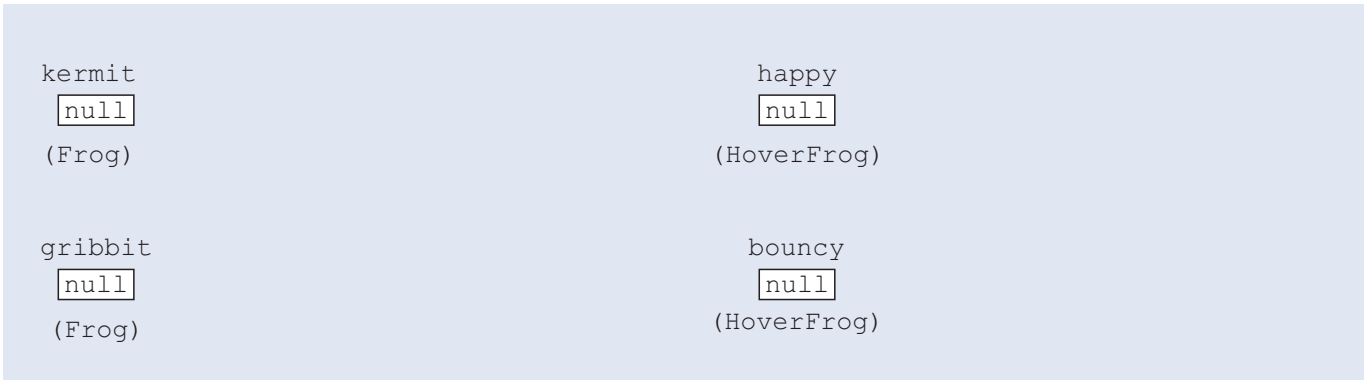

Figure 13 All variables hold **`null`**

(a) `kermit = new Frog();`

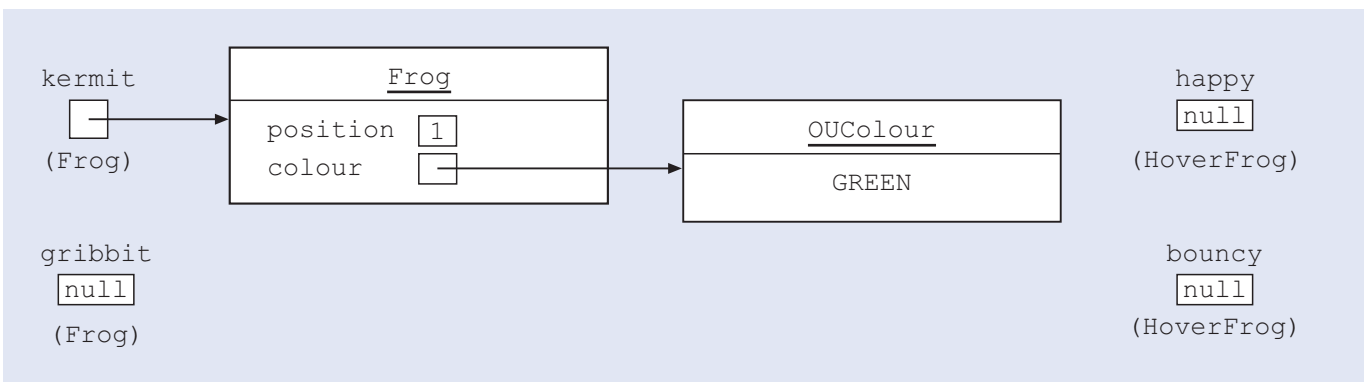

Figure 14 **`kermit`** references a **`Frog`** object

Note that in Figure 14, the instance variable `colour` of the `Frog` object is shown correctly referencing an instance of `OUColour`, and the instance variable `position` directly holding the value `1`. To prevent our variable reference diagrams in this discussion of Activity 7 becoming too cluttered, we will simplify the representation of `OUColour` objects to a box with the name of a colour in it.

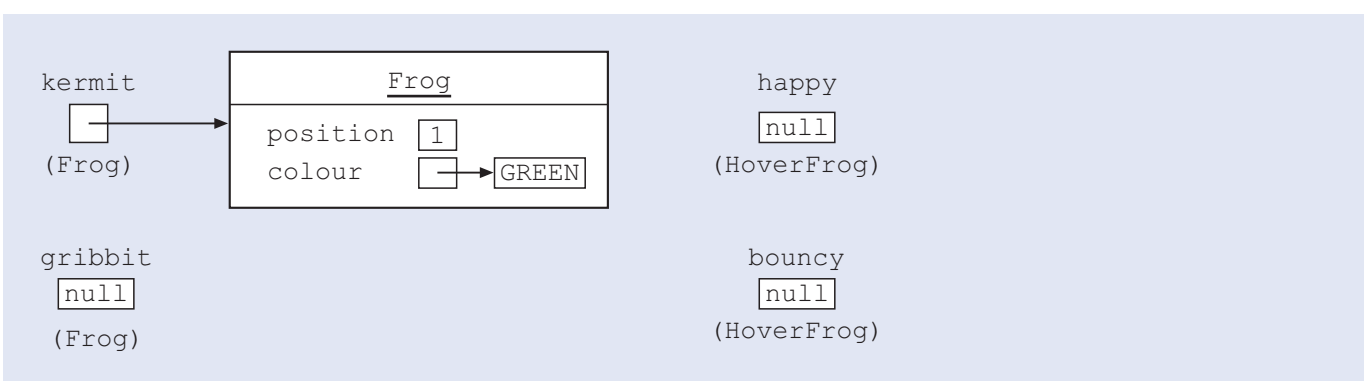

Figure 15 **`kermit`** references a **`Frog`** object (simplified)

`kermit` now holds a reference to a `Frog` object. This is shown by the arrow between the block of memory labelled by the variable `kermit` and the `Frog` object – put another way, `kermit` now holds the address of the `Frog` object in memory.

(b) `happy = new HoverFrog();`

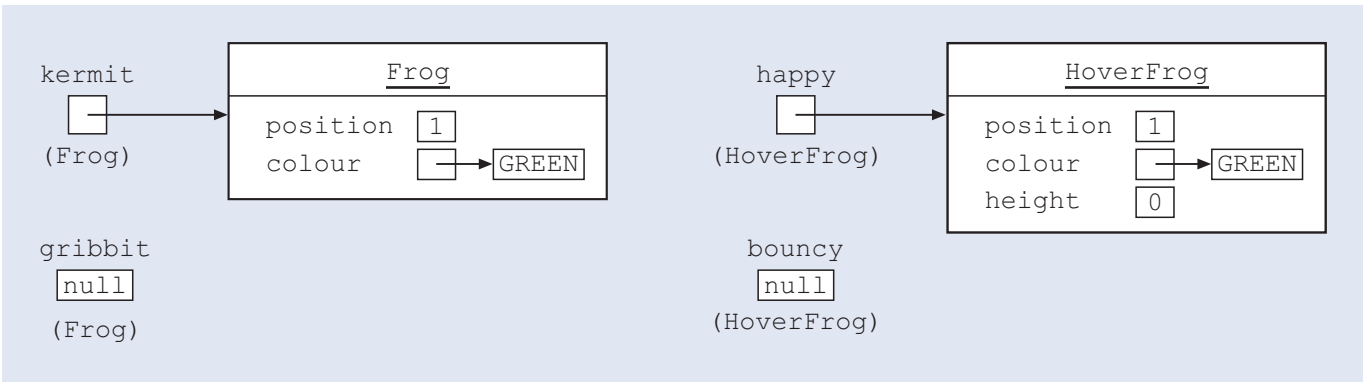

Figure 16 Now **happy** references a **HoverFrog** object

(c) `kermit.right();`

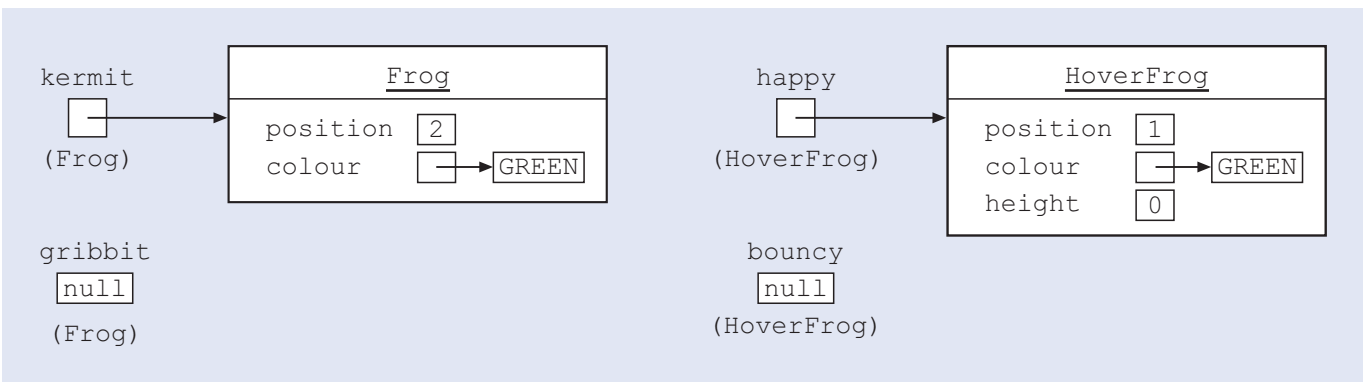

Figure 17 The **position** of the **Frog** object referenced by **kermit** has now been set to **2**

(d) `kermit.right();`

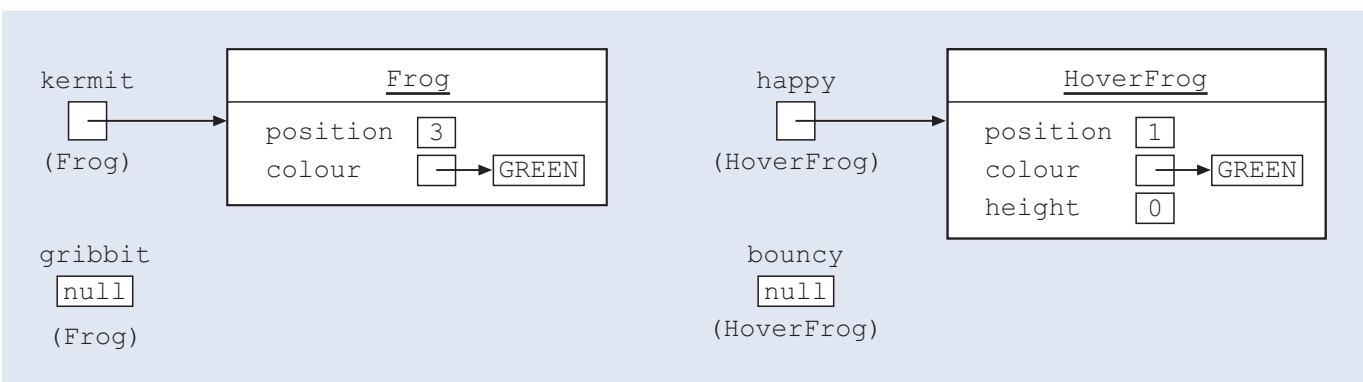

Figure 18 The **position** of the **Frog** object referenced by **kermit** has now been set to **3**

(e) `happy.setColour(OUColour.RED);`

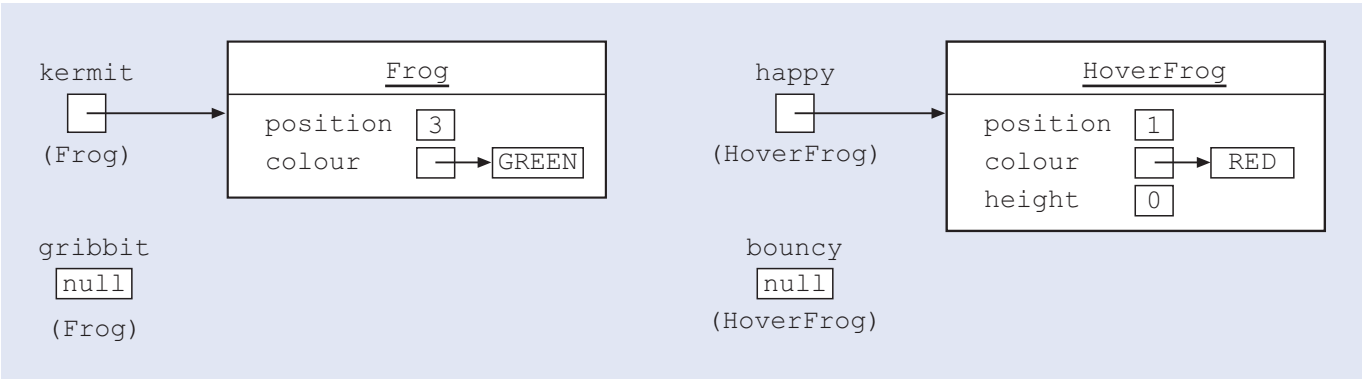

Figure 19 The **colour** of the **HoverFrog** object referenced by **happy** has now been set to **OUColour.RED**

(f) `gribbit = kermit;`

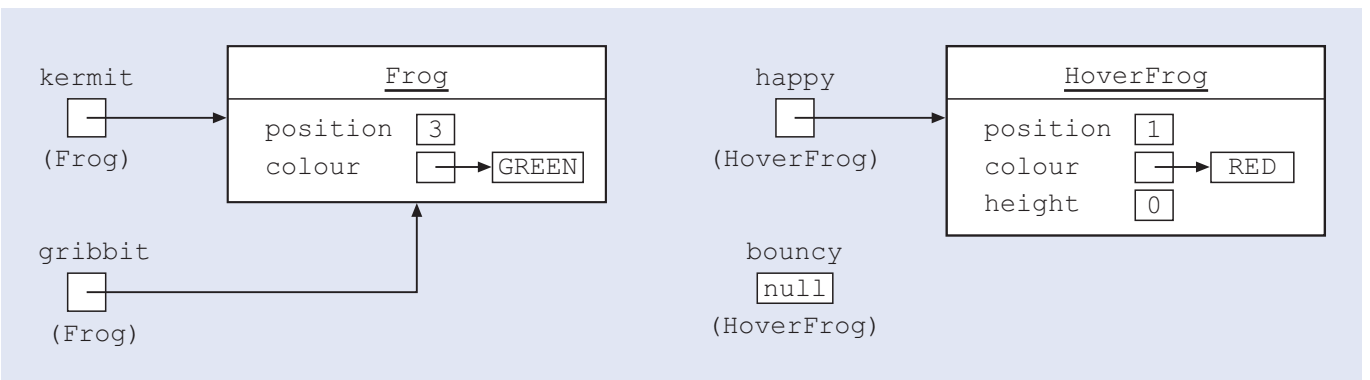

Figure 20 Now **gribbit** and **kermit** reference the same **Frog** object; **bouncy** still holds the value **null**

(g) `gribbit.setColour(OUColour.BLUE);`

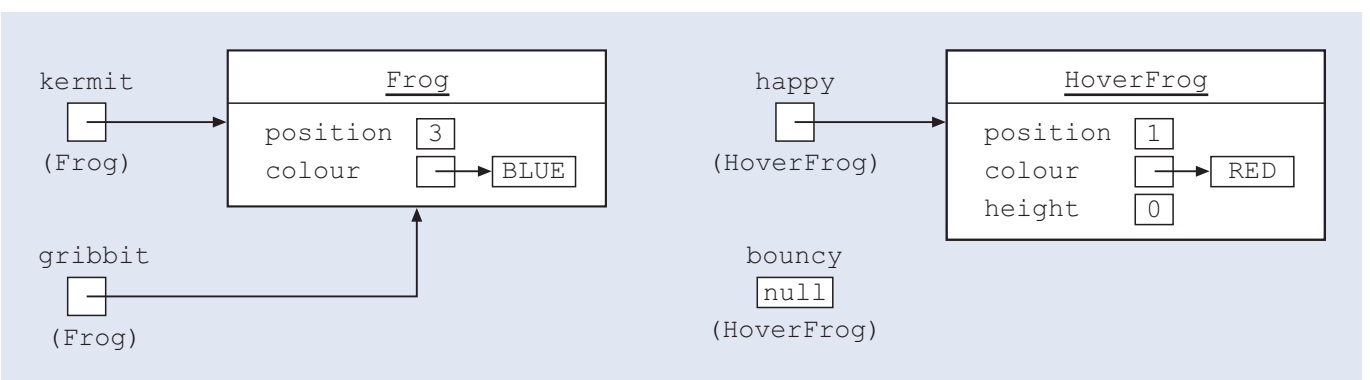

Figure 21 The **colour** of the **Frog** object referenced by both **gribbit** and **kermit** has now been set to **OUColour.BLUE**

(h) `kermit.right();`

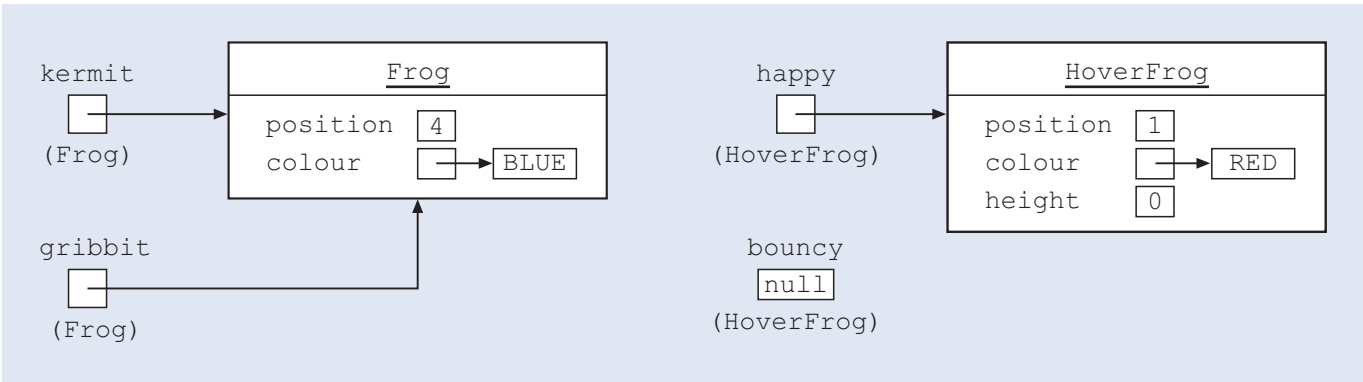

Figure 22 The **position** of the **Frog** object referenced by both **gribbit** and **kermit** has now been set to **4**

(i) `bouncy = kermit;`

This produces an error message as you cannot assign a `Frog` object to a variable of type `HoverFrog`.

(j) `kermit = happy;`

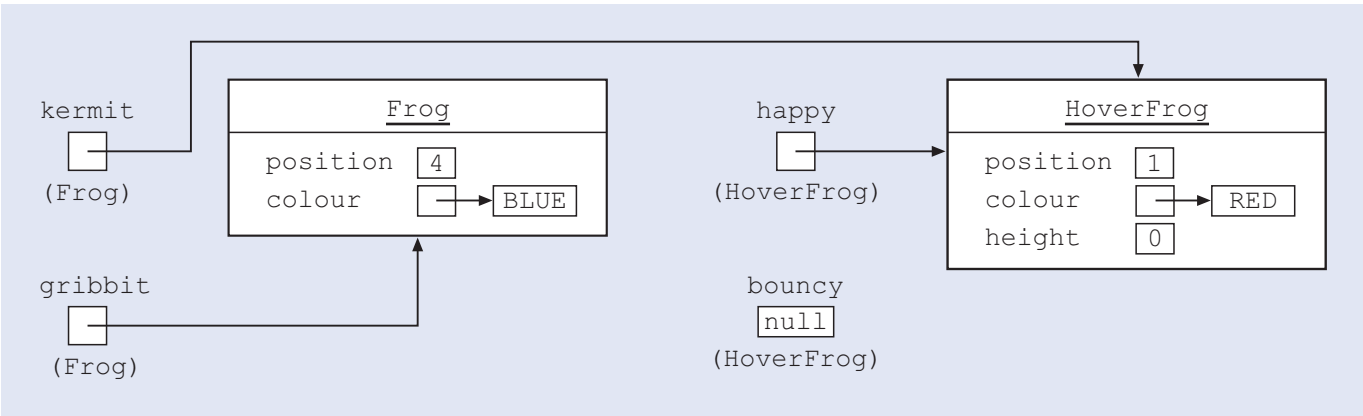

Figure 23 **kermit** and **happy** now reference the same **HoverFrog** object; **gribbit** references a **Frog** object and **bouncy** still holds the value **null**

(k) `happy.up();`

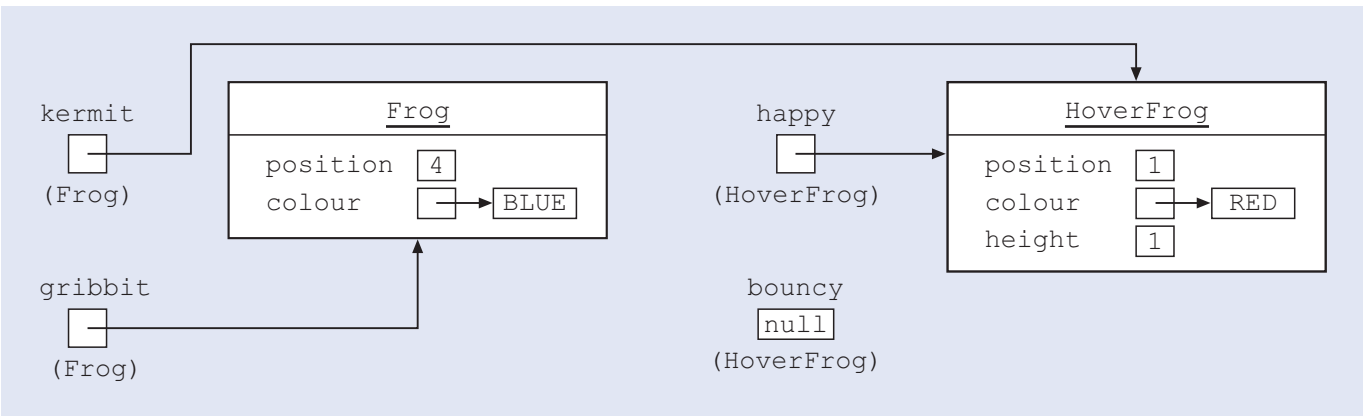

Figure 24 The **height** of the **HoverFrog** object referenced by both **kermit** and **happy** has now been set to **1**. **bouncy** still holds the value **null**

(l) `kermit.up();`

This produces an error as, although `kermit` now references a `HoverFrog` object, `kermit` has been declared as being of type `Frog`. The interpreter therefore refuses to send a message to `kermit` that is not in the protocol of `Frog` objects.

In the above sequence of statements note that although there were four variables involved only two objects were created.

It is important to distinguish assigning an object to a variable from sending a message to the object that a variable references. Consider the following example.

Example 1

Suppose the following statements have been executed (starting from a cleared OUWorkspace).

```
Frog kermit = new Frog();
Frog gribbit;
kermit.right();
kermit.setColour(OUColour.RED);
gribbit = kermit;
```

The situation is shown in Figure 25.

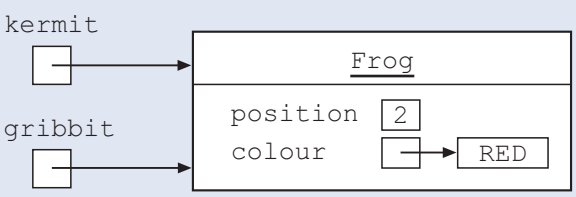

Figure 25 **kermit** and **gribbit** are referencing the same **Frog** object

If we now send a message using either `kermit` or `gribbit` as the receiver, the *same* object will be the actual receiver of the message. Thus

```
gribbit.right();
```

produces the situation shown in Figure 26.

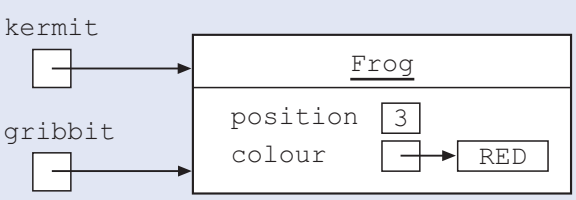

Figure 26 **kermit** and **gribbit** are still referencing the same **Frog** object

The message–send

```
kermit.setColour(OUColour.BLUE);
```

results in Figure 27.

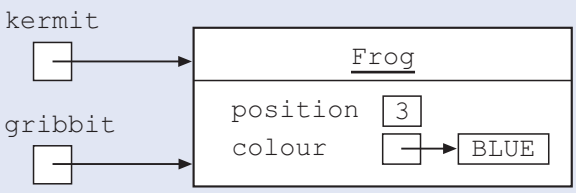

Figure 27 **kermit** and **gribbit** are now blue

On the other hand, if we assign a different object to one of the variables, then the other one is unaffected. Thus

```
kermit = new Frog();
```

results in Figure 28.

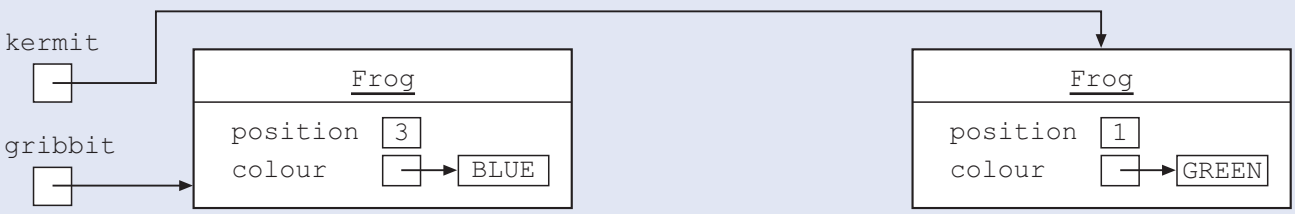

Figure 28 **kermit** and **gribbit** are now referencing different **Frog** objects

Another technique for visualising references to objects is to make use of the Amphibians graphical display window, which you will do in the next activity.

ACTIVITY 8

Launch BlueJ if necessary. From the Project menu navigate to and open the project called Unit3_Project_1. From BlueJ's Tools menu open the OUWorkspace. From the Graphical Display menu of the OUWorkspace choose Open to bring up the Amphibians window.

1 In the OUWorkspace declare two variables for `Frog` objects by typing and executing the following:

```
Frog kermit;
Frog perseus;
```

Next create and assign a new `Frog` object to the variable `kermit` by executing the code

```
kermit = new Frog();
```

2 Now make the object referenced by `kermit` turn brown and move it to the third stone to the right by executing

```
kermit.right();
kermit.right();
kermit.setColour(OUColour.BROWN);
```

Next assign the `Frog` object referenced by `kermit` to the variable `perseus` by executing the following statement:

```
perseus = kermit;
```

Now execute the following statements one by one and observe what happens in the Amphibians window.

```
perseus.setPosition(1);
perseus.right();
kermit.right();
perseus.right();
kermit.right();
```

What is the resulting behaviour in the Amphibians window? What is the resulting state (use the inspector to inspect `kermit` and `perseus`)? How do you account for what you observe?

3 Now execute the code

```
perseus = new Frog();
```

What do you observe in the Amphibians window?

4 Now execute

```
kermit.right();
```

What do you observe in the Amphibians window?

5 Now execute

```
perseus.setColour(OUColour.BLUE);
```

What do you observe in the Amphibians window?

6 Finally, execute the following assignment statement:

```
perseus = kermit;
```

What do you observe in the Amphibians window?

DISCUSSION OF ACTIVITY 8

1 The code `kermit = new Frog();` created a new instance of the class `Frog`, which you could observe in the Amphibians window.

2 The statement `perseus = kermit;` assigned the address of the `Frog` object referenced by `kermit` to the variable `perseus`. Now both variables, `perseus` and `kermit`, reference the very same `Frog` object. It is important to realise that this assignment did not alter in any way the fact that the variable `kermit` still references the same `Frog` object. You can have as many variables as you like referencing a given object.

Because `perseus` and `kermit` reference the same object there is still only one `Frog` object visible in the Amphibians window.

To demonstrate further that the assignment statement really did make the variable named `perseus` reference the same `Frog` object that is referenced by `kermit` you should have observed that sending messages using either the variable `perseus` or the variable `kermit` made the same single `Frog` object change state (as observed in the Amphibians window).

3 `perseus = new Frog();`
After the execution of this assignment statement, the variable `perseus` references a new instance of `Frog`, which is displayed in the Amphibians window – there are now two frogs in the Amphibians window.

4 `kermit.right();`
The variable `kermit` evidently still references the original `Frog` object. This is demonstrated by the fact that when you send messages to `kermit` the original `Frog` object responds.

5 `perseus.colour(OUColour.BLUE);`
This demonstrates that the variable `perseus` really does reference a new `Frog` object, as the second `Frog` object in the Amphibians window turns blue.

6 `perseus = kermit;`
As soon as this assignment statement is evaluated, the second `Frog` object disappears from the Amphibians window. The variable `perseus` once again references the same object as `kermit`, and the `Frog` object that was referenced by `perseus` prior to the assignment statement is no longer available as it is no longer referenced by a variable. At some later stage the Java Virtual Machine will remove it from memory. The removal of unreachable objects is called garbage collection.

Assignment to value type variables

It is important to distinguish the way assignment works for reference type variables (reference semantics) from what happens with value type variables (value semantics).

Example 2

Executing the following statements:

```
int myNumber = 17;
int yourNumber = 30;
```

produces the situation shown in Figure 29.

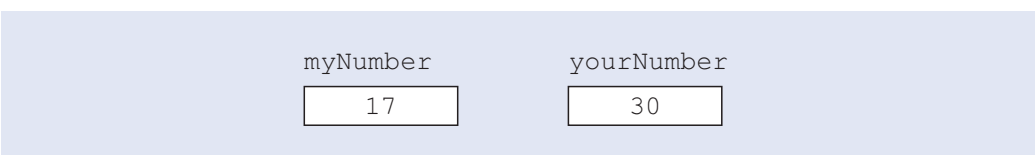

Figure 29 **myNumber** and **yourNumber** have different values

If you now execute

```
yourNumber = myNumber;
```

then a *copy* of the value stored in `myNumber` is put into `yourNumber`, overwriting the value 30.

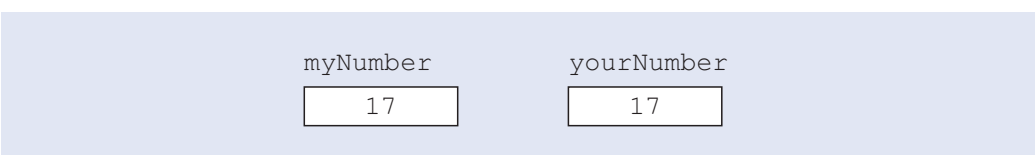

Figure 30 **myNumber** and **yourNumber** have the same values

The two variables are completely independent of each other even although they *contain* the same value. Compare this with the situation after `gribbit = kermit;` was executed in Example 1, where the two variables ended up *referencing* the same object.

If you now execute the statement

```
myNumber = myNumber + 5;
```

you get the situation shown in Figure 31.

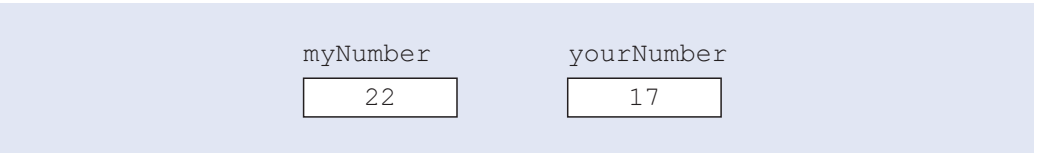

Figure 31 **myNumber** and **yourNumber** have different values

To understand this, consider how `myNumber = myNumber + 5;` is executed. First the expression `myNumber + 5` is evaluated; since `myNumber` currently holds the value `17` this gives a value of `22`. This value `22` is now assigned to (stored in) the variable `myNumber`. Since the variable `yourNumber` is quite separate from `myNumber`, `yourNumber` is not affected in any way.

4 Textual representations of objects and primitive values

Throughout this unit you have seen how the value obtained from evaluating an expression is displayed in the Display Pane of the OUWorkspace. When these values have literal value equivalents, like numbers (e.g. `1`), Booleans (e.g. `true`) and strings (e.g. `"Fred"`), this is fairly straightforward, as Java knows how to display (as text) values that have literal equivalents. But what about complex objects like `Frog` objects? If a message returns a `Frog` object as its answer, how could that be displayed in the Display Pane? How would Java know what to display? This section explains how to give an object a textual representation, which can be used to describe that object in the Display Pane.

4.1 The message `toString()`

In Java every object responds to a message called `toString()`. On receiving such a message an object will return as the message answer a textual representation of itself.

The nature of the textual representation varies with the class of the object. The textual representation may indicate only the class of the object and its address in memory. This is the case for `Account` objects; for example, the message expression

```
myAccount.toString();
```

might return the string shown in Figure 32.

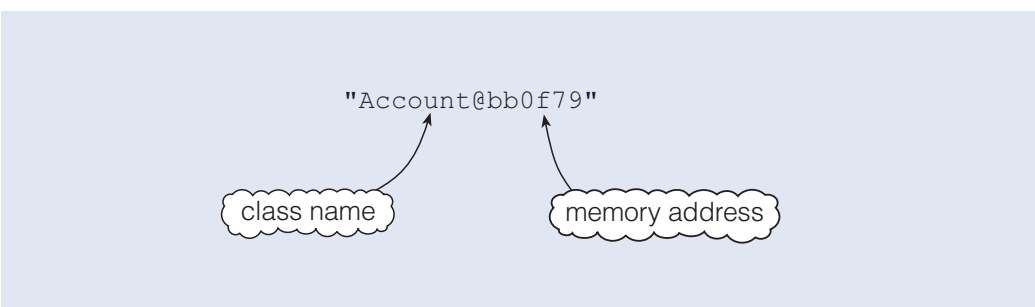

Figure 32 A textual representation of an **`Account`** object

This is the default representation, and is used if the programmer has not provided for a more meaningful descriptive text in designing the class.

However, the textual representations of objects of other classes may provide information on the state of the object as well as its class name. For example, more informative text has been built into the design of the `Frog` class and replaces the default. Thus the message expression

```
kermit.toString();
```

might return (depending on the state of the `Frog` object referenced by `kermit`) the much more informative textual representation:

```
"An instance of class Frog: position 1, colour OUColour.GREEN"
```

Note, however, that `Frog` objects cannot be distinguished by their textual representations, as all frogs with the same state would have the same textual

representation. `Account` objects can, however, be distinguished, as the textual representation of an `Account` object includes its unique address in memory.

In the next activity, message answers are revisited and you will examine the way in which the message `toString()` is used to obtain the textual representation of an object.

ACTIVITY 9

Every object has a textual representation associated with it. You will now investigate how textual representations are generated in the Display Pane by using the message `toString()`, which is a message to which all objects respond. If it is not already open, launch BlueJ and, from the Project menu, open Unit3_Project_2. Opening the project displays four rectangles, labelled Frog, HoverFrog, Toad and Account, in the main BlueJ window. These rectangles tell you that this project has those classes available and that you will be able to create instances of those classes in the OUWorkspace.

Open the OUWorkspace and make sure that Show Results is checked.

1 If the OUWorkspace does not have a variable named `kermit`, declare one now and assign a new `Frog` object to it as follows:

```
Frog kermit = new Frog();
```

Next type and execute the following statement to declare a `String` variable:

```
String str;
```

Next type and execute:

```
str = kermit.toString();
```

What has been written in the Display Pane?

From the list of variables double-click on `str` to inspect it. What does the inspector show?

2 Now type, select and execute a statement that consists of just the variable name followed by a semicolon, for example:

```
kermit;
```

Note the textual representation of the result shown in the Display Pane.

Why do you think that the results shown in the Display Pane are similar for steps 1 and 2?

3 To confirm your ideas, repeat steps 1 and 2 with an `Account` object.

DISCUSSION OF ACTIVITY 9

If the Show Results box is checked, the Display Pane always shows the value returned from evaluating the last expression in a statement (or series of statements).

1 When you execute `str = kermit.toString();` the text `"An instance of class Frog: position 1, colour OUColour.GREEN"` is displayed in the Display Pane. This string is the value to which the message expression `kermit.toString()` evaluates.

Inspecting the variable `str` shows that the result of evaluating `kermit.toString()` has been assigned to the variable `str`.

2 The result of evaluating just `kermit;` is the `Frog` object referenced by `kermit`, yet the text `An instance of class Frog: position 1, colour OUColour.GREEN` (note that there are no quotes around it) is displayed in the Display Pane! This text is the contents of the string which is the value of `kermit.toString()`. How did this happen?

The reason for this is that if the last expression in a statement (or series of statements) evaluates to an *object*, the OUWorkspace automatically sends the message `toString()` to that object so that the textual representation of that object can be written in the Display Pane.

3 You should have found that if you declare an `Account` object, for example `myAccount`, and it is returned as the value of an expression, the text displayed in the Display Pane is the contents of the string `myAccount.toString();`.

To conclude, the Display Pane of the OUWorkspace displays the textual representation of the value computed from the final evaluation in a statement or series of statements. Just as the icons that represent the `Frog`, `Toad` and `HoverFrog` objects in the Amphibians window and the various microworlds encountered in *Units 1* and *2* are not the objects themselves but are graphical representations, the results of evaluating expressions that are shown in the Display Pane of the OUWorkspace are textual representations of objects or primitive values. Note that displaying the results of evaluating expressions is not a feature of Java but a feature of the OUWorkspace, a programming tool written in Java that we have designed to let you test snippets of Java code.

5 Summary

Variables and types

- Data is stored (or information about where to find it) in named memory locations. These named memory locations are called variables because they can contain different values, at different times, during the execution of a program. Variable names must follow the rules for Java identifiers.
- A type is a set of values and the set of operations which are permitted on those values. Java has two very different kinds of types – primitive data types and reference types.
- Value type variables *hold* the values of primitive types but reference type variables *reference* objects.
- Variables are declared to be of a particular type and can only be assigned values of a compatible type.
- A variable can only reference a single object at any one time, but many variables can reference the same object. Variable reference diagrams can be useful for visualising a series of assignments.
- An object does not know what variables it is referenced by. Objects with no references are garbage collected (destroyed) by the Java Virtual Machine.
- Primitive data types can be considered to be general-purpose data types from which objects are built.
- In Java, any sequence of characters enclosed in double quotes is an instance of the class `String`. The individual characters in a string are values of the primitive type `char`.

Expressions

- Expressions perform the work of a Java program. Among other things, expressions are used to compute values and to help control the execution flow of a program. The job of an expression is two fold: to perform the computation indicated by the elements of the expression and to return some value that is the result of the computation.
- Expressions are built using operands and operators.
- Sub-expressions can be combined using parentheses to create compound expressions that evaluate to a single value.
- An assignment statement assigns the value of the expression on the right-hand side of the assignment operator (=) to the variable on the left-hand side of the operator.
- A message that returns a message answer is called a message expression.

Objects

- Objects are created by using the operator `new` and a constructor.
- Every object has an associated textual description. The default description gives the class and memory address of the object. A programmer may choose to provide a fuller description for instances of some classes.
- All objects understand the message `toString()` which returns the textual representation of the receiver.

BlueJ and the OUWorkspace

- The BlueJ system encompasses a programming language, a library of classes and a development environment. All commercial systems contain these three parts, with the major variation being in the environment.
- The OUWorkspace is a tool (integrated into BlueJ) that enables the writing and testing of code in a quick and convenient way.
- The Display Pane has two functions. First, it is where Java writes any error messages if you make a mistake in any code you are testing in the Code Pane. The second function of the Display Pane is to display the value of the final expression evaluated in a statement (or series of statements) in the Code Pane. Both these features are aspects of providing information to the programmer.

LEARNING OUTCOMES

After studying this unit you should be able to:

- explain the meaning of *type*;
- explain the difference between the values of primitive data types and the values of classes (objects);
- explain what a variable is and how it is declared;
- explain the difference between a value type variable and a reference type variable;
- understand that once a variable has been declared for a certain type, only values of a compatible type can be assigned to that variable;
- explain what is meant by assignment;
- draw variable reference diagrams and answer questions about the practical effects of assignments;
- understand that objects not referenced by a variable are automatically garbage collected, i.e. destroyed;
- launch BlueJ and open a specified project and then open the OUWorkspace;
- use the OUWorkspace to execute statements which evaluate expressions and assign the results to variables;
- create instances of a class by using the `new` operator and a constructor;
- inspect an object;
- recognise and correct mistakes by reading error reports in the Display Pane of the OUWorkspace;
- use the message `toString()` to get the textual description of an object;
- evaluate arithmetic, string, Boolean and message expressions both simple and compound;
- understand and control (using parentheses) the precedence of evaluation in Java;
- use sub-expressions and nesting to build compound expressions so that the evaluated value from one sub-expression can be used as the operand for another sub-expression.

Glossary

assign See **assignment**.

assignment When using objects, assignment is the process which results in the variable on the left-hand side of the assignment operator referencing the object returned by the expression on the right-hand side (this is called assignment using **reference semantics**). When using values of **primitive data types**, assignment is the process that results in the variable on the left-hand side containing a copy of the *value* returned by the right-hand side (this is referred to as assignment using **value semantics**).

assignment statement A **statement** that tells Java to make a **variable** reference a particular object or to hold a particular primitive value (*see* **assignment**).

compound expression An **expression** built up using other sub-expressions; for example, the following is a compound expression: `(3 + 2) * (6 - 3)`

concatenation The joining of two strings. In Java the string concatenation operator is + (the plus sign). For example, `"Milton "` + `"Keynes"` evaluates to `"Milton Keynes"`.

constructor A special type of message used to initialise a newly created object.

expression Code that evaluates to a single value. Expressions are formed from variables, operators and messages.

garbage collection The process of destroying objects, which have become unreachable because they are no longer referenced by variables, in order to reclaim their space in memory. In certain programming languages, including Java, this process is automatic.

identifier The name of a **variable**.

instance variable A **variable** that is common to all the instances of a class but whose value is specific to each instance. Each instance variable either contains a reference to an object or contains a value of some primitive type. For example, `Frog` objects have the instance variables `colour` and `position`. The values of the instance variables of a particular object represent the state of that object.

integrated development environment (IDE) A software tool that supports the construction, compilation and execution of a program. BlueJ is an example of an IDE that supports the development of programs in Java and includes libraries of classes and facilities for debugging and program design.

literal A comprehensible textual representation of a primitive value or object. For example, `'X'` is a `char` literal, `4.237` is a `double` literal and `"hello there!"` is a `String` literal.

message expression A message-send which evaluates to a value, i.e. the message returns an answer.

`new` An operator used to create an object – used in conjunction with a **constructor**.

primitive data type A set of values together with operations that can be performed on them. The primitive data types in Java provide a set of basic building blocks from which all the more complex types of data can be built. There are three categories of primitive data type: numbers, characters and Booleans.

reference semantics The situation whereby a **variable** holds the *address* of an **object**, rather than a *value*. A **reference type variable** on the left-hand side of an assignment statement always ends up referring to the object on the right-hand side *(*cf. **value semantics**).

reference type variable A variable declared to reference an object of the declared or compatible type.

statement A statement represents a single instruction for the compiler or interpreter to translate into **bytecode**. In Java a statement must always end with a semicolon.

syntax The structure of **statements** in a given language.

value semantics The situation in which a **variable** holds a value, of some **primitive data type**. A **value type variable** on the left-hand side of an assignment statement always ends up holding a copy of the value on the right-hand side (cf. **reference semantics**).

value type variable A **variable** declared to hold a value of the declared or compatible primitive type.

variable A named 'chunk' or block of the computer's memory which can hold either a value of some primitive type or the address (reference) of an object.

variable reference diagram A diagram showing a reference type variable pointing to a representation of an object with the current values of its attributes.

Index