M255 Unit 5
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Dialogue boxes, selection and iteration

Unit 5

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.

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 5497 1

2.1

CONTENTS

Introduction 5

1 Dialogue boxes 6

1.1 Class methods 7

1.2 The class `OUDialog` 7

1.3 The class method `OUDialog.alert()` 8

1.4 The class method `OUDialog.confirm()` 12

1.5 The class method `OUDialog.request()` 14

1.6 Request dialogue boxes with an initial answer 16

1.7 Converting between data types 17

2 Conditions and selection 21

2.1 `if` statements with simple conditions 21

2.2 Comparing values 24

2.3 Handling the output of the Cancel button 31

3 Boolean expressions 33

3.1 Simple comparisons 33

3.2 Boolean operators 34

4 Iteration 42

4.1 Examples of loops 42

4.2 `for` loops 44

4.3 `while` loops 52

5 Summary 60

Glossary 62

Index 63

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 everyday life you often come across statements such as:

> If it is raining then we go to work by car; otherwise we walk.

This is called **conditional selection**. It says that *if* something is true *then* we shall do one thing *else* we shall do another thing.

Similar choices occur frequently in computer programs. For instance, you might be prompted for confirmation that you wish to proceed with an action, such as deleting a file. Your response – whether you click the OK button or not – will determine what the program does. Or, as another example, you might wish to withdraw money from a cash machine. The banking system must then decide whether to meet your request or not, based on whether you have enough funds in the account.

Conditional selection is a fundamental structure in all programming. In software, decisions like those described above are represented by conditions which work out to Boolean values, either `true` or `false`. Which course of action is followed – i.e. what code is executed – will depend on this value.

A second fundamental programming structure is repetition, also called **iteration** or looping. Like conditional selection, repetition is also common in everyday life. We might want to print off 50 copies of a document. Or we might want to go on serving customers while there are still people waiting in a queue.

Examples of iteration from programming are scanning the characters in a string one by one to determine if they are vowels, or repeatedly reading the next line of text from a file until the end of the file is reached.

In a program, iteration might involve incrementing a counter each time an action is repeated and testing the counter to see if the required number of iterations has been completed. Or it could involve checking, before the next iteration, that some condition is still true.

In this unit we also introduce class methods and dialogue boxes. Dialogue boxes are a common and familiar way for a program to interact with a user. It is with dialogue boxes that we shall start as you will be using them throughout this unit.

1 Dialogue boxes

So far in the course, although you have been able to display output, you have not been able to write Java code which allows true two-way interaction with the user.

The class `OUDialog` (note the American spelling) provides facilities both for giving information to the user (output) and for obtaining information from the user (input) using dialogue boxes.

Java provides a much more general form of dialogue box, but the set of dialogue boxes in the class `OUDialog` is designed to be easier to use than the more general form.

Dialogue boxes are not the only way of obtaining input from users – or of displaying output, of course – but they are popular with users and, as you will see, quite friendly for the programmer to work with.

Dialogue boxes come in many forms: some just ask the user to click on a button, while others may require the user to enter something from the keyboard or make a choice between yes and no.

You may have seen Java dialogue boxes already. They are used to report some kinds of error that occur when you use the BlueJ environment. Here is what we got when we pressed the New Class... button in a BlueJ project window and then deliberately left the class name blank.

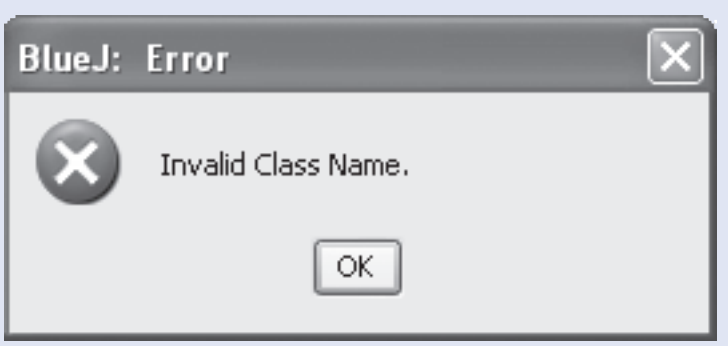

Figure 1 A dialogue box produced in BlueJ

The `OUDialog` class we shall be using provides three different flavours of dialogue box:

alert – displays a message

confirm – displays a question and asks the user to click Yes or No

request – asks the user to type something into the dialogue box's input box.

Each of these types of dialogue box is **modal**, that is, one which will not allow you to interact with another part of the program or system until you have responded to it by clicking one of the buttons presented.

These three kinds of dialogue box are found in other programming languages, not just Java.

This section begins with a look at class methods. We then look in detail at how the class methods in `OUDialog` are used.

1.1 Class methods

Classes, as well as defining instance methods, can also define **class methods**. These are methods that can be executed irrespective of whether any instances of the class have been created. Statements that result in class methods being executed look very like statements that result in an instance method being executed. For example:

(a) `SomeClass myVar = new SomeClass();`
`myVar.doSomething();`

ultimately results in the *instance* method `doSomething()` being executed.

(b) `SomeClass.doSomething();`

results in the *class* method `doSomething()` being executed.

The difference between the two is as follows. In (a), `myVar` is a variable that references an object and `myVar.doSomething()` is a message-send that ultimately results in a `doSomething()` method being executed. When a message-send is compiled, the compiler produces bytecode that at run-time passes the receiver and the message to the JVM (Java Virtual Machine). It instructs the JVM to find out the class of the receiver and to search for a method that matches the message's signature (starting from the receiver's own class and then searching up the inheritance hierarchy until the method is found). When the method is found, the JVM then invokes that method, with the receiver, so that within the method the receiver can be referenced by the pseudo variable `this`.

In (b), `SomeClass.doSomething()` is not a message-send, even though it may look like one. `SomeClass` is not a variable that references an object; it is the name of a class. Only objects can be sent messages and, *in Java*, classes are not objects. In Java, when we are talking about class methods, classes are more like traditional function or procedure libraries. Code such as `SomeClass.doSomething()` is taken by the compiler and translated into much simpler bytecode that instructs the JVM at run-time to go directly to the class `SomeClass` and invoke the method `doSomething()`.

Hence in this course we describe code such as `SomeClass.doSomething()` in terms such as 'the method `doSomething()` is invoked on the class `SomeClass`', whereas we describe code such as `myVar.doSomething()` in terms such as 'the message `doSomething()` is sent to the object referenced by `myVar`'.

You will learn much more about class methods in *Unit 7*. For now all you need to remember is that to invoke a class method you simply write the class name followed by the class method using the now-familiar dot notation. Class methods are quite common in object-oriented languages, such as Java. One of the things they are often used for is providing general utilities, and the dialogue boxes provided by the class `OUDialog` are a good example.

1.2 The class `OUDialog`

`OUDialog` has class methods with the following headers:

```
static void alert(String prompt)
static boolean confirm(String prompt)
static String request(String prompt)
static String request(String prompt, String initialAnswer)
```

Note the Java keyword `static`, which indicates that a method is a **class method**. Class methods are also called static methods.

To display an alert dialogue box you would write a statement like this:

```
OUDialog.alert("Dialogues are very useful!");
```

This *looks* like a message-send, but is not because `OUDialog` is a class, not an object.

When this is executed an alert box pops up to display some information and stays there until the user clicks the OK button. Other `OUDialog` class methods expect the user to enter a response, which is used as the method's return value. The return value can be stored in a variable for use subsequently.

Here is an example showing an `OUDialog` class method being invoked and the answer being assigned to a variable. Apart from the fact that in this case the method is a class method, the pattern is just like any other example where a return value is assigned to a variable for subsequent use.

```
boolean answer;
answer = OUDialog.confirm("Is this a class message?");
```

We shall explain all this in more detail below. The examples in this subsection are just to show how straightforward invoking class methods is in practice.

Note that if we use the term 'method' on its own we normally mean instance method. When we mean class method we shall always be specific.

1.3 The class method `OUDialog.alert()`

This class method in `OUDialog` has the signature `alert(String)`. It takes a `String` argument, which is displayed in a modal dialogue box. As its name implies, it is often used when alerting the user to something they need to be aware of. For example, executing

```
OUDialog.alert("Remember to save your file before exiting");
```

results in the dialogue box shown in Figure 2.

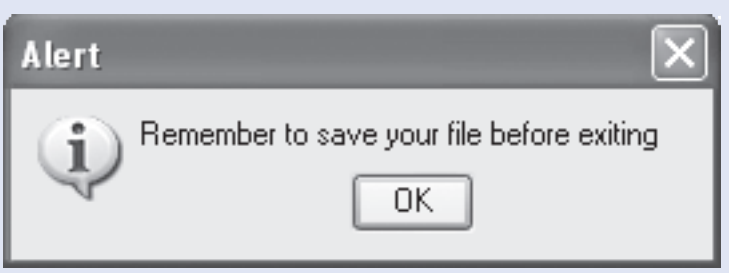

Figure 2 A dialogue box produced using **`OUDialog.alert()`**

To dismiss the dialogue box, you must click the OK button. This method requires nothing else from the user; it does not return a value, and no further action takes place after OK has been clicked.

SAQ 1

Write down a statement that when executed displays the following dialogue box.

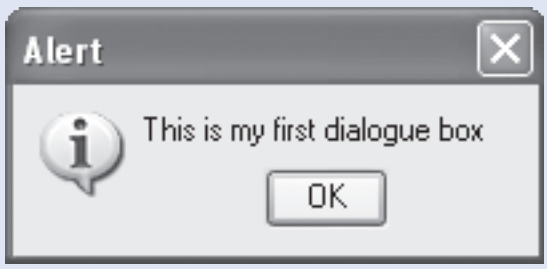

Figure 3 Another dialogue box produced using `OUDialog.alert()`

ANSWER

```
OUDialog.alert("This is my first dialogue box");
```

Note that, although the method `alert()` may often be used to issue a warning, it can be used to display any text whatsoever that the programmer chooses.

Exercise 1

Explain how each line of the following sequence of statements is executed, and sketch the dialogue box that results from execution of the complete sequence. (Line numbers are provided for reference.)

This exercise makes use of the message `toUpperCase()` which, when sent to a string, returns a string consisting of the original string all in upper case.

```
String name;                                    //1
name = "Patrick";                               //2
name = name.toUpperCase();                      //3
OUDialog.alert("The name was " + name);         //4
```

Solution

Line 1 declares a variable of type `String` whose identifier is `name`.

Line 2 makes the variable `name` reference the `String` object `"Patrick"`.

Line 3 makes the variable `name` reference the new `String` object returned when the object referenced by `name` (`"Patrick"`) is sent the message `toUppercase()`.

Line 4 uses the method `alert()` of the `OUDialog` class with the argument resulting from concatenating `"The name was "` with the object referenced by `name`, which from line 3 is now `"PATRICK"`.

The dialogue box produced looks like this.

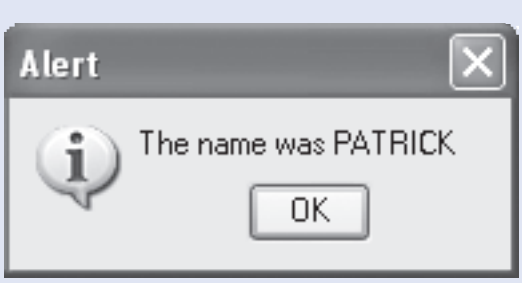

Figure 4 Dialogue box displaying code in upper case

If an empty string had been assigned to `name` in line 2 of the code in Exercise 1, the statement sequence (given below) would have resulted in the dialogue box shown in Figure 5.

```
String name;
name = "";
name = name.toUpperCase();
OUDialog.alert("The name was " + name);
```

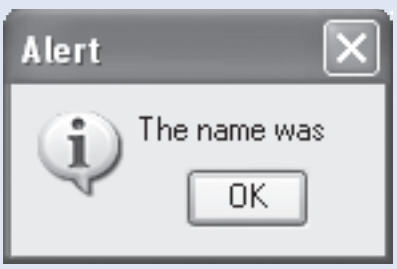

Figure 5 The dialogue box that would result if **name** were assigned an empty string

There is no project for this activity.

ACTIVITY 1

Launch BlueJ and open the OUWorkspace.

In the workspace practise using `OUDialog.alert()` by executing statements with a variety of string arguments. Some suggestions are given below but obviously you can make up your own.

```
OUDialog.alert("My name is Bond. James Bond.");
OUDialog.alert("Cannot accept a number larger than 100");
OUDialog.alert("stop".toUpperCase() + "!");
OUDialog.alert("500");
```

DISCUSSION OF ACTIVITY 1

Note that you were able to display the `String` object `"500"`. Remember that strings can contain a variety of different characters – not just alphabetic characters – and that a string consisting of numeric characters, such as `"500"`, is not the same as the number `500`.

Remember too that a 'warning' produced by the `OUDialog` method `alert()` is just displayed text and has no other effect. For example, it does not actually stop users entering whatever they please. If you wanted to restrict the range of numbers a user could input, you would have to write other code to enforce this.

ACTIVITY 2

This activity will give you further practice in using the method `alert()`. For this and subsequent activities we advise you to write down the answers on paper before launching BlueJ.

In order to test your solutions open Unit5_Project_1 and the OUWorkspace.

1 Write a single statement which, when executed, will display 'My name is Methuselah' in a dialogue box.

2 Write a single statement which, when executed, will convert the string `"Flood Warning!"` to upper-case letters and display it in a dialogue box.

3 Add a single statement to the end of the code below so that it will display 'I am studying M255' in a dialogue box. Your expression should make use of the two variables `aMessage` and `courseCode`.

```
String aMessage;
String courseCode;
aMessage = "I am studying ";
courseCode = "M255";
```

4 Modify the two assignment statements in step 3 above so that your dialogue box will display information about the courses you plan to study next year. (If you have not thought about this yet, just make up something.)

5 In the BlueJ window double-click the `Account` class to open the editor.

Now define two new instance methods for the `Account` class which will display the values of the variables `holder` and `balance` in dialogue boxes. Here are the method headers and initial comments for the methods. Make sure to include some explanatory text in your dialogue boxes, using concatenation. Using the + operator will automatically take care of converting the numerical balance into a suitable string representation.

```
/**
 * Displays the holder of the receiver in a dialogue box.
 */
public void displayHolder()

/**
 * Displays the balance of the receiver in a dialogue box.
 */
public void displayBalance()
```

Once you have successfully recompiled the `Account` class, test your new methods in the workspace by creating an instance of the `Account` class, setting its `holder` and `balance`, then sending it the messages `displayHolder()` and `displayBalance()`.

Please note that here (as in other parts of the unit) the *single* quotes around a sentence merely indicate what words the dialogue box should display – the quote marks themselves do *not* form part of the desired display.

DISCUSSION OF ACTIVITY 2

1 To produce the required dialogue box you would need to execute a statement like:

```
OUDialog.alert("My name is Methuselah");
```

2 The single statement which will display 'Flood Warning!' in upper-case letters is:

```
OUDialog.alert("Flood Warning!".toUpperCase());
```

3 You were asked to extend the given statement sequence, so that execution would produce a dialogue box with the text 'I am studying M255'. The code you need to add is:

```
OUDialog.alert(aMessage + courseCode);
```

4 To display information about the course you plan to study next year, you would need to change the two assignment statements to something like:

```
aMessage = "Next year I plan to study ";
courseCode = "M256";
```

The statement

```
OUDialog.alert(aMessage + courseCode);
```

will remain the same.

5 This was our solution:

```
/**
 * Displays the holder of the receiver in a dialogue box.
 */
public void displayHolder()
{
    OUDialog.alert("The holder is " + this.getHolder());
}

/**
 * Displays the balance of the receiver in a dialogue box.
 */
public void displayBalance()
{
    OUDialog.alert("The balance is " + this.getBalance());
}
```

You may have used **local variables** in your methods. For example:

```
/**
 * Displays the holder of the receiver in a dialogue box.
 */
public void displayHolder()
{
    String theHolder = this.getHolder();
    OUDialog.alert("The holder is " + theHolder);
}
```

Using local variables to hold intermediate results in a method can make the code easier to read. Variables declared inside a method, like `theHolder` in the code above, are **local** to the method, cannot be accessed from outside it, and exist only for as long as the method is executed.

1.4 The class method `OUDialog.confirm()`

The method `confirm()` takes a string as an argument, usually a question such as 'Do you really want to delete this file?'.

Like `alert()`, the string argument of `confirm()` is displayed in a modal dialogue box but, instead of OK, there are two buttons, labelled Yes and No.

For example, execution of

```
OUDialog.confirm("Are you over 16?");
```

results in the dialogue box shown in Figure 6.

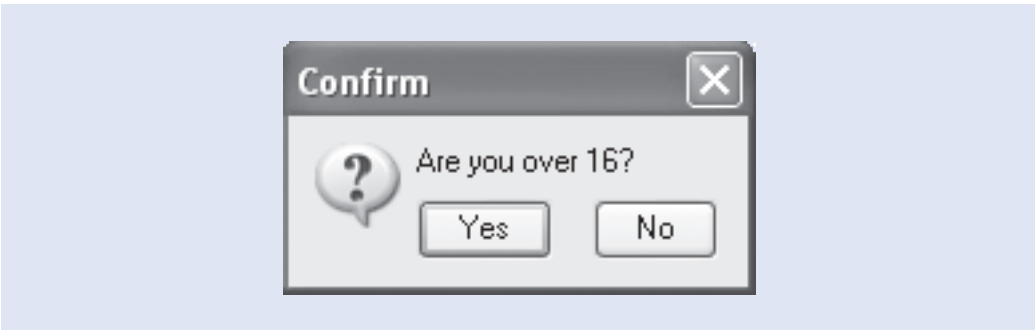

Figure 6 An example using `OUDialog.confirm()`

To dismiss the dialogue box, you must click either the **Yes button** or the **No button**. A confirm dialogue box returns a Boolean value. Clicking on Yes returns the value `true` and clicking on No returns the value `false`. Depending on what value is returned, the program can choose between alternative courses of action.

ACTIVITY 3

There is no project associated with this activity.

If it is not already open, launch BlueJ and open the OUWorkspace.

In the workspace practise using `confirm()`. Try examples like the following, executing them one at a time:

```
OUDialog.confirm("Ready to start?");
OUDialog.confirm("Have you won the lottery yet?");
OUDialog.confirm("Do you really want to delete the file "
    + "diary.txt?");
```

In each case try both the Yes and No buttons and observe the textual representation of the method's return value shown in the Display Pane.

DISCUSSION OF ACTIVITY 3

You should have observed that when the Yes button is clicked, `true` is returned as the method's return value, and when the No button is clicked, `false` is returned.

One of the statements above is longer than the physical line, which raises the question of how you should lay it out. If a statement has to be split across two or more lines, you should aim to make the statement as easy as possible for a reader to understand. It should be made obvious that the line is split.

There are some things you cannot do. One is that you cannot have a break between lines in the middle of a string. For example this is not permissible:

```
OUDialog.confirm("Do you really want to delete
    the file " + "C:\my projects\diary.txt?");
```

Also, you cannot break an identifier. For example, this is illegal:

```
OUDialog.con
firm("Do you really want to delete the file " + "diary.txt?");
```

However, breaks may be placed elsewhere. One common practice is to break a line before an operator such as + and to indent the new line, for example:

```
OUDialog.confirm("Do you really want to delete the file "
    + "C:\my projects\diary.txt?");
```

Breaks may also be placed after a comma (which is not inside a string). If in doubt, remember that the main criterion is **readability**, so try to place line breaks at a point that makes the structure of the statement clear.

We shall look at formatting of code in more detail in Section 1 of *Unit 8*.

1.5 The class method `OUDialog.request()`

Whereas the method `alert()` returns no value and `confirm()` returns either `true` or `false`, the method `request()` asks the user for input and, if OK is clicked, returns whatever is in the **input box** as a `String` object.

For example, executing

```
inputName = OUDialog.request("Hi! What is your name?");
```

will produce the dialogue box shown in Figure 7 (assuming `inputName` has been already declared as a `String` variable).

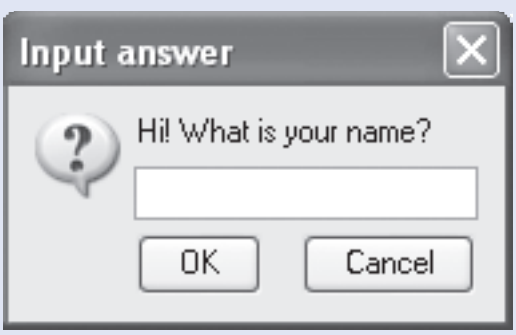

Figure 7 An example using `OUDialog.request()`

Remember that single quotes in normal text are just being used to mark the words to be displayed, and do not form part of the display themselves.

If you were to type 'Marta Friedman' and click OK, the `String` `"Marta Friedman"` would be returned and assigned to the variable `inputName`.

SAQ 2

The dialogue box in Figure 8 is produced as a result of executing the class method `request()`. What is returned if OK is clicked?

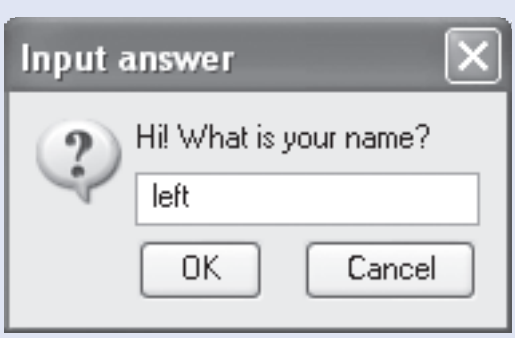

Figure 8 An example using `OUDialog.request()`

ANSWER

Clicking OK with 'left' in the dialogue box results in the `String` object `"left"` being returned.

It is important to understand what happens when the **Cancel button** is clicked in a dialogue box such as the one above, and how this differs from clicking OK with an empty input box.

If Cancel is clicked, then the method `request()` will return the value `null`.

On the other hand, if OK is clicked when the input box is empty, an empty string `""` will be returned.

Note that there is a very big difference between `null` and an empty string. An attempt to send any message to `null` is *always* illegal, whereas the empty string has the full protocol of `String` objects; for example, it responds to the message `length()` with the answer `0` as expected.

It is also important not to confuse an empty string with one containing a single space `" "`. If you send the message `length()` to `" "` the answer will be `1`, not `0`, so there is no doubt of the difference between the two.

SAQ 3

What would be the return value from the method `request()` in SAQ 2 if instead of entering some characters in the **input box** the user had left it blank before clicking OK?

ANSWER.....

If the input box had been left blank, the method's return value would have been an empty string, `""`.

You have learnt that unless Cancel is clicked, the method `request()` returns a string. (Remember that if Cancel is clicked `null` is returned.) If the input box is left blank this will be an empty string, but regardless of whether or not the string is empty, it is likely that it will be required for use later in the program (even the fact that a string is empty may be important information) and the program will therefore need to ensure that it is referenced by some variable.

ACTIVITY 4

By combining different methods from the protocol of `OUDialog` you can code a two-way interaction with the user.

Suppose you want to give the user the option of entering a name that the program will then display in upper case. The class method `request()` can be used to prompt the user for a name. The string which is returned must then be sent an appropriate message to produce the upper-case version. Then the method `alert()` can be used to display the upper-case version of the name, concatenated with the string `"The name given was "`.

Open the OUWorkspace in BlueJ.

Complete the sequence of statements below by replacing the comments with suitable code:

```
// Declare name as a String variable.
name =        // Insert code which generates a request dialogue box.
name = name.toUpperCase();
// Display name in upper case in a dialogue box.
```

Try out your solution in the workspace.

DISCUSSION OF ACTIVITY 4

Our code is given below:

```
String name;
name = OUDialog.request("Hi! What is your name?");
name = name.toUpperCase();
OUDialog.alert("The name given was " + name);
```

Note how the variable `name` is used first to refer to the string returned by the method `request()`, and then to the string of upper-case characters that is returned from the message `toUpperCase()`.

The identifier you choose for a variable does not affect the way in which the code is executed, though you should try to use something informative. An equally good identifier in this case would have been `input`. The following code would have resulted in exactly the same actions:

```
String input;
input = OUDialog.request("Hi! What is your name?");
input = input.toUpperCase();
OUDialog.alert("The name given was " + input);
```

The code could also be written without a variable:

```
OUDialog.alert("The name given was "
    + OUDialog.request("Hi! What is your name?").toUpperCase());
```

Although the layout helps to clarify the structure of the code, this is harder to understand than the earlier version. It is often clearer to break up a single complex statement into a number of simpler statements.

1.6 Request dialogue boxes with an initial answer

Another class method of `OUDialog` has the signature `request(String, String)`. This method is similar to the method `request()` given earlier, which had the signature `request(String)`. However, as indicated by its signature, this method has two arguments. The second argument lets the programmer provide a default input by giving an 'initial answer' which will be returned if the user simply clicks OK without typing in the input box.

For example, executing

```
OUDialog.request("What is your name, Elvis or Buddy?", "Elvis");
```

would result in the dialogue box shown in Figure 9.

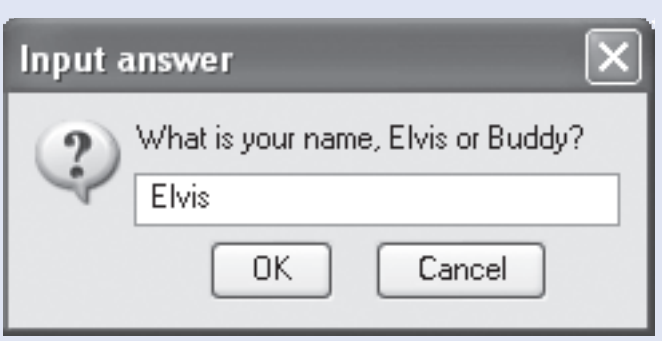

Figure 9 An example using **`OUDialog.request(String prompt, String initialAnswer)`**

The dialogue box suggests that you type Elvis or Buddy (and provides a default input, Elvis, for you). Anything you type in response to the question – and it could be neither Elvis nor Buddy but something entirely different – will be the answer. If you click OK straight away the answer will be 'Elvis'.

SAQ 4

Write down a statement that produces a dialogue box asking users whether they are using a PC or a Mac, and offers the default answer 'Mac'.

ANSWER.....

```
OUDialog.request("What type of computer are you using, PC or Mac?", "Mac");
```

1.7 Converting between data types

The `OUDialog` class methods introduced above all take `String` objects as arguments, and both `request()` methods return a value that is a `String` object. You may be wondering if `OUDialog` methods can be used only with strings or whether, for instance, numeric data types can also be used with dialogue boxes. They can, but before you can do this we need to explain about conversion between types.

SAQ 5

What do you think would happen if you tried to execute the following statement?

```
OUDialog.alert(100);
```

ANSWER.....

If you tried to execute `OUDialog.alert(100)` it would generate an error message, as the method `alert()` expects an argument of type `String`, not of type `int`.

The first problem we shall solve is how numerical data can be displayed in a dialogue box. You saw in *Unit 3* that the **concatenation operator** + for `String` has the convenient property that if one of its operands is a `String`, then the other operand will be automatically converted into its `String` representation, whatever its type.

For example:

```
int num;
num = 10;
OUDialog.alert(num + " green bottles");
```

puts up the expected dialogue box.

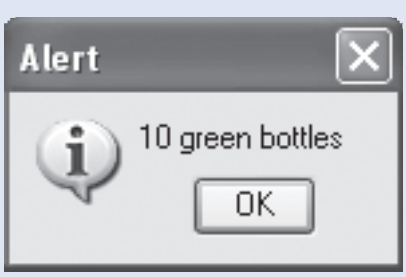

Figure 10 An **`int`** is converted to its **`String`** equivalent

This works equally well with other numerical types, for example:

```
OUDialog.alert("Pi is approximately " + 3.141592653589793);
```

But what if there is no explanatory text and we want to display a number on its own? Well, we might concatenate it to an empty string:

```
OUDialog.alert("" + 42);
```

This does what is wanted but is not exactly elegant! A better solution is to use the utility method `valueOf()` provided by the class `String`.

Just like the `OUDialog` methods this utility is a class method. You invoke it on the `String` class itself, with the `int` value you want converted as an argument. The value returned will be the `String` representation of the `int`, and can be used as the argument in an `OUDialog` message. For example:

```
int num;
num = 42;
OUDialog.alert(String.valueOf(num));
```

In fact `valueOf()` works with any primitive data type; there are multiple versions, each one accepting a different type of argument, such as `float`, `double`, `boolean` and returning an appropriate `String` representation.

This takes care of how to start with a number and produce a string to be shown in a dialogue box. If there is explanatory text the conversion will be automatic; in the rare cases where there is no text we can use `valueOf()`.

But how about the opposite problem – how to get a number *into* the program via a dialogue box? The return value from the dialogue box will be a `String`, and you cannot just assign this to a variable of type `int`. If you attempt to do so an error will occur. So we need a way to get from the `String` to the equivalent `int`.

Do not confuse the class `Integer` with the primitive data type `int`. There is a close relationship between them, which we will explore later in the course, but they are not the same thing.

There is an `Integer` class that (amongst other things) provides a utility – once again a class method – for exactly this purpose. The method is called `parseInt()`.

As an example of its use, suppose you want to convert the string `"1905"` to the corresponding `int` value `1905`. You use the `parseInt()` method of the `Integer` class, with `"1905"` as an argument.

```
Integer.parseInt("1905");
```

This will return the `int` value `1905`.

Example

The following code will accept a string via a dialogue box, convert it to its `int` equivalent and assign the result to the variable `inputNumber`:

```
int inputNumber;
String inputString;
inputString = OUDialog.request("Please enter a number");
inputNumber = Integer.parseInt(inputString);
```

SAQ 6

What is the result from executing the statement `String.valueOf(100);`?

ANSWER.....

The result from executing the statement `String.valueOf(100);` is the string `"100"`.

Building on the solution to SAQ 6, the statement

```
OUDialog.alert(String.valueOf(100));
```

would display the following dialogue box.

Figure 11 A dialogue box displaying a number as a string

SAQ 7

Suppose that your age, expressed as a number, is held by the `int` variable `age`. Write down a single statement which displays 'My age is xx' in a dialogue box, where xx is the number held by `age`.

ANSWER.....

```
OUDialog.alert("My age is " + age);
```

Exercise 2

Complete the following code so that it prompts a user for their age and assigns the response to the variable `age`.

```
String ageString;
int age;
ageString =   // Insert code which displays a request dialogue box.
age =         // Convert ageString to an int.
```

Solution.....

```
String ageString;
int age;
ageString = OUDialog.request("Please enter your age");
age = Integer.parseInt(ageString);
```

The string which is to be converted to an `int` must represent an integer. If the user enters something like 'fred' or 'M255' that cannot sensibly be converted to a number, then the method `parseInt()` will be executed with an inappropriate argument and an *exception* will occur. Exceptions are a way programs indicate that something unexpected has occurred and normally result in the code ceasing execution. Exceptions will be explained in Subsection 3.2 of *Unit 8*.

SAQ 8

Combine the code in Exercise 2 above into a single statement which achieves the same effect.

ANSWER.....

```
int age = Integer.parseInt(OUDialog.request("Please enter your age"));
```

ACTIVITY 5

A common notion in computing is that all computations reduce to input–process–output. Of course this supposes a broader interpretation of the terms 'input' and 'output' than just interaction with a human user via a keyboard/mouse and a screen! But here are two cases that follow the narrower interpretation.

Open the OUWorkspace in BlueJ.

1 In the Code Pane write and test some code which is to be used as part of a survey of how people voted in the last election. Your code should put up a dialogue box displaying the message 'Which party did you vote for?' and assign the answer to a variable called `party`. You can save the researcher time by providing one likely response, say 'Labour', as the initial answer in the input box.

2 This next part of the activity gives you practice in using a number of the methods and techniques that you have learnt. In the workspace write code that prompts the user for their year of birth, works out their age in years and then outputs the age in another dialogue box.

   In order to avoid dealing with the complication of which day of the year they were born on, you can assume that the calculation takes place at the end of whatever year it is when you read this activity.

   You will need to use appropriate `OUDialog` class methods to perform the input and output. You will also need to use `Integer.parseInt()` to convert the string input to an integer, because you need to do arithmetic with it; and you should also use one or more variables. Conversion of the age to a string representation will happen automatically if you include suitable text in the output and use the `+` operator.

DISCUSSION OF ACTIVITY 5

1 The code required to read in the voting survey data is:

```
String party;
party = OUDialog.request("Which party did you vote for?", "Labour");
```

2 Here is some code for calculating someone's age from their year of birth.

```
int yearOfBirth;
int age;
int currentYear = 2006;     // Use an appropriate value here.
String input;
input = OUDialog.request("What year were you born?");
yearOfBirth = Integer.parseInt(input);
age = currentYear - yearOfBirth;
OUDialog.alert("You are " + age + " years old!");
```

Many different solutions would work equally well!

2 Conditions and selection

Here we will outline some situations in which the course of action is determined by a Boolean expression. Whether the Boolean expression evaluates to `true` or `false` determines what program statements get executed. A Boolean expression used in this way is termed a **Boolean condition**, or simply a **condition**. We shall first look at `if` statements with simple conditions, and then proceed to construct composite conditions for `if` statements.

2.1 `if` statements with simple conditions

Imagine you are conducting a survey and you are asked to keep a count of those persons of age 40 and over. Assuming a variable `fortyAndOver` of type `int` which records the count has already been declared and initialised elsewhere, the following code does the job.

```
boolean result;
result = OUDialog.confirm("Are you 40 or over?");
if (result)
{
    fortyAndOver = fortyAndOver + 1;
}
```

The statement `fortyAndOver = fortyAndOver + 1;` will be executed only if the return value from the method `confirm()` is `true`.

If you also need to keep a separate count of those under 40, then (assuming the `int` variable `underForty` has also been declared and initialised) the code required is this:

```
boolean result;
result = OUDialog.confirm("Are you 40 or over?");
if (result)
{
    fortyAndOver = fortyAndOver + 1;
}
else
{
    underForty = underForty + 1;
}
```

The two kinds of condition selection above illustrate the two forms of an `if` statement. The first form is the `if-then` statement; the second, closely related, is the `if-then-else` statement. The structures are named after the Java keywords involved, `if` and `else` – the '`then`' is implied.

The `if-then` statement has the form:

```
if (condition)
{
    then statement block
}
```

Although `then` is not actually a Java keyword we will leave it in code styling when referring to a statement block, to indicate we are talking about a section of code.

Here `condition` must be Boolean – it must be an expression that evaluates to either `true` or `false`. It must also be enclosed between parentheses, `(` and `)` – if these are left out Java will report an error.

If the `condition` evaluates to `true` the `then` statement block will be executed, otherwise it will not. A **statement block** consists of one or more statements enclosed between braces, `{` and `}`. Bundling statements together in a block makes them into a unit that will be selected, or not selected, as a whole, so the `if-then` statement can control whether a whole section of code is executed or not.

Note that what we have described, an `if-then` statement, is a statement that contains other statements within it. This nesting of statements within statements is common.

When the statement block has executed or not as the case may be, the flow of program execution will pass on to whatever code follows the `if-then` statement.

The `if-then-else` statement has the form:

```
if (condition)
{
    then statement block
}
else
{
    else statement block
}
```

If the `condition` evaluates to `true`, the `then` statement block is executed. Otherwise, it evaluates to `false` and the `else` statement block is executed.

When either the `then` or the `else` statement block has been executed, the flow of program control will pass on to whatever code follows the `if-then-else` statement.

Where a statement block contains only a single statement it is permissible to leave the braces out. However, we strongly advise against doing so, because it frequently leads to program bugs which are hard to detect. Here is an example of what can go wrong.

Suppose the `if-then` statement at the start of the section were written without braces, like this:

```
boolean result;
result = OUDialog.confirm("Are you 40 or over?");
if (result)
    fortyAndOver = fortyAndOver + 1;
```

Now the programmer decides that it would be a nice gesture to display a message 'Life begins at forty' whenever the person concerned is forty or over. So an extra line of code is added:

```
boolean result;
result = OUDialog.confirm("Are you 40 or over?");
if (result)
    fortyAndOver = fortyAndOver + 1;
    OUDialog.alert("Life begins at forty");
```

To everyone's surprise, when the code is run, the message 'Life begins at forty' is displayed for *everyone*, regardless of age. This has happened because, in the absence of braces, only the statement immediately following `if (condition)` is controlled by the condition. The next statement after that – `OUDialog.alert("Life begins at forty");` – is executed irrespective of whether the condition is satisfied or not.

Once the two statements are enclosed in a block, the code will perform as intended, as shown below:

```
boolean result;
result = OUDialog.confirm("Are you 40 or over?");
if (result)
{
    fortyAndOver = fortyAndOver + 1;
    OUDialog.alert("Life begins at forty");
}
```

Another easy mistake to make is to place a semicolon, for example, after the condition:

```
boolean result;
result = OUDialog.confirm("Are you 40 or over?");
if (result);
{
    fortyAndOver = fortyAndOver + 1;
}
```

This time `fortyAndOver` gets incremented whatever the age! The reason is that Java reads the semicolon as the end of a `then` statement block with nothing in it, like this:

```
if (condition) empty statement here;
{
    fortyAndOver = fortyAndOver + 1;
}
```

Now it does not matter whether `condition` is true or not. All the condition holds sway over is whether or not the empty `then` statement block is 'executed'. Either way, program control then moves on normally and executes whatever comes after the `if-then` statement, which happens to be the block containing `fortyAndOver = fortyAndOver + 1`.

Example

This example uses an `if-then-else` statement in combination with both input from the user and output to the user. We give two versions. Both versions do exactly the same thing: the only difference is that the first version makes use of variables to store intermediate results, whereas the second version does not.

First version

```
boolean result;
String output;
result = OUDialog.confirm("Does two and two make four?");
if (result)
{
    output = "right.";
}
else
{
    output = "wrong.";
}
OUDialog.alert("You are " + output);
```

Second version

```
if (OUDialog.confirm("Does two and two make four?"))
{
    OUDialog.alert("You are right.");
}
else
{
    OUDialog.alert("You are wrong.");
}
```

Exercise 3

Taking the example above as a guide, write code that uses `OUDialog.confirm()` to produce a dialogue box containing the text 'Click a button' and then reports to the user which button has been selected. (You may find it useful to introduce variables to store intermediate results as shown above.)

Solution

```
if (OUDialog.confirm("Click a button"))
{
    OUDialog.alert("Yes clicked");
}
else
{
    OUDialog.alert("No clicked");
}
```

Here is another version, which uses variables to store intermediate results:

```
boolean result;
String button;
result = OUDialog.confirm("Click a button");
if (result)
{
    button = "Yes clicked";
}
else
{
    button = "No clicked";
}
OUDialog.alert(button);
```

2.2 Comparing values

Note that it is generally not possible to determine the value of a **Boolean condition** (which will be either `true` or `false`) simply by reading the program code. In other words, you cannot determine it *statically*. The result can only be determined *dynamically* – by executing the code.

For example, you cannot tell what the answer returned as a result of executing `OUDialog.confirm("Click a button")` will be until the dialogue box has been

produced and a button has been clicked. Indeed, if the answer were known in advance there would be no need to include the dialogue box in the first place!

The `OUDialog` method `confirm()` conveniently returns either `true` or `false` and you can make direct use of this value in an `if` statement. However, things are generally less straightforward. Usually you want to make a selection which depends on a comparison between values – whether one variable has the same value as another, is different, is larger, is smaller, or some combination of these.

Suppose, for example, that you wanted to password protect some sensitive online information. Users would have to enter their password into a dialogue box, and the input string would be compared with the stored password. Suppose that the user input is assigned to the `String` variable `passwordEntered` and that the stored password is `"FirstOfMay"`. The condition to verify the password can be written as the Java expression:

```
passwordEntered.equals("FirstOfMay")
```

The message `equals()` for the class `String` was introduced in Section 2 of *Unit 3*.

If the `String` object referenced by `passwordEntered` has exactly the same sequence of characters in the same order as the message argument `"FirstOfMay"` the message answer is `true`, otherwise it is `false`. The code required is:

```
String passwordEntered = OUDialog.request("Enter your password.");
if (passwordEntered.equals("FirstOfMay"))
{
   OUDialog.alert("Welcome.");
}
else
{
   OUDialog.alert("Access denied.");
}
```

SAQ 9

Rewrite the example above without using the local variable `passwordEntered`.

ANSWER.....

```
if (OUDialog.request("Enter your password.").equals("FirstOfMay"))
{
   OUDialog.alert("Welcome.");
}
else
{
   OUDialog.alert("Access denied.");
}
```

Exercise 4

Suppose that a new message, called `simplyRed()`, is required in the protocol of the `Frog` class. Here are the initial comment and method heading.

```
/**
 * If the colour of the receiver is red, move the receiver
 * right twice; if not move the receiver right once.
 */
public void simplyRed()
```

Write down (on paper) the code for the method.

Solution

```
/**
 * If the colour of the receiver is red, move the receiver
 * right twice; if not move the receiver right once.
 */
public void simplyRed()
{
    if (this.getColour().equals(OUColour.RED))
    {
        this.right();
        this.right();
     }
     else
     {
        this.right();
    }
}
```

Exercise 5

In this exercise you will look in detail at the method `debit()` of the `Account` class. In the code for this method you can see an example of the use of an `if` statement. Examine the code and write down an explanation of the way in which the method achieves the effect desired. (You can imagine that you will be posting your explanation to a FirstClass conference in response to a query from another student.)

```
/**
 * If the balance of the receiver is equal to or greater than the
 * argument anAmount, the balance of the receiver is debited by the
 * argument anAmount and the method returns true.
 *
 * If the balance of the receiver is not equal to or greater than the
 * argument anAmount, the method simply returns false.
 */
public boolean debit(double anAmount)
{
    if (this.getBalance() >= anAmount)
    {
        this.setBalance(this.getBalance() - anAmount);
        return true;
    }
    else
    {
        return false;
    }
}
```

You met the operator >= in Subsection 2.1 of *Unit 3*.

Solution

Our explanation of how the method works is as follows.

If the balance is greater than or equal to the amount to be withdrawn then the expression `this.getBalance() >= anAmount` evaluates to `true`; otherwise it evaluates to `false`. The debit is carried out only if there are sufficient funds, and the method answers `true` if this is the case and `false` otherwise.

In the next activity you will need to test whether an integer is odd or even. You can test whether an integer `x` is odd using an expression such as `x % 2 == 1`, where `%` is Java's remainder (or modulus) operator. This expression evaluates to `true` if `x` is odd, and `false` otherwise.

You met the operator `%` in Subsection 2.1 of *Unit 3*.

ACTIVITY 6

Launch BlueJ and open Unit5_Project_2. Double-click on the `Frog` class to open the BlueJ editor.

Write a method `oddRightTwo()` for the `Frog` class which moves a frog right twice if it is currently in an odd position. If the current position is even, then the frog should not be moved but a suitable message should be output in a dialogue box.

When you have written your method, compile the `Frog` class. Now open the OUWorkspace. From the Graphical Display menu select Open to make the Amphibians window visible.

Now create an instance of `Frog` in the workspace and send it the message `oddRightTwo()`, observing what happens in the Amphibians window. Then change the frog's position so that it is at an even position and re-send the message, checking that the behaviour is as intended.

DISCUSSION OF ACTIVITY 6

Here is our solution:

```
/**
 * Causes the receiver to move two positions to the right
 * if the position of receiver is odd; otherwise produces a
 * warning dialogue box.
 */
public void oddRightTwo()
{
    int currentPos;
    currentPos = this.getPosition();
    if (currentPos % 2 == 1)
    {
        this.right();
        this.right();
    }
    else
    {
        OUDialog.alert("Position is not odd.");
    }
}
```

The equality operator `==` was introduced in Subsection 2.1 of *Unit 3*.

Alternatively you can do without the local variable `currentPos` that is used above to remember the current position.

```
/**
 * Causes the receiver to move two positions to the right
 * if the position of receiver is odd; otherwise produces a
 * warning dialogue box.
 */
public void oddRightTwo()
{
    if (this.getPosition() % 2 == 1)
    {
        this.right();
        this.right();
    }
    else
    {
        OUDialog.alert("Position is not odd.");
    }
}
```

ACTIVITY 7

Open Unit5_Project_2. Double-click on the `Frog` class to open the BlueJ editor.

1 Write a method `rightIfGreen()` that moves a `Frog` object three times to the right or left, depending on whether the frog is green or another colour. If the frog is green then it should move to the right three times; if it is another colour it should move three times to the left. When you have written your method compile the `Frog` class.

Now open the OUWorkspace. From the Graphical Display menu select Open to make the Amphibians window visible.

2 Create an instance of `Frog` in the workspace and send it the message `rightIfGreen()`, observing what happens in the Amphibians window. Then change the frog's colour and re-send the message, checking that the behaviour is as intended.

DISCUSSION OF ACTIVITY 7

1 Here is our solution:

```
/**
 * Increments position of receiver by 3 if colour of receiver
 * is green else decrements position of receiver by 3.
 */
public void rightIfGreen()
{
    if (this.getColour() == OUColour.GREEN)
    {
        this.right();
        this.right();
        this.right();
     }
```

```
        else
        {
            this.left();
            this.left();
            this.left();
        }
    }
```

2 We used the following code in the OUWorkspace to check that `rightIfGreen()` works as intended:

```
Frog sam = new Frog();
sam.rightIfGreen();
sam.brown();
sam.rightIfGreen();
```

ACTIVITY 8

Open Unit5_Project_2. Double-click on the `Frog` class to open the BlueJ editor.

1 Write a method `extremeLeft()` which answers `true` if the position of a `Frog` object is `1` and `false` otherwise. When you have written your method compile the `Frog` class.

2 Now open the OUWorkspace, making sure that Show Results is ticked. Create an instance of `Frog` in the workspace and send it the message `extremeLeft()`, observing what appears in the Display Pane. Then move the frog one step to the right and re-send the message, checking that the message answer has changed appropriately.

3 So far you have tested your `Frog` methods on `Frog` objects, but of course the methods are inherited by `HoverFrog` and so should work equally well with `HoverFrog` objects. To check that this is the case, create an instance of `HoverFrog` and carry out some tests with it as you did with the `Frog` object.

DISCUSSION OF ACTIVITY 8

1 Here is our solution:

```
/**
 * Answers true if the position of the receiver is 1;
 * otherwise answers false.
 */
public boolean extremeLeft()
{
    return(this.getPosition() == 1);
}
```

The following also works, but is less elegant:

```
/**
 * Answers true if the position of the receiver is 1;
 * otherwise answers false.
 */
public boolean extremeLeft()
{
    if (this.getPosition() == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Because the condition already evaluates to either `true` or `false`, you can return it directly. It may take you a little while to feel confident that the first version is correct. The longer solution is acceptable, although we would encourage you to move towards the more compact approach.

2 We used the following code, executing it line by line:

```
Frog les = new Frog();
les.extremeLeft();              // Display Pane displays 'true'
les.right();
les.extremeLeft();              // Display Pane displays 'false'
```

3 We used the following, with the same results as in part 2 of the activity.

```
HoverFrog hf = new HoverFrog();
hf.extremeLeft();
hf.right();
hf.extremeLeft();
```

ACTIVITY 9

In this activity you are asked to write code using dialogue boxes to convert a name, input by the user, into all upper-case or all lower-case letters, and then display the converted version. Here is the specification for the code.

- First the user is prompted to input their name. This name is then displayed and the user is asked to confirm that they want it converted to upper-case letters.
- If Yes is clicked, the name is output in upper-case letters.
- If No is clicked, the user is next asked to confirm that they want the name converted to lower-case letters.
- If Yes is clicked at this stage, the name is output in lower-case letters.
- If No is clicked, no further action is taken.

From BlueJ's Tools menu, open the OUWorkspace. Copy the partial solution below into the Code Pane and complete the code where indicated. You will need to use the message `toLowerCase()`, which can be sent to a `String` object in order to convert it to lower-case letters. Execute your code to test that it works correctly.

`toUpperCase()` and `toLowerCase()` were introduced in Subsection 2.3 of *Unit 3*.

```
String input;
input = OUDialog.request("Please type in your name.");
if (OUDialog.confirm("Your name is " + input + ". Convert it to upper case?"))
{
    OUDialog.alert("Your name in upper case is " + input.toUpperCase());
}
else
{
    // Insert code here to handle the case where the user clicks No.
}
```

DISCUSSION OF ACTIVITY 9

Here is our solution:

```
String input;
input = OUDialog.request("Please type in your name.");
if (OUDialog.confirm("Your name is " + input + ". Convert it to upper case?"))
{
    OUDialog.alert("Your name in upper case is " + input.toUpperCase());
}
else
{
    if (OUDialog.confirm("Convert it to lower case?"))
    {
       OUDialog.alert("Your name in lower case is " + input.toLowerCase());
    }
}
```

This example shows how an `if` statement can contain another `if` statement so that more than one condition can be checked.

2.3 Handling the output of the Cancel button

Before going on we shall look briefly at how to handle the situation where a user presses the **Cancel button** in a dialogue box. The methods `OUDialog.request(String prompt)` and `OUDialog.request(String prompt, String initialAnswer)` produce dialogue boxes which contain two buttons – an OK button and a Cancel button. You may recall that clicking the Cancel button results in `null` being returned.

Consider the following scenario: a user of a word-processing program has requested a file save, and a method `request(String prompt, String initialAnswer)` is used to prompt for the file name. The existing file name is given as the initial answer. The user has various courses of action to choose from. They can click OK and save the file under its existing name or they can replace the default name by a new one and click OK.

On the other hand, they may decide not to save the file after all, and click Cancel. This will return `null`, as you have seen. If the program blindly proceeds to treat this like any other message answer, it will attempt to save the file with `null` as the file name. This obviously makes no sense and in fact an exception would occur, as you will learn in Section 1 of *Unit 12*.

What ought to happen if Cancel is clicked? There are various possibilities but clearly the most sensible is to not save the file and to notify the user that this is the case. You need some code that will detect if `null` is returned and take the appropriate action. Here is some Java code that achieves this.

```
String filename;
filename = OUDialog.request("Please specify file name", "currentName");
if (filename == null)
{
    OUDialog.alert("File not saved.");
}
else
{
   /* ToDo: Code to check that the filename entered is valid and
    * if so save the file under that name would go here
    */
}
```

Note that you are not in a position to write the last part of the code yet, since you have not learnt about files, so for now we have just written a comment showing that something still needs to be done and indicating what. Later in the course you will be in a position to write the code.

3 Boolean expressions

So far all our conditions have either taken a Boolean value returned by a dialogue box and used it directly, or involved a limited range of comparisons, such as whether two things are equal, or whether one number is greater than another. To write more versatile code you need a wider range of different comparisons between values.

You also need to be able to combine two or more conditions into a single compound condition which can be evaluated to yield an overall `true` or `false`. For example, you might want frogs to behave in a particular way if their position were to the right of the centre stone and their colour were purple.

In this section we review the equality, relational and logical operators you met in Subsection 2.1 of *Unit 3*. We shall also explain a bit more about how logical operators work.

3.1 Simple comparisons

You will recall from *Unit 3* that Java has a number of **comparison operators** which can be applied to primitive data types.

Equality operators	==	equal to
	!=	not equal to
Relational operators	<	less than
	<=	less than or equal to
	>	greater than
	>=	greater than or equal to

The operator == can also be applied to objects, in which case it tests whether two variables reference exactly the same area in memory, that is, whether they reference the same object. Similarly, for objects, != tests whether its operands reference different objects.

These are all **binary operators**, because they require two operands. They compare one value with another. (An operator which takes only one operand is called **unary**.)

You will remember that if an expression involves a combination of operands, **parentheses** (round brackets) are used where necessary to clarify the order in which the operations should be evaluated.

We have also mentioned the method `equals()` applied to `String` objects. This gives a way of testing if two strings are 'the same' in the sense of representing the identical sequence of characters in the same order, even though the `String` objects concerned may be stored in different locations in memory and so be different objects.

SAQ 10

If `count` and `total` have the values `12` and `14` respectively and `item` has the value `"xyz"`, what are the results of evaluating the following expressions? That is, what do they evaluate to, `true` or `false`?

(a) `count > 12`

(b) `count + 2 != total`

(c) `item.equals("xyz")`

(d) `count + 2 <= 14`

ANSWER

(a) Evaluates to `false`.

(b) Evaluates to `false`; `count + 2` has the value `14`, and it is false that this *does not* equal the value of `total`.

(c) Evaluates to `true`; the two strings have the same characters in the same order.

(d) Evaluates to `true`; `count + 2` has the value `14`, which is less than *or equal to* `14`.

3.2 Boolean operators

Java also has operators which take Boolean operands – any expressions which evaluate to `true` or `false`. The operands could be comparisons for example, or values passed back as the return value from a confirm dialogue box, or values stored in a `boolean` variable. By using the logical operators you can build up compound expressions which test if some combination of conditions is satisfied.

Kind	Java symbol	Operation	Effect
Unary	`!`	not	`!a` is true if `a` is `false`, and vice versa.
Binary	`&&`	and	`a && b` is `true` if `a` is `true` and so is `b`. Otherwise it is `false`.
	`||`	or	`a || b` is `true` if either `a` or `b`, or both of them, are `true`. Otherwise it is `false`.

SAQ 11

In this question, `count` and `sum` have the values in the table:

	count	sum
(i)	10	50
(ii)	20	200
(iii)	4	40
(iv)	9	110

(a) What does the following expression evaluate to for each pair of values of `count` and `sum` in the table above?

```
(count < 10) && (sum <= 100)
```

(b) What does the following expression evaluate to for each pair of values?

```
(count < 10) || (sum <= 100)
```

(c) What does the following expression evaluate to for each value of `count`?

```
!(count <= 10)
```

ANSWER.....

(a) (i) `false` (ii) `false` (iii) `true` (iv) `false`

(b) (i) `true` (ii) `false` (iii) `true` (iv) `true`

(c) (i) `false` (ii) `true` (iii) `false` (iv) `false`

Exercise 6

(a) Suppose a variable `number` of type `int` already exists and has been given some value. Write code to do the following.

(i) Declare three boolean variables, called `positive`, `zero` and `negative`.

(ii) Set `positive` to `true` if `number` is greater than `0` (otherwise set `positive` to `false`), set `zero` to `true` if `number` is equal to `0` (otherwise set `zero` to `false`), and set `negative` to `true` if `number` is less than `0` (otherwise set `negative` to `false`).

(b) Suppose `day` and `month` are variables of type `int` which already exist and have been given values. Write down a single statement that will declare a boolean variable `dateValid` and set it to `true` if `month` is equal to `1` and `day` is in the range `1` to `31` inclusive, `false` otherwise.

Solution.....

(a) A short solution is this:

```
boolean positive = (number > 0);
boolean zero = (number == 0);
boolean negative = (number < 0);
```

However, you may well have used a different approach, such as:

```
boolean positive = false;
boolean zero = false;
boolean negative = false;
if (number > 0)
{
    positive = true;
}
else
{
    if (number == 0)
    {
        zero = true;
    }
    else
    {
        negative = true;
    }
}
```

In this second solution all three variables are first initialised to `false`. When the following lines are executed the value `true` is assigned to the appropriate variable.

The first solution we gave is more elegant. Although we realise it may not always be easy to see how to write code which assigns Boolean values directly in this way, we would encourage you to try to do so.

(b) One answer is:

```
boolean dateValid = (month == 1) && ((day >= 1) && (day <= 31));
```

The following, which uses fewer parentheses, also works:

```
boolean dateValid = (month == 1) && (day >= 1) && (day <= 31);
```

Exercise 7

(a) Suppose the date of my birthday is 16 November and the `int` variables `day` and `month` hold values which specify the current day of the month and the current month, respectively. (For example, for 12 May, `day` would have the value `12` and `month` the value `5`.)

Write a single statement that will cause the appropriate value to be assigned to the `boolean` variable `isMyBirthday`, i.e. `true` if the current date is my birthday, `false` otherwise. (Assume `isMyBirthday` has been declared already.)

(b) A pharmaceutical research organisation is seeking volunteers to take part in a new drug study. One group of volunteers must be aged between 25 and 35 (inclusive) and be non-smokers.

If the variables `age` of type `int` and `isSmoker` of type `boolean` already exist and have been assigned values which specify a volunteer's age and whether or not the volunteer is a smoker, write down a single statement that will result in the `boolean` variable `isEligible` being assigned a value that reflects the volunteer specification. (Assume `isEligible` has already been declared.)

Solution

(a)
```
isMyBirthday = (day == 16) && (month == 11);
```

(b)
```
isEligible = (!isSmoker) && (age >= 25) && (age <= 35);
```

ACTIVITY 10

Open Unit5_Project_2 and the OUWorkspace. From the Graphical Display menu select Open to make the Amphibians window visible.

1 In the workspace create an instance of the `HoverFrog` class and send it some `setHeight()` messages with different arguments, including negative numbers and numbers greater than 6. Observe the effect of each message on the graphical representation of the hoverfrog in the Amphibians window. Inspect the hoverfrog to check that the value of the instance variable `height` is consistent with the height of the hoverfrog icon in the window in each case. You should find the height accurately reflects the value of the instance variable.

2 The following code (which is incomplete) reads in a number from a user, tests whether it is in the range 0 to 6, and informs the user whether or not it is in range. Without looking at the code for the method `setHeight()`, complete the code by writing the condition we have described as a comment in the code.

```
String inputString;
int number;
inputString = OUDialog.request("Input a number");
number = Integer.parseInt(inputString);
if // Write a condition to test if number is in the range 0–6 inclusive.
{
    OUDialog.alert("Number is in range.");
}
else
{
    OUDialog.alert("Number is out of range.");
}
```

Test your code in the workspace with a variety of numbers, both within and outside the range. Test at the boundaries, that is, make sure it works correctly when the user inputs 0 or 6.

DISCUSSION OF ACTIVITY 10

1 The code for the method `setHeight()` is written so that it only sets the height to values in the range 0 to 6 inclusive. If the method argument is outside this range, no action is taken.

2 Our solution was this:

```
String inputString;
int number;
inputString = OUDialog.request("Input a number");
number = Integer.parseInt(inputString);
if ((number >= 0) && (number <= 6))
{
    OUDialog.alert("Number is in range.");
}
else
{
    OUDialog.alert("Number is out of range.");
}
```

ACTIVITY 11

Open Unit5_Project_2. Double-click on the `HoverFrog` class to open the BlueJ editor.

Write a method called `inTheCorner()` for the `HoverFrog` class which answers `true` if the receiver has any of the following combinations of values for its instance variables `height` and `position`:

`height` set to `0` and `position` set to `1`

`height` set to `0` and `position` set to `11`

`height` set to `6` and `position` set to `1`

`height` set to `6` and `position` set to `11`

For any other combinations, the method answers `false`.

Compile `HoverFrog` and close the editor window.

Now open the OUWorkspace. From the Graphical Display menu select Open to make the Amphibians window visible.

Create an instance of `HoverFrog`. Send it the message `inTheCorner()` and verify that the result shown in the Display Pane is `true`.

Now use the messages `setPosition()` and `setHeight()` to move the hoverfrog to a range of different locations, and send it the message `inTheCorner()` at each one, checking that the results are as expected.

DISCUSSION OF ACTIVITY 11

There are many possible solutions. Here are a few variations:

```
/**
 * Checks whether the receiver is 'in the corner'.
 * The corners are the following locations:
 * height 0 and position 1;
 * height 0 and position 11;
 * height 6 and position 1;
 * height 6 and position 11.
 */
public boolean inTheCorner()
{
    return
        (((this.getHeight() == 0) && (this.getPosition() == 1))
        || ((this.getHeight() == 0) && (this.getPosition() == 11))
        || ((this.getHeight() == 6) && (this.getPosition() == 1))
        || ((this.getHeight() == 6) && (this.getPosition() == 11)));
}
```

A much neater solution would be:

```
public boolean inTheCorner()
{
    return
        (((this.getHeight() == 0) || (this.getHeight() == 6))
         && ((this.getPosition() == 1) || (this.getPosition() == 11)));
}
```

The next possible solution uses nested combinations of `if` statements instead of using the `&&` operator.

```
public boolean inTheCorner()
{
    // If the receiver is in a 'corner' position, answer true.
    if (this.getHeight() == 0)
    {
        if ((this.getPosition() == 1) || (this.getPosition() == 11))
        {
            return true;
        }
    }
```

```
        if (this.getHeight() == 6)
        {
            if ((this.getPosition() == 1) || (this.getPosition() == 11))
            {
                return true;
            }
        }
        // Otherwise answer false
        return false;
    }
```

Notice how the `return` statement is used to return a value and avoid further processing as soon as an answer has been determined.

It is always possible to use nested combinations of `if` statements to replace an `&&` operator. The solution you choose will depend on factors such as the clarity and efficiency of your code.

Finally, here is a solution that uses a 'helper' method:

```
    public boolean inTheCorner()
    {
        return this.isAt(0,1) || this.isAt(0,11) || this.isAt(6,1) || this.isAt(6,11);
    }

    /**
     * Helper method for inTheCorner.
     */
    private boolean isAt(int height, int position)
    {
        return (this.getHeight() == height) && (this.getPosition() == position);
    }
```

We tested our code by creating a hoverfrog and 'walking it round the box' with the sequence of statements given below.

```
HoverFrog hf1 = new HoverFrog();
hf1.right();
hf1.setPosition(11);
hf1.up();
hf1.setHeight(6);
hf1.left();
hf1.setPosition(1);
hf1.down();
hf1.setHeight(0);
```

After each statement we executed `hf1.inTheCorner();` to check the result. These tests should give alternately `true` and `false` answers!

Finally, we tested in the middle.

```
hf1.setPosition(6);
hf1.setHeight(3);
```

& and |

Before leaving this section we shall take a brief look at two additional Boolean operators that you may meet. These are `&` and `|` and, although they can sometimes be used to achieve the same effect as the operators `&&` and `||`, the way they are implemented is significantly different.

To understand the difference, we first look at how `&&` and `||` are implemented.

What && does

`a && b` is `true` only if both `a` and `b` are `true`. When `a && b` is evaluated, the value of `a` is determined first. If `a` evaluates to `false`, the JVM deduces immediately that `a && b` is `false` and skips the evaluation of `b`.

Only when `a` is `true` will `b` also be evaluated.

What || does

Similarly, `a || b` is `true` if either or both of `a` and `b` are `true`. When `a || b` is evaluated, the value of `a` is determined first. If `a` evaluates to `true`, the JVM deduces immediately that `a || b` is `true` and skips the evaluation of `b`.

Only when `a` is `false` will `b` also be evaluated.

In case this seems a little tricky, the following pair of flow charts may help.

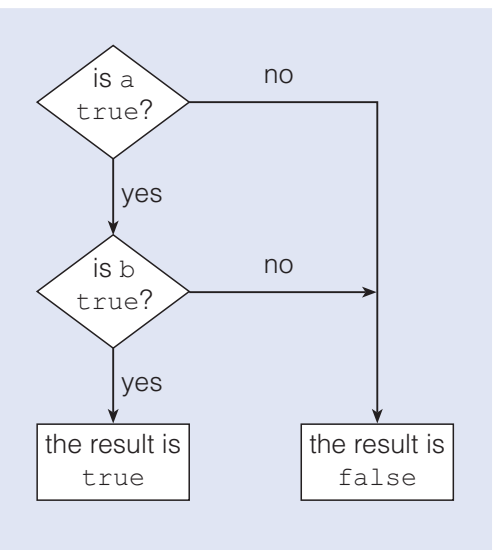

Figure 12a How `&&` is implemented

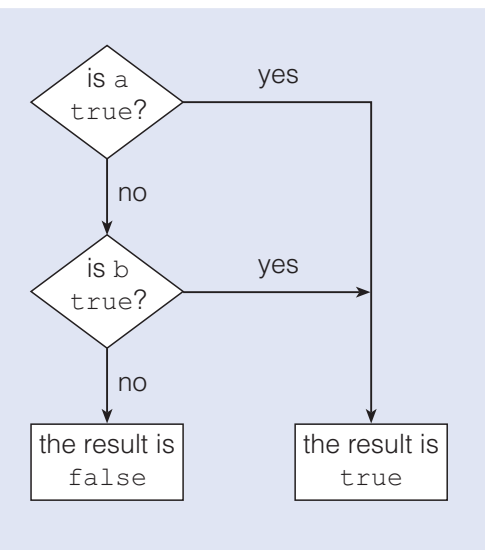

Figure 12b How || is implemented

This economical scheme – evaluating only as much as necessary – is called **lazy evaluation** or **short-circuit evaluation**.

In contrast, when `a & b` and `a | b` are evaluated both operands are evaluated *first* and only then is the result calculated. There is no short circuiting: both operands are always evaluated.

Why would this matter? Well, imagine, for example, that a programmer wanted to check whether `acc1` actually referenced an `Account` object and if so, whether the account had a balance greater than 500. Suppose they wrote the following, using the non-lazy operator `&`:

```
boolean checkResult = (acc1 != null) & (acc1.getBalance() > 500);
```

See what would happen if `acc1` were `null`. Both operands would be evaluated. When the operand on the right was evaluated, an attempt would be made to send the message `getBalance()` to a non-existent object. This would cause an exception.

Using the lazy operator `&&` there would be no problem. If the first operand evaluated to `false` the second operand would not be evaluated at all.

Because of cases like this it is generally safer to stick to the lazy operators `&&` and `||`. This is what we shall be doing in M255.

4 Iteration

In Java there are two main ways of causing an action, or group of actions, to be executed repeatedly – in other words to be *iterated*.

The first mechanism is a `for` loop. A `for` loop is suitable when you want to repeat an action a given number of times or when you want to run through a known range of values performing some action for each value in the range. After completing the specified number of iterations the iteration stops.

The second mechanism is a `while` loop. A `while` loop is suitable when you cannot tell in advance when the iteration should end. The iteration will continue looping while some particular condition is true. As soon as the condition becomes false the iteration stops.

The methods `System.out.println()` and `System.out.print()`

The methods `System.out.println()` and `System.out.print()` provide a way of producing textual output: `System` is a class that contains an object `out` that has methods `println()` and `print()`. These methods output their argument. When you use the OUWorkspace the output appears in the Display Pane. The difference between them is that:

- `System.out.println("Some text")` prints 'Some text' then starts a new line for whatever comes next;
- `System.out.print("Some text")` just prints 'Some text' with no new line following it.

In this section you will use these methods to produce textual output.

4.1 Examples of loops

To illustrate loops here are some short snippets of code. Do not worry about the details for the moment; we shall explain more in Subsection 4.2 below. These examples are simply meant to give a flavour of how `for` and `while` loops can be used.

First we look at some `for` loops.

This code outputs the same string three times:

In *Unit 3* you met the postfix operators `++` and `--`, which respectively increment and decrement a numerical variable by 1.

```
for (int numberOfTimes = 1; numberOfTimes <= 3; numberOfTimes++)
{
    System.out.println("Hello");
}
```

The following runs through the range of numbers from 20 to 30 inclusive, printing each number:

```
for (int value = 20; value < 31; value++)
{
    System.out.println(value);
}
```

The next code fragment accepts a number from the user and produces a string consisting of that number of 'x's. This demonstrates that the number of iterations can be determined by the value of a variable.

```
int numberOfX;
String result;
result = ""; // Empty string to start with
numberOfX = Integer.parseInt(OUDialog.request("Please enter a number"));
for (int num = 1; num <= numberOfX; num++)
{
    result = result + "x";
}
System.out.println(result);
```

Now we examine a couple of `while` loops. In the following example the user is repeatedly invited to enter a number. While the number they enter is less than 100, an alert dialogue box is put up informing them of the fact. As soon as the number entered is 100 or more, the condition becomes `false` and the loop terminates.

```
while (Integer.parseInt(OUDialog.request("Input a number.")) < 100)
{
    OUDialog.alert("Your number was less than 100.");
}
```

The point about this is that there is no way of predicting in advance what numbers the user will actually enter. You cannot know how many numbers there will be, nor can these numbers be expected to run neatly in some particular sequence. The code just has to keep looping round as long as the input is less than 100 and will not stop looping until the first value of 100 or more actually materialises.

The final example we give outputs the odd numbers in sequence 1, 3, 5, ... while the current odd number is less than 30:

```
int number = 1;
while (number < 30)
{
    System.out.println(number);
    number = number + 2; // Next odd number
}
```

Once each number has been output the code adds 2 to it to generate the number that will be used next time round the loop.

In this example we did not have to use a `while` loop, because the range of values we wanted to iterate through was known. We could have used a `for` loop:

```
for (int number = 1; number < 30; number = number + 2)
{
    System.out.println(number);
}
```

In fact it turns out that anything that can be done with a `while` loop can also be achieved with a `for` loop and vice versa. However, in most applications you are likely to find one form of loop is better than the other, in that it is more convenient and natural.

In the following subsections we look in detail at the structure of `for` and `while` loops, and put them to use.

4.2 for loops

A Java `for` loop has the following structure:

```
for (init; test; inc)
{
    statement block
}
```

The individual constituents are:

- `init` typically declares and initialises a variable,
- `test` is a condition which must be a Boolean expression,
- `inc` updates the variable declared in `init`; typically it increments (or decrements) it.

The sequence `init; test; inc` *must* be enclosed between parentheses. If these are missing Java will report an error. `init` and `test`, but *not* `inc`, must be followed by semicolons.

In fact the statement block can consist of zero statements, but normally this is not very useful.

The `for` statement block – often called the body of the loop – consists of one or more statements enclosed between braces to form a block. If there is only a single statement the braces can be left out, but we advise against it. This is because if you later decide to add extra statements to the loop body it is very easy to forget to insert the braces.

When the loop is executed:

1 `init` is executed once.

2 `test`, which must produce a `boolean` result `true` or `false`, is evaluated.

3 If `test` evaluates to `false`, the iteration terminates at this point. If `test` evaluates to `true`, the statement block is executed.

4 `inc` is executed.

5 Execution returns to step 2.

This keeps looping round until `test` evaluates to `false`. (Of course, it is possible that in some cases this will happen when step 2 is reached for the first time; if so, the iteration will stop immediately.) When the iteration terminates, the flow of program execution will pass on to whatever code follows the statement block.

It is possible for both `init` and `inc` to be replaced by a sequence of statements (including an empty sequence), but further discussion of this falls outside the scope of this unit.

Let us see how the steps above work out for the following example:

```
for (int move = 0; move < 3; move++)
{
    frog1.right();
    frog1.croak();
}
```

We trace what happens as a table.

Code	Value of move	Value of (move < 3)
`int move = 0;`	0	
`(move < 3)`	0	true
`frog1.right();`	0	
`frog1.croak();`	0	
`move++`	1	
`(move < 3)`	1	true
`frog1.right();`	1	
`frog1.croak();`	1	
`move++`	2	
`(move < 3)`	2	true
`frog1.right();`	2	
`frog1.croak();`	2	
`move++`	3	
`(move < 3)`	3	false

In the example above the variable `move` acts as a loop counter. All the `for` loops you will meet in M255 will have a similar counter, although it is possible to use Java `for` loops in other ways.

You may have noticed that we declared and initialised the loop variable `move` in the first line of the `for` loop:

```
for (int move = 0;  ...
```

This has the effect of making the variable `move` local to the loop: it can only be used and seen inside the loop header and body, and once the loop terminates `move` will be undefined. As `move` is declared as local to the `for` loop, it would not be listed in the OUWorkspace's Variables Pane since it does not exist outside the loop.

If you wanted `move` to also be available outside the loop, you would declare it before the first line of the loop:

```
int move;
for (move = 0;  ...
```

Now `move` will be accessible inside the loop but will also continue to exist after the loop has ended. This is not advisable and is considered bad programming style.

You may also have wondered why we initialised `move` to `0`, and used the condition `move < 3`. Why not initialise the counter to `1` and test `move <= 3`, which would have the same effect? The reason is that it is a common convention (although not essential) in Java to start loop counters from `0` rather than `1`. We thought it would be useful to introduce you to this idiom now, even though we did not use it in the previous examples.

Note too the use of the postfix operator ++ to increment `move` by 1. An alternative would be `move = move + 1`. Similarly, if you wanted to decrement by 1, you could use `move--` or `move = move - 1`. The form you use is up to you; we suggest you adopt whichever you feel most comfortable with, and you will see both styles in the rest of the course.

Exercise 8

In this pen-and-paper exercise you should assume that `frog1` references a `Frog` object and `aNumber` has been declared as an `int` and assigned some value.

(a) Using the example above as a guide, write a `for` loop that will move `frog1` left `aNumber` times. (Hint: you will need to use `aNumber` in the test expression of the loop.)

(b) Write a new method for the `Frog` class, with the method header `public void leftBy(int aNumber)`, which uses a `for` loop to move the receiver to the left by the number of positions specified by the argument, `aNumber`, in single steps of a stone at a time.

Solution

(a)

```
for (int move = 0; move < aNumber; move++)
{
    frog1.left();
}
```

(b)

```
/**
 * Moves the receiver to the left aNumber times.
 */
public void leftBy(int aNumber)
{
    for (int move = 0; move < aNumber; move++)
    {
        this.left();
    }
}
```

In this next activity you will use `for` loops to organise some frogs in the Amphibians window to perform a set of simple dances.

ACTIVITY 12

From BlueJ, open Unit5_Project_2 and the OUWorkspace, then make the Amphibians window visible by selecting Open from the Graphical Display menu.

1 You can view a dance as a short sequence of movements, which are repeated for the duration of the dance. To make frogs dance you first need to consider which of the messages that could be sent to a `Frog` object would make it exhibit dance-like behaviour in the Amphibians window. You should be familiar enough by now with the protocol of `Frog` objects to agree that sending the messages `left()`, `right()` and `jump()` to an instance of the `Frog` class would result in behaviour that could be used as the basis to simulate frogs dancing.

To start, create two instances of the `Frog` class by executing in the workspace:

```
Frog sam = new Frog();
Frog lew = new Frog();
```

Now execute the following group of statements in the workspace and observe the effect in the Amphibians window:

```
sam.right(); sam.right(); sam.jump(); sam.left();
```

To turn this sequence of movements into a dance you need to send this series of messages repeatedly to the same `Frog` object. Of course, one way to do this is write the series out as many times as it takes. If, for example, four repetitions were required, you could use the following:

```
sam.right(); sam.right(); sam.jump(); sam.left();
sam.right(); sam.right(); sam.jump(); sam.left();
sam.right(); sam.right(); sam.jump(); sam.left();
sam.right(); sam.right(); sam.jump(); sam.left();
```

Clearly, this is mightily inconvenient! And what if you wanted the number of repetitions to depend on the value of a variable? You could not look at the value and then add extra lines of code to the program while it was already executing.

Fortunately, as you have seen, Java provides a solution. The following `for` statement does what we require:

```
for (int count = 0; count < 4; count++)
{
    sam.right(); sam.right(); sam.jump(); sam.left();
}
```

The body of this loop consists of the four statements that make up the sequence of steps, and it will be executed four times.

In the workspace write code that will cause the frogs you have created to move as specified below.

2 Make the frog `sam` jump five times.

3 Make `sam` jump five times, with the value 5 stored in a variable `numberOfJumps`.

4 Move `sam` directly to the central stone. Make the frog perform the rightward dance

one step right, one step right, jump, one step left

three times, followed by the leftward dance

one step left, one step left, jump, one step right

three times.

5 Move the two frogs directly to their central stones. The frogs should then each perform the rightward sequence

one step right, one step right, jump, one step left

alternately a total of three times. The two frogs should then, alternately in the same order, perform the leftward sequence

one step left, one step left, jump, one step right

again repeating this three times in all.

Observe the Amphibians window to test that the frogs move as expected.

DISCUSSION OF ACTIVITY 12

1 The `Frog` object referenced by `sam` should perform the sequence of movements:

one step right, one step right, jump, one step left.

2 The following loop will make `sam` jump five times:

```
for (int count = 0; count < 5; count++)
{
    sam.jump();
}
```

3 The following loop will also make `sam` jump five times:

```
int numberOfJumps = 5;
for (int count = 0; count < numberOfJumps; count++)
{
    sam.jump();
}
```

Of course, if the value held in `numberOfJumps` were changed from `5`, the number of iterations would be different.

4 The following code will make `sam` move to the centre stone, then do a rightward dance followed by a leftward one.

```
sam.setPosition(6);
for (int count = 0; count < 3 ; count++)
{
    sam.right(); sam.right(); sam.jump(); sam.left();
}
for (int count = 0; count < 3 ; count++)
{
    sam.left(); sam.left(); sam.jump(); sam.right();
}
```

5 The following code will make `sam` and `lew` alternately do a rightward dance followed by a leftward one.

```
sam.setPosition(6);
lew.setPosition(6);
for (int count = 0; count < 3; count++)
{
    sam.right(); sam.right(); sam.jump(); sam.left();
    lew.right(); lew.right(); lew.jump(); lew.left();
}
for (int count = 0; count < 3; count++)
{
    sam.left(); sam.left(); sam.jump(); sam.right();
    lew.left(); lew.left(); lew.jump(); lew.right();
}
```

SAQ 12

Why would the following be an incorrect solution in step 5 of Activity 12?

```
sam.setPosition(6);
lew.setPosition(6);
for (int count = 0; count < 3; count++)
{
    sam.right(); sam.right(); sam.jump(); sam.left();
}
for (int count = 0; count < 3; count++)
{
    lew.right(); lew.right(); lew.jump(); lew.left();
}
for (int count = 0; count < 3; count++)
{
    sam.left(); sam.left(); sam.jump(); sam.right();
}
for (int count = 0; count < 3; count++)
{
    lew.left(); lew.left(); lew.jump(); lew.right();
}
```

ANSWER.....

The intention was that `sam` would perform the sequence

one step right, one step right, jump, one step left

then `lew` would do the same sequence, and so on, alternately; not that `sam` would do the sequence three times before `lew` began.

Java programming style

Good Java style normally dictates that we keep to one statement per line. However, the dance examples contain large numbers of short statements and in many cases we have felt it preferable to put several of these on one line. Our reasoning is that

```
sam.right(); sam.right(); sam.jump(); sam.left();
```

is much more compact than

```
sam.right();
sam.right();
sam.jump();
sam.left();
```

Putting the movements that form a basic dance step on the same line also makes it easier to understand the dance sequence.

ACTIVITY 13

1 In the OU workspace write a `for` loop that will output the 9 times table, using `System.out.println()`. The output should appear like this:

```
1 times 9 is 9
2 times 9 is 18
```

and so on, until

```
12 times 9 is 108
```

2 Write code that will prompt the user for a number using a request dialogue box, then output whatever times table they have chosen.

DISCUSSION OF ACTIVITY 13

1 Here is our solution:

```
for (int number = 1; number <= 12; number++)
{
    System.out.println(number + " times 9 is " + number * 9);
}
```

2 Here is our solution:

```
int tableNumber;
tableNumber =
   Integer.parseInt(OUDialog.request("Which times table do you want?"));
for (int number = 1; number <= 12; number++)
{
    System.out.println(number + " times " + tableNumber
       + " is " + number * tableNumber);
}
```

In the next activity you will consolidate your knowledge of `for` loops by writing code that requires two `for` loops, one nested inside the other.

ACTIVITY 14

In this activity, using the OUWorkspace, you will write code that will print to the Display Pane the well-known song '10 men went to mow', missing out the first verse (because it's different from the others and we want to keep things simple).

The song begins with:

2 men went to mow, went to mow a meadow,
2 men, one man and his dog,
Went to mow a meadow.

The next verse is:

3 men went to mow, went to mow a meadow,
3 men, 2 men, one man and his dog,
Went to mow a meadow.

Then we have 4 men and so on. This goes on until the last verse:

10 men went to mow, went to mow a meadow,
10 men, 9 men, 8 men, ..., 2 men, one man and his dog,
Went to mow a meadow.

To do this you will need two `for` loops, one inside the other. The outer loop should declare a variable `verseNumber` that is incremented from `2` to `10`; then in the `for` loop statement block you should first print the first line of the song:

x men went to mow, went to mow a meadow,

where x is the current value of `verseNumber`.

Then you should start the inner loop. The inner loop should declare a variable `numberOfMen` that is decremented from the current value of `verseNumber` down to `2`; then in the `for` loop statement block you should print just the beginning of the second line of the verse:

x men,

where x is the current value of `numberOfMen`. Next close the inner `for` loop and print

one man and his dog,
Went to mow a meadow.

Then finally close the outer loop.

Schematically, you will then have the following:

```
Outer loop – verseNumber increments up from 2 to 10
{
    First line of verse
    Inner loop – numberOfMen decrements down from verseNumber to 2
    {
        Second line of verse, except "one man and his dog,"
    }
    "one man and his dog,"
    "Went to mow a meadow."
}
```

You will need to use both `System.out.println()` and `System.out.print()`. These were described at the beginning of this section.

DISCUSSION OF ACTIVITY 14

Here is our version of the code:

```
for (int verseNumber = 2; verseNumber <= 10; verseNumber++)
{
    System.out.println(verseNumber + " men went to mow, went to mow a meadow,");
    for (int numberOfMen = verseNumber; numberOfMen > 1; numberOfMen- -)
    {
        System.out.print(numberOfMen + " men, ");
    }
    System.out.println("one man and his dog,");
    System.out.println("Went to mow a meadow.");
}
```

4.3 `while` loops

When the number of repetitions in a loop is not fixed at the outset, but depends on achieving some condition, the appropriate structure is a `while` statement, or `while` loop.

A Java `while` loop has the following structure:

```
while (condition)
{
    statement block
}
```

When the loop is executed:

1 `condition`, which must be a Boolean expression, is evaluated;

2 if `condition` evaluates to `false`, the iteration terminates at this point; if `condition` evaluates to `true`, the statement block (also called the body of the loop) is executed;

3 execution returns to step 1.

When the iteration terminates, the flow of program execution will pass on to whatever code follows the statement block.

As it stands, this does not explain how the loop can ever terminate. Something has to happen to make `condition` go from `true` to `false`. Very commonly the value of the condition will depend on something declared and initialised prior to the loop, which gets changed each time the body of the loop is executed, so that eventually the condition becomes false. Another possibility is that the value of the condition may depend on input from the user, or from some other source such as a file. You saw an example of this in Subsection 4.1, where a loop was terminated as soon as the user entered a number which was 100 or more.

Example

This example shows how to move a frog – whose position is assumed to be somewhere to the left of the central stone – to the central position in the Amphibians window (stone 6) by repeatedly executing the statement `frog1.right();` while the frog's position is less than 6.

```
while (frog1.getPosition() < 6)
{
    frog1.right();
}
```

The following table traces what happens if the frog is initially at position 1.

Code	Value of `frog1.getPosition()`	Value of `(frog1.getPosition() < 6)`
`(frog1.getPosition() < 6)`	1	true
`frog1.right();`	2	
`(frog1.getPosition() < 6)`	2	true
`frog1.right();`	3	
`(frog1.getPosition() < 6)`	3	true
`frog1.right();`	4	
`(frog1.getPosition() < 6)`	4	true
`frog1.right();`	5	
`(frog1.getPosition() < 6)`	5	true
`frog1.right();`	6	
`(frog1.getPosition() < 6)`	6	false

Exercise 9

By long tradition, a dance in the amphibian world begins with all the participants in the central position dressed in a red costume. Before the dance can commence, the frogs participating must move to the central stones and put on their red clothing in readiness.

Rather than moving directly to these stones, however, they hop towards the centre stone by stone. That is, they jump then move one step left or right, as appropriate, repeating this sequence until they reach the central stone. How many hops a given frog takes to reach the centre will of course depend on its starting position.

First consider the situation where a frog is to the left of the centre position when it starts. The following statement when executed will cause the frog referenced by `sam` to hop right until it reaches the central stone. The Boolean expression is evaluated to see if the frog has still not reached the centre. If it evaluates to `true`, the body of the block is executed to move the frog one stone to the right. This is carried out repeatedly while the expression continues to evaluate to `true`.

```
while (sam.getPosition() < 6)
{
    sam.jump(); sam.right();
}
```

When the frog has made the required number of hops and reached the centre, the condition `(sam.getPosition() < 6)` will become `false`, the iteration will terminate and the frog will stop hopping.

(a) The code above assumes that at the start the frog is positioned to the left of the centre. Suppose that on the contrary the frog is initially positioned to the *right* of the central stone. Write the code that is now needed to make the frog hop to the centre.

(b) How can the code that you have just written be combined with the code we gave earlier so that the frog will hop to the central stone from *any* position? (Hint: do not think too hard. The answer is quite straightforward.)

(c) Write (on paper) a method of `takeUpYourPosition()` for the `Frog` class which when sent to a `Frog` object will make it hop to the centre and put on its red costume (remember the red costume?).

Solution

(a) The code needed is:

```
while (sam.getPosition() > 6)
{
    sam.jump(); sam.left();
}
```

(b) Simply put the two one after another, like this:

```
while (sam.getPosition() < 6)
{
    sam.jump(); sam.right();
}
while (sam.getPosition() > 6)
{
    sam.jump(); sam.left();
}
```

(c) First write the initial comment and method header. Open up a pair of braces to receive the body of the method.

```
/**
 * Makes the receiver move to the central stone, hopping one stone at a time.
 * Sets its colour to red on arrival.
 */
public void takeUpYourPosition()
{
}
```

Now copy the code written in (b) into the space between the braces. Go through the code and replace `sam` everywhere by `this`, the receiver of the message corresponding to the method we are writing. Add the colour change at the end. Hey presto!
Here is the finished method:

```
/**
 * Makes the receiver move to the central stone, hopping one stone at a time.
 * Sets its colour to red on arrival.
 */
public void takeUpYourPosition()
{
    while(this.getPosition() < 6)
    {
        this.jump(); this.right();
    }
    while (this.getPosition() > 6)
    {
        this.jump(); this.left();
    }
    this.setColour(OUColour.RED);
}
```

ACTIVITY 15

Instead of using a `while` loop as in Exercise 9, you could move the frog to the central stone by working out how far to move and in what direction, and then use a `for` loop. On paper, rewrite the solution given above in part (b) of Exercise 9 using this different approach. There is no need to write an actual method; assume the code will be executed in the workspace, so that instead of `this` you should use a specific instance of `Frog` such as `sam`.

Then open Unit5_Project_2 and the OUWorkspace. From the Graphical Display menu select Open to make the Amphibians window visible. Enter your code into the workspace. Create an instance of `Frog` referenced by `sam` and test your code on it.

DISCUSSION OF ACTIVITY 15

We first set the initial position with:

```
Frog sam = new Frog();
int initial = 4;
sam.setPosition(initial);
```

You can then move `sam` to the central stone as follows:

```
if (sam.getPosition() < 6)
{
    int jumps = 6 - sam.getPosition();
    for (int count = 0; count < jumps; count++)
    {
        sam.jump(); sam.right();
    }
}
if (sam.getPosition() > 6)
{
    int jumps = sam.getPosition() - 6;
    for (int count = 0; count < jumps; count++)
    {
        sam.jump(); sam.left();
    }
}
sam.setColour(OUColour.RED);
```

Repeat the test with different values for the variable `initial`.

Exercise 10

A frog is taking part in a sponsored hop with the following rules. Beginning from whatever position the frog happens to be at when the sponsored hopping starts, it repeatedly hops one stone to the right. For each hop it makes, 10 pounds is credited to a bank account, which starts with a balance of zero. The frog continues hopping until either (a) it reaches stone 11, or (b) 80 pounds has accumulated in the account. (Of course, if the frog is already at stone 11 or beyond no money will be earned.)

Assume that the frog and the bank account have already been created, using the following statements:

```
Frog mel = new Frog();
Account acc = new Account();
```

You need not worry about the attributes `holder` and `number` of the account; in this exercise you are concerned only with the balance.

Write down the code that will cause the frog to hop rightwards according to the sponsorship rules given above.

Solution

```
while ((mel.getPosition() < 11) && (acc.getBalance() < 80))
{
    mel.jump();
    mel.right();
    acc.credit(10);
}
```

Exercise 11

A well-known puzzle goes like this. A snail starts at the bottom of a well 10 metres deep. (Presumably it fell down!) Each night the snail climbs up 3 metres towards the mouth of the well, but each day it slips back again by 2 metres. How many days will it take the snail to escape from the well?

Write a `while` loop to simulate this problem. You will need two variables, one to keep track of the snail's height and the other to record the number of days. You will also need to check inside the body of the `while` loop to see if the snail is out of the well yet. If so, it will not slip back 2 metres during the next day, because it has escaped.

Solution

```
int distanceClimbed = 0;
int days = 0;
while (distanceClimbed < 10)
{
    // Night – climb up.
    distanceClimbed = distanceClimbed + 3;
    // Escaped yet?
    if (distanceClimbed < 10)
    {
        // If not, slip back during day.
        distanceClimbed = distanceClimbed - 2;
    }
    days = days + 1;
}
System.out.println(days);
```

Endless looping

There is a potential danger with `while` loops. It is very easy to write code accidentally that repeats endlessly. If the body of the loop does not change the variable that the loop condition is testing, or does not change it in the right way, the condition may always be `true`, and the loop will carry on repeating forever. Here is an example:

```
while (frog1.getPosition() != 6)
{
    frog1.right();
    frog1.right();
}
```

Here, the programmer intended `frog1` to move repeatedly two steps to the right, until its position equalled 6. But what will happen if `frog1` sets off from an odd-numbered stone? Since it moves two steps at a time, it will always be on an odd numbered stone, 1, 3, 5, 7, ... No odd number can equal 6!

The expression `(frog1.getPosition() != 6)` will therefore *always* be true, and the frog will continue moving two steps right forever. This is not what was intended.

On the other hand there can be situations where we actually *require* a program to run forever. In Java a common way of coding this is:

For instance, an airline reservation system might be intended to run for 24 hours a day, never stopping.

```
while (true)
```

When the test expression `true` is evaluated the result is invariably `true`, so the loop will run endlessly.

SAQ 13

What would happen if you set the position of `frog1` to `3` before executing the following expression?

```
while (frog1.getPosition() != 6)
{
    frog1.left();
}
```

ANSWER.....

The statement forming the body of the loop would continue to decrease the position of `frog1` (that is, to 2, 1, 0, –1, –2, ...). Hence the Boolean condition would never return `false`. This means that the loop would be executed forever!

Exercise 12

Using pen and paper, write a new method for the `HoverFrog` class, with the method signature `moveTo(int)`, which first sets the `height` of the receiver to `0` and then uses a `while` loop to move it to the height specified by the argument, increasing the height in steps of 1.

Solution

```
/**
 * Sets the height of the receiver to 0 and then
 * moves it to the height specified by the argument,
 * increasing the height in steps of 1 to aHeight.
 */
public void moveTo(int aHeight)
{
    this.setHeight(0);
    while (this.getHeight() != aHeight)
    {
        this.up();
    }
}
```

Note that if you sent the message `moveTo()`, as coded in Exercise 12, with an argument value outside the range `0` to `6` you would find that it looped for ever. Since it is impossible to move hoverfrogs outside this range, the condition `this.getHeight() != aHeight` will always be `true`. The method could be made safe from this risk by adding an extra test.

```
public void moveTo(int aHeight)
{
    if ((aHeight >= 0) && (aHeight <= 6)) //extra test
    {
        this.setHeight(0);
        while (this.getHeight() != aHeight)
        {
            this.up();
        }
    }
}
```

ACTIVITY 16

Open Unit5_Project_1 and the OUWorkspace.

Imagine a user saving money in a bank account, represented by an instance of the `Account` class referenced by `herAccount`. The user's savings target is 500 pounds. In the workspace, create a new instance of `Account` with

```
Account herAccount = new Account();
```

Here is the specification of the code we would like you to write. Please do not start coding until you have read the specification *and* the advice that follows.

You are asked to write code which uses a `while` loop to repeatedly check the balance of `herAccount` to see if 500 pounds has been reached yet. If this target has not yet been achieved, a request dialogue box is to be displayed asking the user how much they want to pay in. The amount they enter (which should be a whole number not a decimal) is to be added to the balance of `herAccount`. As soon as the balance of `herAccount` is at least 500 pounds, the iteration should cease and the message 'Congratulations, you have met your target!' should be displayed in an alert dialogue box.

We suggest you work as follows.

1 In the workspace type an empty block, as follows. Do *not* include the first line with the `while` in at this point; you will add that in a minute.

```
{
}
```

We know this code is not very exciting so far, but bear with us!

Now inside this block add code that will prompt the user for the amount of the deposit and add this to the balance of the account (remember to use `Integer.parseInt()` to convert the string returned by the dialogue box into an integer). You should also use `credit()` to increase the balance of `herAccount`.

Look at our solution if you need to.

Once this block of code is written, *test it.* It should put up a dialogue box and then add whatever figure you enter to the balance of `herAccount`. You can check the balance by inspecting `herAccount`. By doing this test you are making sure that the code in the body of the loop works correctly. You can see that unless this code is right in the first place there is no point in repeating it over and over again in a loop!

If it does not work, then go back and try to correct it.

The beauty of doing things this way is two fold:

(a) You break the problem into more manageable chunks.

(b) You know before you add the `while` that the account balance is getting updated correctly – so the chances of accidentally writing an endless loop are greatly reduced.

2 Now add the `while` with the appropriate condition in front of the block, to construct the complete `while` statement. Remember the alert dialogue box that was specified; that goes at the end.

Execute your code to verify that it works to specification.

DISCUSSION OF ACTIVITY 16

1 This is our version of the code that should go inside the block:

```
int inputAmount = Integer.parseInt(
    OUDialog.request("How much do you want to deposit?"));
herAccount.credit(inputAmount);
```

2 This is the whole thing:

```
Account herAccount = new Account();
while (herAccount.getBalance() < 500)
{
    int inputAmount = Integer.parseInt(
        OUDialog.request("How much do you want to deposit?"));
    herAccount.credit(inputAmount);
}
OUDialog.alert("Congratulations, you have met your target!");
```

Alternatively, you could have written the code without a variable to hold the amount input:

```
Account herAccount = new Account();
while (herAccount.getBalance() < 500)
{
    herAccount.credit(Integer.parseInt(
        OUDialog.request("How much do you want to deposit?")));
}
OUDialog.alert("Congratulations, you have met your target!");
```

5 Summary

After studying this unit you should understand the following ideas.

- Classes can define class methods in addition to instance methods.
- The execution of a class method does not involve a message-send to an object.
- Dialogue boxes are a means of obtaining input from, and displaying output to, the user. In this course dialogue boxes are created using class methods of the `OUDialog` class.
- The class methods `Integer.parseInt()` and `String.valueOf()` can be used to convert a string to a number and vice versa.
- A sequence of statements can be made into a statement block by enclosing it in braces.
- An `if-then` statement is used to determine whether or not to execute a statement block depending on a Boolean condition which evaluates to `true` or `false`.
- An `if-then-else` statement is used to select between two alternative statement blocks depending a Boolean condition. If the condition evaluates to `true` the `then` statement block is executed, otherwise the `else` statement block is executed.
- The operators `==` `!=` `<` `<=` `>` `>=` are used for comparing values from primitive data types.
- The operator `==` can also be applied to objects, in which case it tests whether two variables reference exactly the same area in memory, that is, whether they reference the same object. Similarly `!=` tests whether its operands reference different objects.
- The message `equals()` can be used to test if two strings contain the same characters in the same order.
- More complicated Boolean expressions can be composed by using the logical (Boolean) operators `&&`, `||` and `!`.
- The operators `&&` and `||` use short-circuit evaluation.
- Java statement blocks can be executed repeatedly using `for` and `while` loops as control structures. The number of iterations is controlled by a condition.

LEARNING OUTCOMES

After studying this unit you should be able to:

- invoke class methods in the class `OUDialog` to create dialogue boxes to display output to the user or to gain input from the user;
- write Java statements which convert string values to number values and vice versa;
- use `if` statements to select alternative statement blocks for execution depending on a condition;
- construct more complex conditions using comparison and logical operators;
- use the Java `for` statement to execute a statement block a fixed number of times;
- use the Java `while` statement to execute a statement block a variable number of times depending on some condition.

Glossary

block See **statement block**.

Boolean condition A **Boolean expression** used to control the conditional execution of a **statement block**.

Boolean expression An expression that evaluates to either `true` or `false`. Boolean expressions can be simple or complex and can involve a number of variables.

Boolean operators Operators used to combine simple **Boolean expressions** to form more complex Boolean expressions, which in turn can be combined with other Boolean expressions. They are also known as logical operators.

class method **A method** that is executed as the result of an invocation (not by sending a message to an *instance* of a class). Class methods in Java are specified by including the `static` modifier in the **method header**.

condition See **Boolean condition**.

conditional selection The use of `if` statements to select and execute alternative statement blocks based upon the value of a **Boolean condition**.

dialogue box A mechanism whereby users can be given information by the system or provide information to the system on request. Dialogue boxes are implemented by **class methods** of `OUDialog`. Examples of these methods are:

```
alert(String prompt)
confirm(String prompt)
request(String prompt)
request(String prompt, String initialAnswer)
```

iteration Also referred to as repetition. The repeated execution of a statement block for as long as some **condition** continues to be true.

statement block A statement or sequence of statements 'bundled together' for use in a particular context. Any sequence of statements can be turned into a block by enclosing it in braces.

`static` A Java keyword which defines a variable or method as belonging to a class rather than its instances.

Index

A
`alert` 8
and 34
B
binary operators 33
Boolean
  condition 21, 24
  expression 33
  operators 34
C
Cancel button 15, 31
class method 7
comparison operators 33
concatenation operator 17
condition 21
conditional selection 5
`confirm` 12
D
dialogue boxes 6
E
equality operators 33
equal to 33
F
`for` loop 44
G
greater than 33
greater than or equal to 33
I
`if-then` statement 21
`if-then-else` statement 21
input box 14–15
`Integer` 18
iteration 5
L
lazy evaluation 40
less than 33
less than or equal to 33
local variable 12
M
modal dialogue box 6
N
No button 13
not 34
not equal to 33
`null` 15
O
or 34
`OUDialog` 6
P
parentheses 33
`parseInt` 18
R
readability 14
relational operators 33
`request` 14
S
short-circuit evaluation 40
statement block 22
`static` 7
T
`toUpperCase` 9
U
unary operator 33
V
`valueOf` 18
W
`while loop` 52
Y
Yes button 13