M255 Unit 2
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Object concepts

Unit 2

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 5494 0

2.1

CONTENTS

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

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

Introduction

This unit builds upon the object concepts introduced in *Unit 1*.

Here we will look further at messages and discuss:

- how an object's response to a message may depend on the state of its attributes;
- how some messages need extra information in the form of arguments;
- how some messages change the state of the receiver, while others simply return a message answer – often the value of one of the receiver's attributes;
- polymorphic messages – messages that objects of more than one class may respond to, but that might yield different behaviour depending on the class of the receiver;
- object collaboration – how, once an object is sent a message, that object may then need to send messages to another object(s) to help it carry out the behaviour associated with that message.

We also introduce the **superclass/subclass** relationship between classes, whereby a subclass inherits the protocol and attributes of its superclass but then defines additional attributes and behaviour for its instances. This superclass/subclass relationship is *central* to object-oriented programming and results in robust and economical code.

Much of your work in this unit will involve practical activities and you will be using a variety of microworlds from the Amphibian Worlds application introduced in *Unit 1*. However, in addition to `Frog` objects, these microworlds are also inhabited by `HoverFrog` and `Toad` objects.

In Section 5 you will also use a simple programming tool, called Accounts World, which has been developed specifically for this unit. In addition to allowing you to write Java code this programming tool will enable you to create objects for the first time (`Account` objects).

All of the concepts introduced in this unit will be revisited and explored from different points of view in subsequent units.

SAQ 1

What description is given in *Unit 1* for an object?

ANSWER.....

Unit 1 describes an object as a self-contained unit of software that holds data and knows how to process that data. Each object is able to communicate with other objects via messages.

1 Classes and protocols

This section explores the ideas of *class* and *protocol*, which were introduced in *Unit 1*. We also look at the *initialisation* of an object. The term *polymorphism* is defined.

In *Unit 1* you looked at a simple microworld called Two Frogs. In this unit you will explore a whole series of similar worlds, all involving various kinds of amphibian. As we go on we shall gradually introduce additional features designed to illustrate important ideas in object-oriented programming. This may give the impression that the classes concerned are changing; in fact it is simply that the full protocol of the class is only being revealed in stages.

1.1 Frogs and toads

For the next activity you will be using the Amphibian Worlds application and, in particular, its Two Frogs and Toad microworld. This has a new species – toads – represented by **instances** of a new class, `Toad`. In the microworld, the icons you see are, of course, a graphical representation of `Frog` and `Toad` objects. The visual representation of a `Toad` object in the Graphics Pane is slightly different from that of a `Frog` object; this reflects the way that the program designer has chosen to portray these objects in the user interface. It is a cosmetic difference that emphasises the different behaviours of `Frog` and `Toad` objects. (In a different graphical interface this particular difference in appearance might not be used.)

In Activities 1 and 2 you will be thinking about the class of the `Toad` objects and the class of the `Frog` objects.

ACTIVITY 1

Launch the Amphibian Worlds application from your desktop and choose Two Frogs and Toad from the Microworld menu.

1 You will see three objects, referenced by the variables `frog3`, `frog4` and `toad1`. For each object, what are the initial values of the attributes `position` and `colour`? (Another way of phrasing this is to ask how each object has been *initialised*.)

2 Does `toad1` respond to the message `jump()`? Does `toad1` behave in the same way as the `Frog` objects (`frog3` and `frog4`)? To find out, select `toad1` and send it some messages. You can do this by clicking the message buttons or writing the Java code in the Code Pane. Try sending the same messages to one of the `Frog` objects and to `toad1`. Note particularly how the `Frog` objects and `toad1` respond to the messages `left()`, `right()` and `home()`.

3 What evidence is there that the objects `toad1`, `frog3` and `frog4` belong to different classes?

DISCUSSION OF ACTIVITY 1

1 When the microworld is first opened, `toad1` is brown and is in position 11, whereas both `frog3` and `frog4` are green and in position 1. In other words, `Frog` objects are initialised with position `1` and colour `GREEN`, but the object `toad1` is initialised with position `11` and colour `BROWN`.

2 You saw in *Unit 1* that the objects referenced by `frog1` and `frog2` behave identically in response to the same messages – they are instances of the same class, `Frog`. The indications that the object referenced by `toad1` and the `Frog` objects may belong to different classes lie in the observations that the two kinds of object behave differently in response to the messages `left()`, `right()` and `home()`, and that only the `Frog` objects respond to the message `jump()`.

   In response to the messages `left()` and `right()`, the `Frog` objects move position by one unit, whereas the object referenced by `toad1` moves position by two units. The object referenced by `toad1` does not respond to the message `jump()`, whereas the two `Frog` objects jump and land in their original position. On this evidence we suspect that `toad1` does not belong to the `Frog` class.

3 The differences observed between `toad1` and the two `Frog` objects are not conclusive evidence that they belong to different classes, but they are sufficient for us to suspect that they do. The implementation of this Amphibian world is, however, that `toad1` belongs to the `Toad` class whereas `frog3` and `frog4` belong to the `Frog` class.

`Frog` and `Toad` objects look slightly different, as the Graphics Pane has been designed to display them differently in order to help the viewer. Their visual representation is distinct from their state. You can tell this, as opening an **Inspector** window on either a `Frog` or a `Toad` object shows that there is no attribute that records what icon should be used to represent it. Unlike the colour and position, the icon used in the Graphics Pane is not part of the object's state.

ACTIVITY 2

The protocol of instances of a class is the set of messages these instances (objects) understand.

In this activity you will once again be using the Two Frogs and Toad microworld in the Amphibian Worlds application.

1 Determine as much of the protocol of `Toad` objects as you can by sending each message to the object referenced by `toad1` using the buttons in the microworld. How does this compare with the protocol of `Frog` objects?

2 Select `toad1` in the scrollable pane and click on the Inspect button. An **Inspector** window will open on `toad1`. What attributes has the object referenced by `toad1`, and how do these compare with those of the `Frog` objects?

Close any open Inspector windows before proceeding.

DISCUSSION OF ACTIVITY 2

1 When you select `toad1` and send it in turn each of the messages given by the buttons in the microworld, you find that it behaves in the following ways in response to the following messages.

   `left()` – moves *two* positions to the left

   `right()` – moves *two* positions to the right

   `home()` – moves to (or remains on) the rightmost black 'stone'

   `green()` – turns green (unless already green)

   `brown()` – turns brown (unless already brown)

   `croak()` – croaks audibly (and displays a red '!')

   When any of the messages `up()`, `down()` or `jump()` is sent to a `Toad` object (`toad1`), an error report appears in the Display Pane saying that the object cannot

respond to the message. The messages `up()`, `down()` and `jump()` are not in the protocol of `Toad` objects.

As already noted, `Frog` and `Toad` objects respond to the messages `left()`, `right()` and `home()` in different ways. Neither `Frog` nor `Toad` objects respond to the messages `up()` or `down()`. The `Frog` objects respond to the message `jump()`, but the `Toad` object does not. From the messages we know about, the protocol for the `Toad` object contains only `left()`, `right()`, `home()`, `green()`, `brown()` and `croak()`.

Although the `Toad` and `Frog` objects belong to different classes, behaving differently in response to the messages `left()`, `right()` and `home()`, this does not prevent them from behaving identically in response to some messages, e.g. `green()` and `brown()`.

2 The information displayed in the Inspector window that opens on `toad1` when you press the Inspect button after highlighting the variable `toad1` shows that `toad1` has the same attributes as the `Frog` objects – `position` and `colour`.

Initial state of an object

When the microworld Two Frogs and Toad is opened, it creates one `Toad` and two `Frog` objects, each with appropriate state. You saw in the Graphics Pane that the `Toad` object icon was a different `colour` and in a different initial `position` from the `Frog` object icons – and each icon reflects the state of the corresponding software object. We say that the `Toad` and `Frog` objects are **initialised** differently, i.e. their states are different when they are newly created. It is useful to regard a class as something that provides the 'template' for each of its instances.

1.2 Objects of different classes

You have seen that `Frog` and `Toad` objects do not have the same protocol, since `Toad` objects do not respond to the message `jump()`. Both `Frog` and `Toad` objects respond to the messages `left()`, `right()`, `home()`, `green()`, `brown()` and `croak()`. In the cases of the messages `green()` and `brown()` the **behaviours** are the same. However, you found that although both `Frog` and `Toad` objects respond to the messages `left()`, `right()` and `home()`, their behaviours are different. How a message is interpreted depends on the class to which the object that receives the message belongs.

In inspecting the `Frog` and `Toad` objects, you have seen that the variables `frog3` and `frog4` reference instances of the class `Frog` and that `toad1` references an instance of the class `Toad`. It is important to distinguish between an instance of a class (for example, `frog1`) and its class (in this case, `Frog`). The class is the 'factory' for creating instances of the class.

Although `Frog` and `Toad` objects have the same attributes, they have different yet overlapping protocols, and behave in a different way in response to some of the same messages. From our discussion of classes in the previous unit, these differences indicate that `Frog` and `Toad` objects belong to different classes.

1.3 Classes and writing software

Adopting the idea of **classes** can save a programmer work. One of the goals of software development is to avoid replicating the same code over and over again.

When building object-oriented software, the programmer may have to specify the behaviour of many objects that must collaborate to make some system or program work (for example, the character objects in a word-processed document). In particular, it will be necessary to specify somehow the attributes that each object will have and the messages to which each object will respond.

It would be perfectly possible to build every object afresh from the ground up, by providing all the code necessary to specify the attributes and behaviour – but, when a program can involve thousands of objects, this would be unbearably tedious. Moreover, we would probably make many errors, and every individual object would need to be tested separately to make sure it was correct. So we look for a way to save effort by writing code only once and reusing it. This is what classes do for us. In any given program you find very often a group of objects that are essentially the same. For example, in a payroll program you would expect to find many objects representing employees. In a drawing program there might be many objects that represent rectangles. When writing a program, whenever you identify that a set of objects all belong to a common classification, with the same attributes and behaviour, you can define a template for this kind of object. The attributes and behaviour will need to be specified just once, in the class definition, and then you can use the class to generate an object of that particular kind whenever you like.

Then, however many instances of that class are created, each will automatically have the same attributes and respond to the protocol for the class in the same way, with no additional work on the part of the programmer.

Moreover, once you have tested your code and know it is correct you can rely on all the objects belonging to that class working as intended; you do not need to test them individually.

Of course, different instances of the same class may acquire different states (for example, `frog1` may have its `colour` attribute set to `GREEN`, while `frog2` has the value of this attribute as `BROWN`.

If the attributes or any part of the protocol needs to be changed, the change need be made only once – to the class. The change will automatically apply to all instances of the class.

The idea of class is a key part of object-oriented software development. Identifying common properties at the planning stage, so that objects can be grouped appropriately into classes, enables programs to be written in a concise and economical way.

1.4 Polymorphism

You have seen that `Frog` objects and `Toad` objects respond to some messages with the same name, even though they behave differently in response to some of these messages. In fact, it is unusual for two classes to have such similar protocols as these `Frog` and `Toad` objects. There are typically major differences in the protocols of any two chosen classes. However, it is not unusual for a particular message to be in the protocol of more than one class. Thus while different classes rarely share exactly the same protocol, it is not unusual for protocols to overlap. Such overlaps can be very useful. To take an example, in the graphical interface in the microworld, the nearly complete overlap of the protocols of the `Frog` and `Toad` objects means that they can share the same buttons.

In a word processor it might be helpful if the user could double the size of items such as characters, words, pictures and paragraphs by using one key. A sensible way of organising this in object-oriented software is to have character objects, word objects,

picture objects and paragraph objects (instances of different classes) all understand the message `doubleSize()`.

There is a special term for a message to which objects of more than class can respond. We say that the message is *polymorphic*. There is no need for the response to be the same for the different classes; in fact it probably will not be. For example, `Frog` and `Toad` objects react in their own different ways to the message `home()`. Polymorphic messages are very common in object-oriented programming. You will see later that polymorphism is very useful. Any message to which objects of more than one class can respond is said to be polymorphic or to show **polymorphism**.

Imagine that all the staff in a company have gone to a general staff meeting. At the end of the meeting, the speaker says: "Thank you everyone for coming. Now I'll let you get back to whatever you have to do next."

The members of the audience will all understand this and they will each know what to do next. Customer Services staff will go back to dealing with customers, Accounts staff will resume keeping track of accounts, Deliveries staff will continue where they left off arranging deliveries, and so on.

Everyone will respond to the same instruction, but they will respond differently, according to what department they come from. The speaker does not have to ask each person where they work and then tell them what to do; the speaker does not need to know how to do the jobs, or even what the various jobs *are*. Each member of the audience will know for themselves what to do, without being told.

SAQ 2

In your own words, what does it mean to say that an instance of a class is initialised?

ANSWER.....

Instances of a class have attributes. It is usual for a newly created object to have its attributes initialised to some initial values. For example, the instances of the class `Frog` have the attributes `colour` and `position`. The instances of the class `Frog` are initialised so that the value of `colour` is `GREEN` and the value of `position` is `1`.

SAQ 3

In a certain word processor, characters, words, paragraphs and pictures all understand the message `doubleSize()`, although these objects are instances of different classes. What word is used to describe this kind of situation?

ANSWER.....

The message `doubleSize()` is polymorphic with respect to instances of the (hypothetical) classes `Character`, `Word`, `Picture` and `Paragraph`.

SAQ 4

Explain the word polymorphism, using an example from the `Toad` and `Frog` classes as an illustration.

ANSWER.....

A polymorphic message is a message to which objects of more than one class can respond.

The message `left()` is polymorphic as it can be sent to instances of different classes. All instances of the class `Frog` interpret the message `left()` by subtracting 1 from the current value of their `position`, whereas instances of the class `Toad` subtract 2 from the

current value. The message `green()` is also polymorphic. It is understood by instances of more than one class, for example, `Toad` and `Frog` objects. The behaviour caused by sending `green()` to either a frog or a toad is the same.

It is considered good style to give a message a descriptive name. Although the computer would not care if you used an arbitrary name such as `messageF9B()`, human beings who want to understand the program find meaningful names extremely helpful! The message name `left()`, for example, is descriptive of what it does.

2 Classes and subclasses

We now explore classes and subclasses. We also introduce the term superclass. An object's response to a message sometimes depends on its state when the message is received; this state-dependent behaviour is illustrated in Activity 4.

Hoverfrogs

In this section you will meet a new species of amphibian – the hoverfrog.

ACTIVITY 3

Launch the Amphibian Worlds application and then choose HoverFrogs from the Microworld menu. In this microworld there are two `HoverFrog` objects, referenced by the variables `hoverFrog1` and `hoverFrog2`. (You will see a rotor blade on the head of the graphical representation of each `HoverFrog` object.)

You will now explore the messages to which `HoverFrog` objects respond, and how they behave in response to these messages. The results you obtain will help you to decide how `HoverFrog` objects are related to `Frog` objects.

1 Do `HoverFrog` objects respond to the same messages and behave in the same way as `Frog` or `Toad` objects? To find out, send some messages to the `HoverFrog` objects and observe the effects.

2 What is the protocol of the `HoverFrog` objects (to the extent shown in this microworld)? Describe the resulting behaviour for each message in the protocol.

DISCUSSION OF ACTIVITY 3

1 A `HoverFrog` object can respond to the same messages to which `Frog` and `Toad` objects can respond, and it responds to all of these messages in the same way as a `Frog` object. In *addition*, a `HoverFrog` object can respond to messages to which a `Frog` object cannot respond, namely `up()` and `down()`.

2 The protocol for the `HoverFrog` objects as shown in the microworld HoverFrogs is `left()`, `right()`, `home()`, `up()`, `down()`, `jump()`, `green()`, `brown()` and `croak()`. The usual behaviour of the `HoverFrog` objects in response to each of these messages is shown below.

`left()` – moves one position to the left

`right()` – moves one position to the right

`home()` – moves to (or remains on) the leftmost black 'stone' (which then turns yellow)

`up()` – moves up by one on the six 'steps' above the 'stone'

`down()` – moves down by one on the six 'steps' above the 'stone'

`jump()` – jumps and lands again on the same 'stone'

`green()` – turns green (unless already green)

`brown()` – turns brown (unless already brown)

`croak()` – croaks audibly (and displays a red '!')

In the discussion of Activity 3, the descriptions of the behaviour of a `HoverFrog` object in response to the messages `up()`, `down()` and `jump()` were incomplete. The discussion took no account of the fact that the response to these messages varies according to the state of the `HoverFrog` object when it receives the message. Exploration of state-dependent behaviour forms the basis of the next activity.

ACTIVITY 4

This activity explores how the behaviour of an object sometimes varies according to the state it has when it receives a message.

Launch the Amphibian Worlds application and then open the microworld HoverFrogs. Experiment with the `up()` and `down()` messages – send each of the messages `up()` and `down()` in succession to the same object and note the object's behaviour. Then expand the descriptions given in the discussion of Activity 3 for the behaviour of a `HoverFrog` object in response to these messages, taking into account the way that this depends on the **receiver's** state.

In the same way, give an improved description of the behaviour of a `HoverFrog` object in response to the message `jump()` that takes account of its dependency on the receiver's state.

Can you guess an attribute that `HoverFrog` objects possess that `Frog` and `Toad` objects do not? Make a guess before being tempted to look at the attributes of a `HoverFrog` object with an inspector.

Select a variable that references a `HoverFrog` object and press the Inspect button. What are the attributes of a `HoverFrog` object?

Send the `up()` and `down()` messages to a `HoverFrog` object, inspecting its state after each message. What sort of values can be held by the attribute that is changing in response to these messages? Is a `HoverFrog` object initialised identically to a `Frog` object?

DISCUSSION OF ACTIVITY 4

What is missing in the discussion of Activity 3 is an account of the behaviour of a `HoverFrog` object when it is at its minimum height (on a stone) and the message `down()` is sent to it, and when it is at its maximum height and the message `up()` is sent to it. In both cases no visible action results. Also, when a `HoverFrog` object is not at its minimum height the message `jump()` has no visible effect.

`up()` – moves up by one on the six 'steps' above the 'stone' (unless already at maximum height)

`down()` – moves down by one on the six 'steps' above the 'stone' (unless already at minimum height)

`jump()` – jumps and lands again on the same 'stone' (provided it is at minimum height)

`HoverFrog` objects have attributes `colour`, `position` and `height`.

By experimenting with a `HoverFrog` object and using the inspector you should have discovered that the attribute `height` only ever holds integer values of `0` to `6`.

In the Graphics Pane, `HoverFrog` objects appear to be initialised identically to the `Frog` objects. However, this is not strictly correct, as a `HoverFrog` object has an additional attribute, `height`, which is initialised to `0`.

State-dependent behaviour

You have seen that the way that an object responds to a message may not depend just on its class. It may also depend on its state. In other words, the behaviour of an object in response to a message may be **state-dependent**.This can be illustrated by sending the message `up()` or `down()` to a `HoverFrog` object. In the microworld, a `HoverFrog` object icon in the Graphics Pane does not reflect a response to any message requesting it to go higher than 'step 6' or lower than 'step 0'. This is because the messages that affect the attribute `height` can only give it integer values from `0` to `6`. So any `up()` or `down()` message to a `HoverFrog` object that attempted to set the value of `height` beyond these limits left the value unchanged.

2.2 Subclasses

In our particular Amphibian world, `Frog` and `Toad` objects have to 'remember' only two things that are changeable: their colour and their position. Hence they have only the two attributes, `colour` and `position`, that we considered earlier. In order to be able to respond sensibly to the `up()` and `down()` messages, a `HoverFrog` object needs an extra attribute, `height`. Initialisation of `HoverFrog` objects is also different in that the `height` attribute is set to `0`.

A `HoverFrog` object can respond to all the messages that a `Frog` object can respond to, and it behaves in exactly the same way as a `Frog` object in response to these messages. However, it can also respond to extra messages to which a `Frog` object cannot respond, namely `up()` and `down()`. Furthermore, as noted above, `HoverFrog` objects have all the attributes of `Frog` objects and an additional attribute `height`.

It is clear that there is a relationship between `HoverFrog` objects and `Frog` objects. A `HoverFrog` object can do what a `Frog` object can do, but more, it has all the same attributes, and more; and this is true for all instances of the `Frog` and `HoverFrog` classes. In fact, the programmer took the `Frog` class as a basis and added the extra attribute and protocol required to define the `HoverFrog` class. The relationship between two such classes is described by saying that the `HoverFrog` class is a **subclass** of the `Frog` class, and the `Frog` class is the **superclass** of the `HoverFrog` class. The fact that the `HoverFrog` class is a subclass of the `Frog` class does not mean that a `HoverFrog` object is 'less' than a `Frog` object. The term 'subclass' indicates that the subclass is derived from the superclass. A `HoverFrog` object has at least the attributes and protocol of a `Frog` object.

A subclass may have a different initialisation from its superclass. It may seem as if `Frog` and `HoverFrog` objects have the same initialisation, but this is not strictly true as `HoverFrog` objects have an additional attribute that has to be given an initial value.

The relationship between the `Frog` and `HoverFrog` classes is summarised below.

(a) The protocol of `HoverFrog` objects includes that of `Frog` objects.

(b) `HoverFrog` and `Frog` objects respond in the same way to the messages common to their protocols.

(c) Instances of the `HoverFrog` class have the attributes of instances of the `Frog` class.

(d) `HoverFrog` objects have an additional attribute.

(e) There are messages in the protocol of `HoverFrog` objects that are not in the protocol of `Frog` objects.

(f) The common attributes of the instances of both classes are initialised in the same way.

Not all the points listed above are a necessary part of the superclass/subclass relationship. In later units you will see that it is possible to program a subclass so that its attributes are initialised differently from the corresponding ones in the superclass. You will also see that instances of a subclass may respond to a message in a way that is different from the way instances of the superclass would respond to that message.

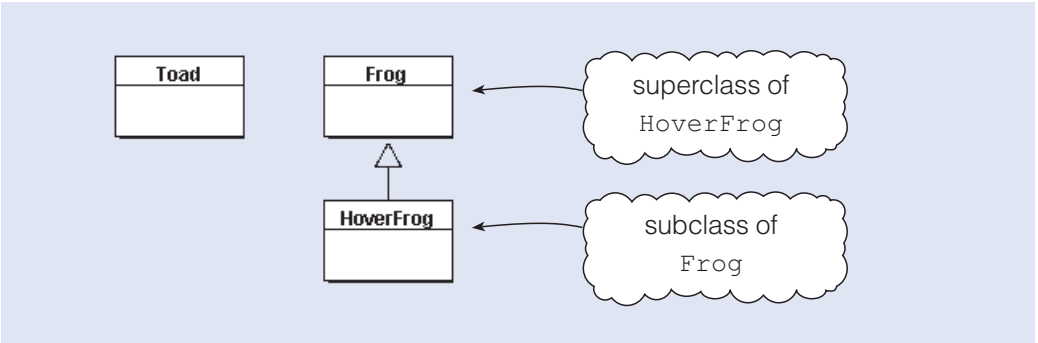

Figure 1 Relationship between the **Frog**, **HoverFrog** and **Toad** classes

The Frog class cannot be a superclass of the Toad class, because instances of Toad cannot respond to jump() messages. So could the Toad class be a superclass of the Frog class? Well frogs and toads behave differently in response to left() and right() messages – in the light of the above discussion, the situation where the Toad class is the superclass of Frog is not precluded but would not be good practice (you will learn more about this in *Unit 6*).

Our use of subclasses should reflect our usual views of the way objects relate: whilst it would be reasonable to say a hoverfrog *is a* frog we would not normally say a frog *is a* toad or vice versa.

Why is this class/subclass relationship important? Frog objects and HoverFrog objects have aspects of their protocol and attributes in common. As you will see in the next subsection, you need to define these common aspects just once – for the superclass (Frog). Then, the fact that HoverFrog is a subclass of Frog automatically ensures that HoverFrog objects also have this common behaviour.

2.3 Programming and subclasses

Using subclasses when building programs enables you to save on programming effort. When building object-oriented software, the programmer has to specify somehow what the objects of each class will be able to do. But sometimes it turns out that the objects of some prospective class (the HoverFrog objects in our example) need to be able to do everything that objects of some existing class can already do (the Frog objects). In addition, objects of the prospective class may need to be able to respond to some extra messages (up() and down()), and may have extra attributes (height). In such a case there is a simple way to save work: the programmer can declare the new class to be a subclass of the existing class (HoverFrog as a subclass of Frog). This ensures that objects of the new class (HoverFrog) have, as a minimum, the same attributes and protocol as the existing class (Frog). This will happen automatically as a consequence of declaring the new class (HoverFrog) to be a subclass of the existing class (Frog). The subclass HoverFrog is said to *inherit* these attributes and this protocol from the superclass Frog. All the programmer needs to do subsequently is to determine any additional attributes (height) and messages (up() and down()) for the subclass.

SAQ 5

Which word or phrase best fits in the following sentence?

The protocol of a HoverFrog object ... the protocol of a Frog object.

(a) is part of

(b) is similar to

(c) includes

ANSWER

The missing word is includes. (`HoverFrog` is a subclass of the class `Frog`. The instance protocol of `HoverFrog` includes the protocol of `Frog`. As you have seen, `HoverFrog` objects understand some additional messages to which `Frog` objects do not respond.)

SAQ 6

What criteria do you think the programmer has applied in deciding to create the class `HoverFrog` as a subclass of the class `Frog`?

ANSWER

The programmer required the instances of the class `HoverFrog` to have all the attributes of the class `Frog`, to respond to all the messages of the instances of the class `Frog`, and to behave in an identical way in response to most of these messages.

3 Message arguments

This section introduces message arguments, which allow the sender to include information in messages.

Messages and arguments

Messages become a much more powerful mechanism when you allow them to include extra information. For example, previously there were separate messages for turning an object green or brown. A message that sets the colour of an object and allows you to state the colour required allows a much more general approach. When a message requires extra information for it to make sense, each required piece of information is called an **argument** of the message.

In our microworlds we have made it possible for you to send messages with arguments through the use of menus. When some more information is required from the user after a button is pressed, a menu will be presented. You must choose an item from the menu before the message triggered by the button can be sent. In Activity 5 you will meet the setColour() button that presents a menu of colours from which one is chosen as the message argument. The use of an argument makes available a wider choice of colours than previously.

ACTIVITY 5

From the Amphibian Worlds application open the microworld Frog & HoverFrog. This microworld contains three new buttons: setColour(), upBy() and downBy(). These buttons are *menu buttons* – in each case a menu will be displayed when the button is pressed. You need to make a choice from the menu. (Remember to highlight a variable referencing a frog or hoverfrog before sending a message.)

1 Explore the setColour() button by using it to send messages to the `Frog` and `HoverFrog` objects.

   Do objects of both classes respond to `setColour()`? Which colours are available?

2 Send messages using the upBy() and downBy() buttons.

   Do both kinds of object respond to a press of these buttons? What information do you have to supply before an object will hover?

3 On the screen you can see more of the protocol of the `Frog` objects. What is the name of the additional message that has been revealed in this activity? What is the protocol of `Frog` objects, as revealed in this and previous activities?

DISCUSSION OF ACTIVITY 5

1 Both the `HoverFrog` and `Frog` objects understand the message sent by pressing the setColour() button. Before you can send a message using one of the upBy(), downBy(), or setColour() menu buttons, you have to provide some information.

   The menu for the setColour() button offers a choice of colours: OUColour.BLUE, OUColour.BROWN, OUColour.GREEN, OUColour.PURPLE, OUColour.RED and OUColour.YELLOW.

2 Only `HoverFrog` objects respond to a press of the upBy() or downBy() buttons.

The menus for the upBy() and downBy() buttons offer a choice of 1, 2, ..., 6. These determine how many 'steps' up or down you want a `HoverFrog` object to move.

3 The protocol of the `Frog` objects is the set of messages that the objects understand. You can now see that the message name `setColour()` is in the protocol of `Frog` objects. The protocol of `Frog` objects as revealed in this activity is `left()`, `right()`, `home()`, `jump()`, `green()`, `brown()`, `croak()` and `setColour()`. `Frog` objects do not know how to respond to messages for hovering.

Some books call an argument a **parameter**.

Each piece of information you supplied is called a *message argument* (or just *argument*). When an argument is required for a message that a button sends, a menu is presented.

The equivalent Java code of selecting the button setColour() and choosing OUColour.RED has the textual form `setColour(OUColour.RED)`. You supply extra information between the brackets to form the message. Some messages always require information to be specified before they can be sent. It is not possible to end such a message without providing the information. The information (here a chosen colour) forms an integral part of the message.

You will learn about class variables in *Unit 7*.

You may have been wondering what arguments such as `OUColour.GREEN` and `OUColour.RED` are, exactly. Well, we have created a set of `OUColour` objects for you to use. These objects are held by the `OUColour` class as *class variables*. All you need to know now is that to get hold of one of these ready-made colours, you just type the name of the class, followed by a full stop and then the colour you want (in capitals): for example, `OUColour.YELLOW`.

3.2 Message names

We have said that when a message requires an argument, the message is incomplete until an argument is chosen. Thus `left()` is a message, whereas `setColour()` is not a message until an argument is provided. That means that it is not accurate to talk about 'the message `setColour()`', since this will not be a message until you pick a particular argument. But what if you want to talk about the entire family of messages represented by `setColour()`, irrespective of the choice of a particular argument? It would be helpful to have words to make this distinction.

The term **message name** is used for this purpose. For example, the message name for the message `setColour(OUColour.RED)` is `setColour()`.

The phrase 'message name' is also used with a message that does not take an argument. In this case, the distinction between message and message name is less obviously useful, but the distinction still exists. So, for example, you might say either of the following.

'`Frog` objects understand the message `left()`.'

'I sent a message to a `Frog` object using the message name `left()`.'

Thus the message name for the message `setColour(OUColour.PURPLE)` is `setColour()`, while the message name for the message `left()` is just `left()`.

In summary, a message is something you send, but a message name is the name of a message or family of messages. So, when we give the instance protocol of `HoverFrog` as `left()`, `right()`, `home()`, `up()`, `down()`, `jump()`, `green()`, `brown()`, `croak()`, `setColour()`, `upBy()` and `downBy()`, we are giving the protocol in terms of message names.

SAQ 7

(a) What is the message name in the message `setColour(OUColour.YELLOW)`?

(b) What is the message name in the message `jump()`?

ANSWER.....

(a) `setColour()`

(b) `jump()`

SAQ 8

What is the message name in each of the following?

(a) `frog1.setColour(OUColour.BROWN)`

(b) `hoverFrog2.green()`

ANSWER.....

(a) `setColour()`

(b) `green()`

ACTIVITY 6

If the microworld Frog & HoverFrog is not open, open it now.

In the Java programming language, there is a convention for supplying arguments. Where a message requires an argument, you write the message name followed by the argument in the brackets that always follow.

```
setColour(OUColour.RED)
```

In the Code Pane, to send a message to change the colour of an object, it is necessary first to type the name of the variable referencing the receiver, then a dot, and then the message name with the argument in brackets, followed by a semicolon. For example,

```
frog5.setColour(OUColour.RED);
```

Then press the Execute button.

Try sending some colour-changing messages to both `frog5` and `hoverFrog3` using the Code Pane. If you make a mistake and obtain an error report in the Display Pane, clear the Display Pane by pressing the Clear button and then check the spelling, spacing and capital letters in your typing. (Colour objects have names in upper-case letters and must be preceded by `OUColour.`)

Type `frog5.setColour();` – with the argument omitted – and press the Execute button. You will see that the error report appears in the Display Pane.

DISCUSSION OF ACTIVITY 6

The objects referenced by `frog5` and `hoverFrog3` should have changed colour in accordance with the messages you sent.

ACTIVITY 7

To make `hoverFrog3` hover using the message `upBy()`, an argument has to be supplied; the argument here is a number chosen from the integers `1` to `6`.

So the following is typed in the Code Pane to form a message asking `hoverFrog3` to move up two 'steps', i.e. by two units:

```
hoverFrog3.upBy(2);
```

Send this message. Check that your typing matches the above exactly, taking particular care with the spacing, semicolon and capitalisation, before pressing the Execute button.

Try sending several versions of the same message with other arguments, and also try sending some messages using `downBy()`.

What is the protocol of `HoverFrog` objects, as revealed in this and previous activities?

DISCUSSION OF ACTIVITY 7

The protocol of `HoverFrog` objects as revealed in this activity is `left()`, `right()`, `home()`, `up()`, `down()`, `jump()`, `brown()`, `green()`, `croak()`, `setColour()`, `upBy()` and `downBy()`.

SAQ 9

What is the difference in general between a message and a message name? What is the difference between a message and a message name when the message has no arguments?

ANSWER.....

A message name is the textual form of a message except that any arguments it takes are not shown. If a message takes no arguments anyway, the message and its name will be indistinguishable.

SAQ 10

What is the message name in each of the following?

(a) `oldBicycle.remove(bell)`

(b) `oldBicycle.remove(bell, newBicycle)`

ANSWER.....

(a) `remove()`

(b) `remove()`

4 Message answers, enquiry messages and collaborating objects

We start this section with two activities that look at answers generated in response to some messages. We then look at accessor messages, distinguishing between a message that modifies the state of an object and a message that interrogates the object's state. We conclude this section by discussing collaborating objects and sequence diagrams.

4.1 Message answers

In the following activities you will explore messages that request information from the receiver. These activities show that message answers can be used by objects that need to collaborate (typically to share information).

ACTIVITY 8

From the Amphibian Worlds application open the Three Frogs microworld. The Graphics Pane in this microworld shows representations of three `Frog` objects – `frog6`, `frog7`, `frog8` – and has two new buttons, labelled getColour() and sameColourAs(). You will see that some messages result in a message answer, but some do not.

Select a `Frog` object and send it the message `getColour()` by pressing the button labelled getColour(). Look at the text in the Display Pane and make a note of what you read.

Now send the same object the message `setColour(OUColour.RED)` using the button labelled setColour() and look at the Display Pane: has any text been added?

Use the Code Pane to type and execute some similar examples. Look in the Display Pane after each line has been executed.

What are the differences in the behaviour of the `Frog` objects in response to messages with names `getColour()` and `setColour()`?

DISCUSSION OF ACTIVITY 8

When the getColour() button is pressed, some text describing the colour of the selected `Frog` object, for example `OUColour.GREEN`, appears in the Display Pane. The message `getColour()` seems to be asking the receiver frog 'What is the value of your `colour` attribute?' and the message is answered. Similarly, if you type `frog6.getColour()` in the Code Pane, the Display Pane shows the text `OUColour.GREEN`.

What you see in the Display Pane is a textual – as opposed to a graphical – representation of the message answer returned in reply to the message. Hence, for example, you will see the text `OUColour.GREEN` rather than a patch of greenish hue.

When a `Frog` object is sent the message `setColour(OUColour.RED)` it responds by changing its `colour` attribute to `OUColour.RED`, and nothing appears in the Display Pane. A `setColour()` message does not return an answer.

In contrast, the message `getColour()` does not change the state of the receiver; it returns as the message answer the value of the attribute `colour`.

ACTIVITY 9

We shall now ask you to examine the sameColourAs() menu button and the message named `sameColourAs()` in the microworld Three Frogs.

The sameColourAs() button is used to send a `sameColourAs()` message to a `Frog` object requesting it to change its colour to the same colour as another `Frog`, `Toad` or `HoverFrog` object. The argument specifies the object from which you want the receiver to take its new colour.

1 Select the variable `frog6` and press the button labelled green().
Select the variable `frog7` and press the button labelled brown().

2 Select the variable `frog6`, press the button labelled sameColourAs() and select the variable `frog7` from the menu. This sends the message `sameColourAs(frog7)` to the selected receiver `frog6`. You can use the button labelled setColour() to send messages to the three `Frog` objects so that each is a different colour again. Practise sending similar messages.

3 Now use the Code Pane to send messages to the three `Frog` objects so that each is a different colour. For example, the following lines, on execution, will change the colour of `frog6` to purple and `frog7` to brown.

```
frog6.setColour(OUColour.PURPLE);
frog7.brown();
```

Now type a line in the Code Pane similar to the one below and execute it.

```
frog6.sameColourAs(frog7);
```

4 Send other messages using the `sameColourAs()` message name.

DISCUSSION OF ACTIVITY 9

In part 2 the receiver changed colour to that of the `Frog` object specified as the argument to the message – `frog6` was initially green, and changed from green to brown in response to the `sameColourAs(frog7)` message. In part 3 you used the Code Pane to send `setColour()` and `sameColourAs()` messages.

For one object (`frog7`, for example) to be used as an argument for a message to another object (`frog6`, for example) involves the *collaboration* of the argument object and the receiver.

SAQ 11

What is the protocol of `Frog` objects as revealed in the microworld Three Frogs?

ANSWER.....

The protocol of `Frog` objects as revealed in this activity is `left()`, `right()`, `home()`, `jump()`, `green()`, `brown()`, `croak()`, `setColour()`, `getColour()` and `sameColourAs()`.

4.2 Accessor pairs of messages

The message `getColour()` has a different purpose from the previous messages you have seen. In sending the message `getColour()`, you are not interested in changing the state of the receiver object. Instead you are interested in obtaining information about its state.

When the message `getColour()` is sent to a `Frog` object, it replies with information about its colour. The requested information is returned as the **message answer**. In sending this message, you ask the `Frog` object for the value of its `colour` attribute. It is usual to say that the message `getColour()` 'returns the colour of the receiver'. In sending the message `setColour(OUColour.RED)` to a `Frog` object, you request the receiver to set its attribute `colour` to `OUColour.RED`. Note that you are not requesting an answer, and no answer is given.

A message answer is sometimes called a **message reply**.

It is common in this way for the protocol for an object to include pairs of messages: one that gets the value of an attribute of the receiver object, and one that allows that attribute to be set. A **getter message** and the corresponding **setter message** (as the get and set messages are termed) together form what is called a pair of **accessor messages**. In our example we have `getColour()` and `setColour()`, where `getColour()` is the getter message name and `setColour()` the setter message name.

SAQ 12

What, if anything, is returned as the message answer when you send the following messages to a `Frog` object that is brown and is on the leftmost 'stone'?

(a) `getColour()`

(b) `setColour(OUColour.RED)`

ANSWER.....

(a) `OUColour.BROWN`

(b) No answer is returned.

4.3 Collaborating objects

The message `right()` sent to a `Frog` object requests a straightforward response from the receiver object: no other objects are involved. Often, however, before an object can act on a message it has to collect some information from another object or objects; it does this by sending them messages. The required information comes back as answers to the messages sent, enabling the object to deal fully with the initial message. In such cases, several objects are 'collaborating' by sending messages to each other. (This is not the only form of collaboration; some objects might collaborate by taking responsibility for particular bits of required behaviour.)

Message answers can also provide information that is to be used with a later message. Sometimes the information is the value of a particular attribute of an object. For example, if a `Frog` object needs to know the colour of another `Frog` object, it would have to send the other `Frog` object a message and the information would be returned as the message answer. There is no reason why message answers should not be used as message arguments in subsequent messages.

Two `Frog` objects **collaborate** when `frog1.sameColourAs(frog2)` is executed, as described below.

The message named `sameColourAs()` when sent to a `Frog` object requests it to change its colour to be the same as another `Frog` object. The argument for `sameColourAs()` specifies the `Frog` object from which the receiver takes its new colour. There is nothing new about asking a `Frog` object to change its colour, but in all previous cases the message indicated explicitly what the new colour should be – as in the message `setColour(OUColour.RED)`. However, in using the message named `sameColourAs()`, an argument specifies another `Frog` object that has the required information (as the value of its attribute `colour`).

A technique that is often used to follow a sequence of messages between collaborating objects is to put yourself in the place of the receiver of the message. To follow the sequence of messages when the message `sameColourAs(frog2)` is sent to `frog1`, this technique requires you to put yourself in the place of `frog1`. You are asked to use this important technique in Exercise 1.

Exercise 1

Imagine that you are the object `frog1`. If you as `frog1` are sent the message `sameColourAs(frog2)`, describe informally the messages you would have to send and the answers you would use in order to satisfy this request.

Solution

As `frog1`, you have been asked to change your colour to the same colour as `frog2`. Now, as a human, you can, of course, see the colour of `frog2` by looking at the graphical interface provided. But, as `frog1`, you have no way of 'seeing' such information. The only way that you, as `frog1`, can find out that colour is to send the message `getColour()` to `frog2`. The colour will be provided to you, as `frog1`, as a message answer. Now you are nearly, but not quite, finished.

You, as `frog1`, still need to change your own colour to the colour given by the message answer. But you cannot just effect this by magic. As you know, the only way you can get an object to do anything is to send it a message. This is often the best way to achieve something even when the object in question is you. Hence the sensible way for you, as `frog1`, to change your own colour in line with the message answer is to send yourself the message `setColour()` using as argument the colour you got from `frog2` as a message answer. (The message named `setColour()` is used with an argument to request the receiver object to change colour.)

Sending a message to yourself may seem a strange way for an object to behave, but it is in line with the rules by which objects communicate. Nothing forbids an object sending a message to itself. Often it is the simplest (and sometimes the only) way for an object to get something done.

The interaction between the objects when the statement `frog1.sameColourAs(frog2);` is typed into the Code Pane by a user and then executed, can be seen in diagrammatic form in Figure 2. This kind of diagram is sometimes known as a **sequence diagram**. The user originates the chain of messages, and will see an effect in the form of a change to the visible `frog1` icon. It is the collaborations between the software objects, rather than between the user (you) and user interface, that are important here.

In Figure 2, vertical lines, called lifelines, represent the objects named above them, solid arrows represent messages from sender to receiver, and dashed arrows represent message answers. Time runs vertically downwards. It is assumed that the attribute `colour` of `frog2` has the value `OUColour.GREEN`.

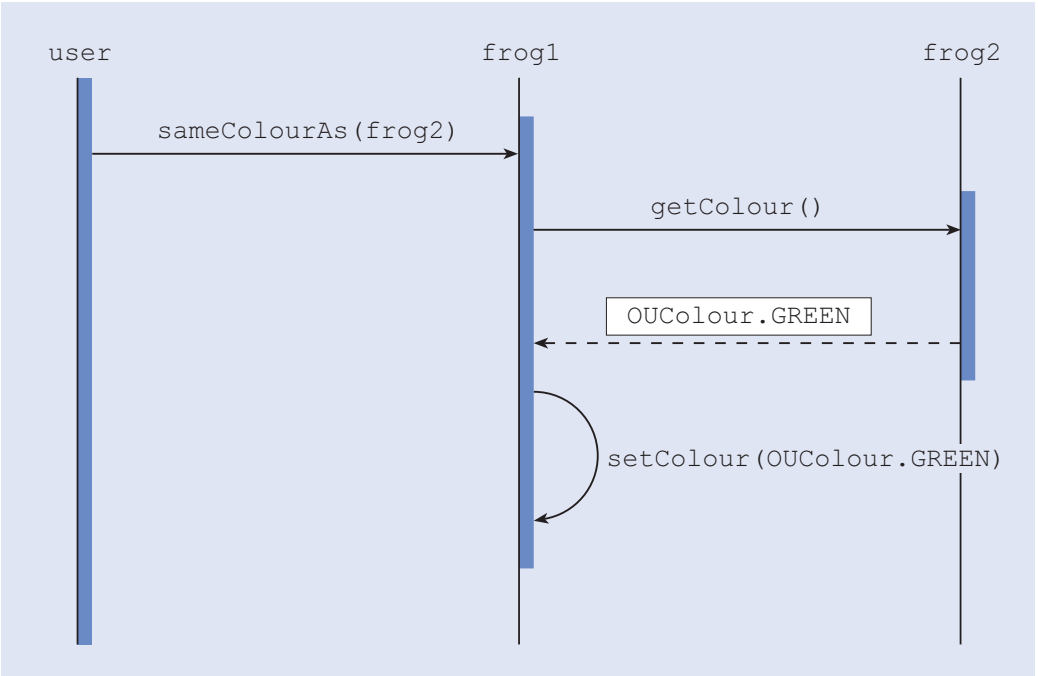

Figure 2 A sequence diagram

Exercise 2

Answer Exercise 1 again using Figure 2, this time stating more formally the messages that `frog1` would have to send on receipt of the message `sameColourAs(frog2)` and what message answers would be involved. Assume that the attribute `colour` of `frog2` has the value `OUColour.GREEN`.

Solution

On receipt of the message `sameColourAs(frog2)`, `frog1` sends the message `getColour()` to `frog2`. The answer to this message is `OUColour.GREEN`. Now `frog1` sends the message `setColour(OUColour.GREEN)` to itself.

In order to get the desired effect, the message answer from the first message is used as the argument to the second message.

In Figure 2, the messages sent from one object to another are shown in order down the page. The messages are ordered from top to bottom. Reading from top to bottom, the sequence is as follows.

(a) The user (you are no longer `frog1`!) sends the message `sameColourAs(frog2)` to `frog1`.

(b) `frog1` sends the message `getColour()` to `frog2`.

(c) `frog2` returns as the message answer the value of its attribute `colour` (`OUColour.GREEN`) to `frog1`.

(d) Finally, `frog1` sends itself the message `setColour(OUColour.GREEN)`, where the argument `OUColour.GREEN` is the message answer from the previous message.

In the above example, the user sends a single message `sameColourAs(frog2)` to `frog1`. The message causes `frog1` to respond by sending two messages in turn. The first of these messages is to `frog2` and has the purpose of eliciting information in the form of a message answer, and the second is to itself. The user does not directly specify the new colour for `frog1`; rather, `frog1` had to ask `frog2` for the information. Then `frog1` changes its own colour (by sending a message to itself). In this example, the user

caused a `Frog` object to send a message to another `Frog` object. The two `Frog` objects worked together in order to do something you (the user) requested. This is an example of **collaborating objects**.

4.4 The `Frog` class

Before proceeding to another scenario, we shall summarise the key features of a class in terms of the `Frog` class.

A class groups together objects that the programmer considers to be similar. Instances of the same class respond to the same set of messages (the instance protocol), have the same attributes and respond in the same way to each message.

All instances of the class `Frog` are created with the same attributes `colour` and `position`, but each `Frog` object is an 'individual' in that the values of its attributes belong to itself. So `frog1` may have `position` as `1` and `colour` as `OUColour.BROWN`, whereas the attributes of `frog2` may have the values `2` and `OUColour.BLUE`. The protocol for the `Frog` objects is defined in the class `Frog`, so all instances understand the messages `left()`, `right()`, `home()`, `jump()`, `brown()`, `green()`, `croak()`, `getColour()`, `setColour()` and `sameColourAs()`, and respond in the same way to each message. The initialisation of each instance is also defined in the class `Frog` so that each `Frog` object is initialised with the value of `position` as `1` and the value of `colour` as `OUColour.GREEN`.

5 A bank account class

In this section we reinforce the notions of object, message and message answer by using them in an everyday situation.

This is done with reference to a particular application, a simple banking system, which is designed to handle very basic bank accounts. Such an account will normally require a certain amount of information to be associated with it. In this example the accounts are very simple and the only information they will store will be the name of the account holder, the account number and the current balance. These accounts will be modelled using `Account` objects, one object for each customer. An `Account` object will have the following attributes: `holder`, `number` and `balance`. Figure 3 shows how such an `Account` object can be represented by an **object-state diagram**.

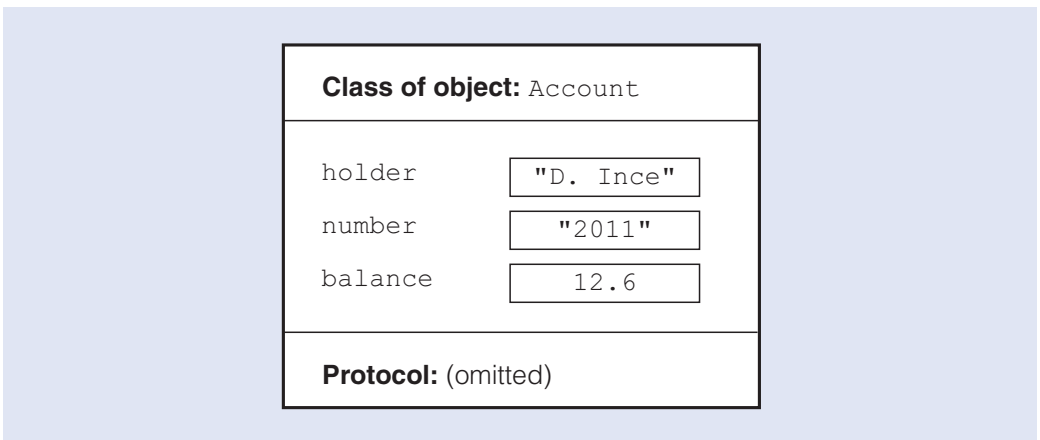

Figure 3 An object-state diagram of an **`Account`** object

The object shown in Figure 3 has attributes `holder`, `number` and `balance`, and each currently has a value – `"D. Ince"`, `"2011"` and `12.6`, respectively. Note that account numbers are treated as textual rather than as numeric quantities.

The diagram reflects the fact that this object belongs to a class called `Account`. There is space in the diagram for the protocol of the object, i.e. a listing of the messages to which the class `Account` objects can respond. The space for the protocol is blank for now; later in this unit you will discover what messages an `Account` object can understand.

A banking system can be modelled (in part) as a set of `Account` objects, each of which corresponds to someone's account with the bank. Some means of referring to each `Account` object is required. A solution would be to have suitably named variables, each referencing an account object, just as was done earlier with `Frog` objects. The `Account` object represented by Figure 3 can be referenced by a **variable** called e.g. `myAccount`. That this particular variable refers to a particular object is symbolised in Figure 4 by means of an arrow pointing from the variable name to the object.

Note the use of a capital A in the middle of the variable name `myAccount`. This convention is common in programming – when a variable name or message name is composed of two or more English words, or abbreviations, the first word starts with a lower-case letter, and a single upper-case letter is used to mark the start of each subsequent word.

Always take care with capitals when typing, since Java distinguishes between upper-case and lower-case letters.

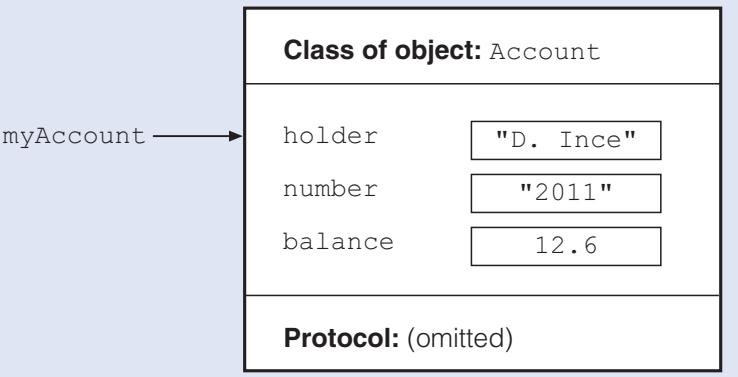

Figure 4 A named **Account** object

You will carry out the activities in this section using a simple programming tool for `Account` objects called Accounts World, which allows you to create new `Account` objects, and to send messages to them.

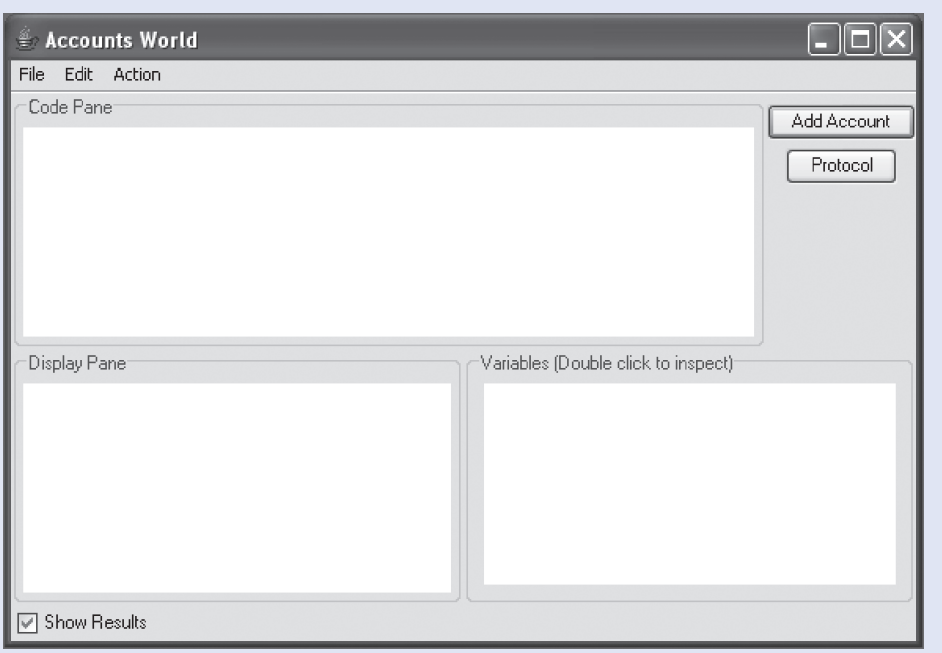

Figure 5 The Accounts World programming tool

The Accounts World has three panes. The top pane in the Accounts World is called the Code Pane. This is where you type statements that you want to be executed. Statements are not executed until they have been highlighted (selected) and their execution requested by choosing Execute Selected from the Action menu. To the right of the code pane are two buttons. The button labelled Add Account is used to create `Account` objects (you will be prompted for a variable name when you click this button). The other button, labelled Protocol, is used to open another window, which you can use to get information on the protocol of `Account` objects.

Below the Code Pane there are two further panes. On the left there is the Display Pane. This has two functions. Firstly, it is where the Accounts World's Java interpreter will write any error messages if you make a mistake in any statement(s) you are testing in the Code Pane. For example, if you attempt to send a message to an `Account` object, which is not in its protocol (such as misspelling the `getBalance()` message, we encounter

later, as `geetBalance()`) the Java interpreter that is used to parse and execute code in the Code Pane will write `Semantic error: Message geetBalance() not understood by class 'Account'` in the Display Pane. The second function of the Display Pane is to display the value of the last expression evaluated in a statement (or series of statements). This will occur only if you check the Show Results check box. To the right of the Display Pane is the Variables Pane. This pane will display any variables declared in the Code Pane.

Remember that when you close the Accounts World any objects you created will be lost.

5.1 Creating and inspecting `Account` objects

Most of your study time in this subsection will be taken up with practical activities.

These activities are intended to give you a flavour of how objects and messages can be used to represent an everyday situation: the use of bank accounts. In the Amphibian microworlds you worked with objects that had already been created for you; here you will create new objects and send messages to them. In doing so, you will be introduced to terminology about code in Java that will be very important later in the course.

Typing and dealing with errors

You may find that unexpected problems arise when you try to execute code that you have typed. This is often the result of incorrect typing. To help avoid these errors you may like to read the following guide and return to it if you receive error reports.

Error messages may result from Java not understanding what you have typed in when you try to execute the code, or from a typing error causing you apparently to ask Java to send a strange message to a strange object. The error messages displayed in the Display Pane may not always make much sense, but do not worry – in the majority of cases, the solution can be found among the following points, which you should glance over before you start. So take note of the following tips before typing into the Code Pane or when you get an error message in the Display Pane.

1 Check spelling, capitalisation (or its absence), spaces and punctuation. All of these can alter the meaning of a variable name or message.

2 Make sure you are selecting the desired text, the whole of the desired text and nothing but the desired text when you execute it.

3 By convention all message names begin with lower-case letters, but message names may include upper-case letters (e.g. `getNumber()`).

4 Use the Variables Pane to check you are using the name of the variable referencing the `Account` object exactly as you declared it when you created it. You may have typed it in differently from the way it appears in this text. In particular, check spelling and capitalisation.

5 When you are creating a new `Account` object, you are asked to supply a name for a variable to reference it. Make sure that the name contains no spaces.

6 Use the protocol for the `Account` class to check you are spelling message names correctly.

7 Later, you will meet messages with multiple arguments. There should be commas separating multiple arguments. Java does not require a space after such a comma and before the following argument, but please include one for clarity.

ACTIVITY 10

Here you are going to create a new `Account` object, referenced by a variable called `myAccount`. In the Accounts World click on the Add Account button. In the window that opens you will need to give the name for a variable to reference the new object. Use the name `myAccount` (be careful about capitalisation) and then press the OK button.

This creates a new `Account` object referenced by the variable named `myAccount`. You have now created an `Account` object, but you do not know yet what state it is currently in. A quick way to find out is to use an inspector. To examine the state of the object referenced by `myAccount`, double-click its variable name in the Variables Pane. An Inspector window will be opened on the newly created `Account` object. Use the Inspector window to answer the following two questions.

1 What is the initial state of the object referenced by `myAccount`?
2 Has the name of the variable `myAccount` had any effect on the value of the `holder` attribute of the object?

When you have finished, close the Inspector window.

Keep the Accounts World open, as the `Account` objects will be used in the next activity.

DISCUSSION OF ACTIVITY 10

When you created your new `Account` object you should have seen the name of the variable you created appear in the Variables Pane.

1 The `holder` and `number` attributes of the new `Account` object are given as `""` (the empty string). The `balance` is given as `0.0`. When an `Account` object is created, its attributes are always initialised in this way.
2 As the inspector shows, the name of the variable referencing an `Account` object and its `holder` have no connection – they are entirely different things.

ACTIVITY 11

You have created an `Account` object, but to change its state, or get it to do anything else, you will need to send it messages. In our banking system, an account belongs to someone (represented by the `holder` attribute of an `Account` object) and the account has an account number and current balance. To make a start, you are going to make the holder of the account be someone named `"Grendel Barty"` with an account number of `"1234"`, and you are going to credit the account with a sum of `100`.

Return to the Accounts World, and send the message `setHolder("Grendel Barty")` to `myAccount` (taking care not to forget the quotes). That is, you will need to type in the Code Pane the following.

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

To execute the message, select it and right-click, and choose the menu item Execute Selected.

Recall, from *Unit 1*, that this is known as a **message-send** and consists of a receiver (the object being sent a message), followed by a full stop and then a message. In the above expression, the receiver is the object referenced by `myAccount` and the message is `setHolder("Grendel Barty")`.

Then type in and execute the message-send

```
myAccount.setNumber("1234");
```

and then

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

Inspect `myAccount` and note what you see.

DISCUSSION OF ACTIVITY 11

When you double-clicked `myAccount`, an inspector was created that showed the current state of the `Account` object referenced by the variable `myAccount`. As a result of your messages the inspector should report that the `holder` is `"Grendel Barty"`, the `number` is `"1234"` and the `balance` is `100.0`.

ACTIVITY 12

In the Accounts World, create two more `Account` objects. In particular, create a new `Account` object referenced by a variable called `hisAccount` (for the holder `"Everest Grundy"`, who has an account number of `"2468"`) and credit it with `200`. Then create a new `Account` object referenced by a variable called `herAccount` (for the holder `"Lucy Nijholt"`, who has an account number of `"1111"`) and credit it with `300`. After you have sent these messages, inspect each of the two objects to check that its subsequent state is what you expect.

Keep the Accounts World open, as the `Account` objects will be used in the next activity.

DISCUSSION OF ACTIVITY 12

Creating the new accounts in the Accounts World is just a matter of clicking on the Add Account button, and providing a variable named `hisAccount` (or `herAccount`) for the `Account` object you want to create. When you have clicked on OK, the `Account` object is created. You should then see the variable name `hisAccount` (or `herAccount`) appear in the Variables Pane.

Once you have created the accounts, the message-sends you should have executed are given below.

```
hisAccount.setHolder("Everest Grundy");
hisAccount.setNumber("2468");
hisAccount.credit(200);
herAccount.setHolder("Lucy Nijholt");
herAccount.setNumber("1111");
herAccount.credit(300);
```

Inspecting your newly created objects should confirm that the object referenced by the variable `hisAccount` is held by `"Everest Grundy"`, with a `number` of `"2468"` and a `balance` of `200.0`; and that the object referenced by the variable `herAccount` is held by `"Lucy Nijholt"`, with a `number` of `"1111"` and a `balance` of `300.0`.

Exercise 3

The only way you have seen to make an object do anything (even to divulge its state) is to send it a message. But in the practical activity above you managed to use an inspector to find out the state of an object referenced by variable `myAccount` immediately after it was created, without apparently sending any message to `myAccount`. Using what you know about objects, can you think of a straightforward explanation of what happened when you used an inspector to find out the state of `myAccount`?

Solution

Double-clicking a variable name creates an inspector, which is itself an object. It was the inspector object that sent a message to `myAccount` to find out its state, and then displayed the result. Hence it still holds true that the only way to find out the state of an object is to send it messages.

Exercise 4

You have seen how **message-sends** consist of a receiver and a message. In the practical work you have also constructed message-sends with arguments, as in `myAccount.setHolder("Grendel Barty")`. In this case the message is composed of a message name, `setHolder()`, and an argument, `"Grendel Barty"`.

For each of the message-sends `myAccount.setNumber("1234")` and `myAccount.credit(100)`, identify the receiver, message name, argument and message.

Solution

For the message-send `myAccount.setNumber("1234")`

- `myAccount` is the receiver,
- `setNumber()` is the message name,
- `"1234"` is the argument,
- `setNumber("1234")` is the message.

For the message-send `myAccount.credit(100)`

- `myAccount` is the receiver,
- `credit()` is the message name,
- `100` is the argument,
- `credit(100)` is the message.

SAQ 13

What is the connection between the name of the variable used to reference an `Account` object and its `holder`?

ANSWER

There is no connection. If you had an `Account` object that was referenced by the variable `thisAcc` and whose `holder` was `"J. Bloggs"`, you could still reference this object using `thisAcc` while sending a message to change the `holder` to `"Mary Brown"`.

SAQ 14

Which of the following is true?

(a) An object may be referenced by at most one variable.

(b) An object must be referenced by exactly one variable.

(c) An object could be referenced by several variables.

ANSWER

(a) False

(b) False

(c) True

5.2 Exploring the protocol of `Account` objects

As in the previous subsection, most of your work here will consist of activities.

Now that you have three `Account` objects this is a good time to explore some more of the `Account` protocol.

ACTIVITY 13

Go to the Accounts World and explore the protocol for the `Account` class. To do this, click on the Protocol button – this will open a window displaying information about the `credit()` message. On the left-hand side of the window is a set of buttons labelled with the names of the messages in the protocol of `Account` objects; clicking one of these buttons will then display the information (documentation) for that message – what it does, what arguments it takes and what value (if any) it returns.

You can now turn your attention to changing the state of an `Account` object, namely that referenced by the variable `herAccount`. (If you had previously closed the Accounts World then you will need to create an `Account` object referenced by the variable `herAccount` and set the balance of the account to `300`.) Send messages to perform the actions listed below, in the order given, by entering the relevant code in the Code Pane. After sending each message, look in the Display Pane at the textual representation of any message answer that is produced.

1 Use the message `getHolder()` to check that the holder of the account referenced by the variable `herAccount` is `"Lucy Nijholt"`.
2 Debit 100 from Lucy's account.
3 Use the message `getBalance()` to check the resulting balance.
4 Set the `number` of Lucy's account to `"2000"`.
5 Send a message to check the account number of Lucy's account.
6 Debit 3000 from Lucy's account.
7 Use the message `getBalance()` to check the resulting balance of Lucy's account.

Draw an object-state diagram to depict the object referenced by `herAccount` and its current state. Add the names of the messages you have used so far to the Protocol section of the diagram.

DISCUSSION OF ACTIVITY 13

The corresponding Java code and message answers are as follows.

1 `herAccount.getHolder();`
(the textual representation of the message answer is `"Lucy Nijholt"`).
2 `herAccount.debit(100);`
(the textual representation of the message answer is `true`).
3 `herAccount.getBalance();`
(the textual representation of the message answer is `200.0`).
4 `herAccount.setNumber("2000");`
(there is no message answer).
5 `herAccount.getNumber();`
(the textual representation of the message answer is `"2000"`).
6 `herAccount.debit(3000);`
(the textual representation of the message answer is `false`).
7 `herAccount.getBalance();`
(the textual representation of the message answer is `200.0`).

A `debit()` message returns the answer `true` if the transaction has been actioned (because the `balance` represents sufficient funds) and `false` if it has not been actioned (because there are insufficient funds).

Figure 6 shows the object-state diagram. Only the part of the protocol you have discovered is shown.

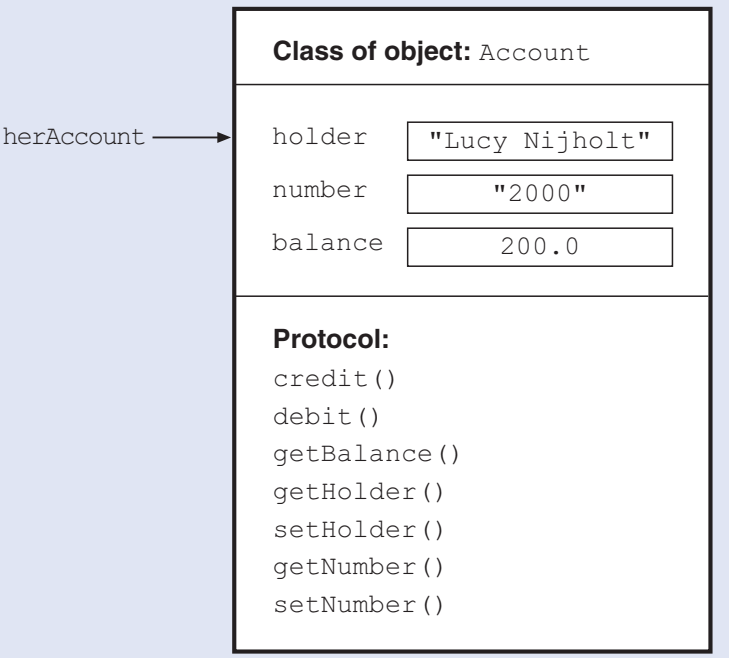

Figure 6 Object-state diagram for **Account** objects

The messages you have used so far are: `credit()`, `debit()`, `getBalance()`, `getHolder()`, `setHolder()`, `getNumber()`, `setNumber()`. Note that, as in the Amphibian Worlds, the message used to find the value of an attribute has a name consisting of the name of the attribute preceded by the word `get`. This is a common convention in Java programming. Similarly, a message used to set the value of an attribute has a name based on the name of the attribute, but preceded by the word `set`.

SAQ 15

Will Java treat `MyAccount` and `myAccount` as the same name?

ANSWER.....

No. The exact spelling and capitalisation matter in names of variables (and in names of messages and classes).

5.3 Collaborating `Account` objects

In this subsection you will be using the Accounts World to explore how objects in the `Account` class can collaborate.

There are some messages in the protocol of the `Account` class that you have not yet tried to send. One of these is the message used to transfer money directly from one

Account object to another. The name for this message is transfer(). In Activity 14 you will use this message to transfer 300 from the account referenced by myAccount to that referenced by herAccount.

The transfer() message requires two arguments. In Java, when a message requires more than one argument, the arguments are listed between the parentheses of the message and are separated by commas. In the case of the transfer() message, the first argument is the Account object to which the money is to be transferred and the second argument is the amount to be transferred. For example, to transfer 300 from the receiving object to account herAccount you would use the message transfer(herAccount, 300).

ACTIVITY 14

In the Accounts World, make sure you have two Account objects referenced by the variables myAccount and herAccount; if you do not have them, create them. Next, send a message to credit myAccount with 500.

Before you start, make sure you know the current balances in myAccount and herAccount. Now, try using transfer() to transfer 300 from myAccount to herAccount. (Click on the Protocol button to see how to use transfer().) Then inspect the two Account objects, referenced by myAccount and herAccount, to see if they have changed state in the way you would expect from the sense of the message.

DISCUSSION OF ACTIVITY 14

To transfer 300 between the accounts you use the message-send

```
myAccount.transfer(herAccount, 300)
```

The money has been transferred in the way you would expect. The answer from this message is true – the transfer was successful.

Exercise 5

The purpose of this exercise is for you to think about how the receiver of a message with name transfer() will carry out its responsibility to respond to this message, in the case where there are sufficient funds in the receiver account for the transfer to be actioned.

Imagine now that you are the object referenced by myAccount, in a state with a balance of 800. Imagine that a user of the system sends you the message transfer(herAccount, 400), which requests you to transfer 400 to the account referenced by herAccount.

Imagine that you must carry out the **responsibility** to respond to this message, using a sequence of messages. To achieve your goal, you can send any message you like (except transfer()) to any Account object. To solve the problem, try to break it up into stages. What would a person do to carry out the same responsibility step-by-step? Try to find a message to help you carry out each stage. Draw a **sequence diagram**, like the one in Subsection 4.3, to help you solve the problem and to illustrate your solution. Your diagram should include the user and any relevant objects. Fill in a table like the following one to show in turn each receiver, each message including any arguments, and message answers, if relevant, of any messages you need to send to satisfy your responsibility.

Strictly speaking, transfer() is not a message, it is only a message name. But most programmers say things like this for simplicity. This habit is fine provided that you make sure you are clear about the difference.

Receiver	Message	Message answer

Hint: You may need to send a message to yourself as a necessary step in discharging your responsibility.

Solution

Just two messages are required. Note that the first message, `debit(400)`, must be sent to yourself in your role as the object `myAccount`.

```
myAccount.debit(400)
herAccount.credit(400)
```

These two messages can be analysed in tabular form as follows.

Receiver	Message	Message answer
`myAccount`	`debit(400)`	`true`
`herAccount`	`credit(400)`	none

In Figure 7 the three vertical lines represent the user and the two objects `myAccount` and `herAccount`. Objects are drawn as vertical lines so that messages sent from one object to another can be shown clearly in order down the page. Bold arrows represent messages. Message answers are not shown as they do not play a major role in this case. The messages are ordered left-to-right and top-to-bottom. Reading from left-to-right, the user first sends the message `transfer(herAccount, 400)` to `myAccount`. To satisfy this responsibility, `myAccount` first sends the message `debit(400)` to itself. Next, `myAccount` sends the message `credit(400)` to `herAccount`. This is all that `myAccount` has to do to discharge the responsibility it took on by accepting the message `transfer(herAccount, 400)`.

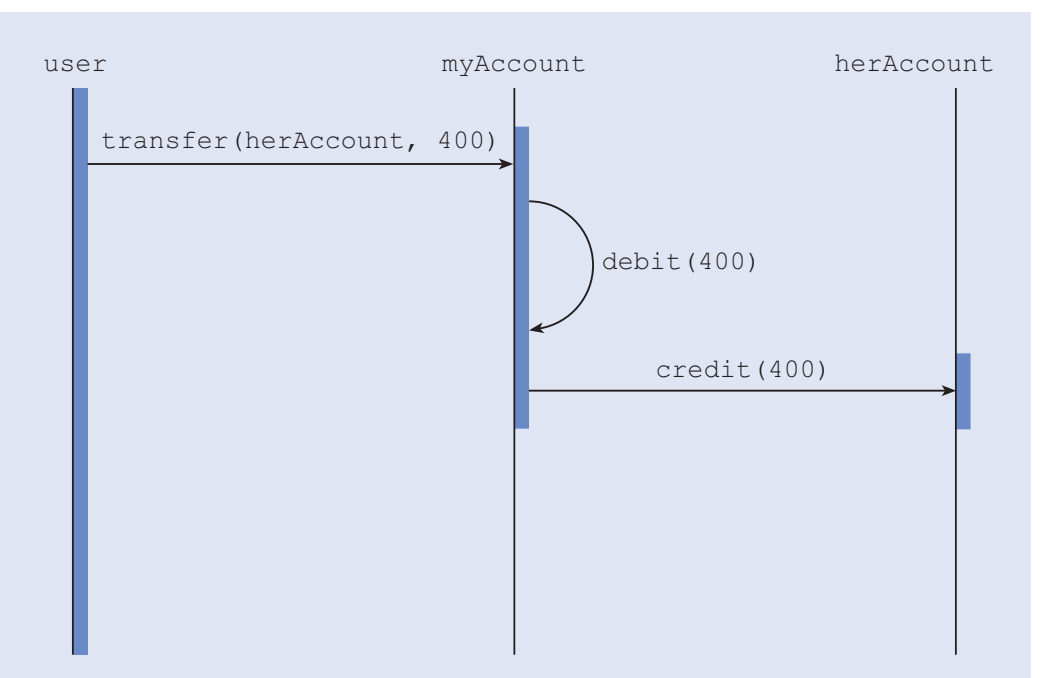

Figure 7 A sequence diagram for the message-send **`myAccount.transfer(herAccount,400)`**

We should emphasise that we are *not* saying that the `transfer()` message is always equivalent to a pair of messages using `debit()` and `credit()`. It will not be equivalent in cases where the receiver of the `transfer()` message has insufficient funds for the transfer. In more detail, what happens when `myAccount.transfer(herAccount, 400)` is executed is that `myAccount` sends itself a `debit(400)` message and only if the message answer is `true` does it then send `herAccount` the message `credit(400)`.

Contrast this with the message `frog1.sameColourAs(frog2)`, where the receiver (`frog1`) always sends itself a message using `setColour()` after getting the `colour` of `frog2` by sending `frog2` a `getColour()` message. You will be able to see how this is done at the level of code once you move on to later units, where methods (*Unit 4*) and conditions (*Unit 5*) are introduced.

Thus, if you were the object referenced by `myAccount`, with a sufficiently large balance, and you received the message `transfer(herAccount, 400)`, you would send messages to appropriate receivers as follows:

1 `debit(400)` to yourself

2 `credit(400)` to `herAccount`

There is a simple way of checking that this scheme works; in your real-life role as user (as opposed to your previous make-believe role as the object referenced by `myAccount`) you could now try sending these two messages. If you do it this way yourself, the final state of affairs should be exactly the same as if you just sent the message `transfer(herAccount, 400)` to the object referenced by `myAccount`, and thus requested it to take the responsibility.

ACTIVITY 15

In the Accounts World, make sure you have two `Account` objects referenced by the variables `myAccount` and `herAccount`; if you do not have them, create them. Ensure that `myAccount` has a `balance` of at least `400`. Then execute the following two message-sends one at a time in the Code Pane, and then inspect the objects to look at the resulting state of affairs.

```
myAccount.debit(400);
herAccount.credit(400);
```

DISCUSSION OF ACTIVITY 15

The effect on the final state of the objects is exactly the same as sending a single message `transfer(herAccount, 400)` to the object referenced by `myAccount`, requesting it to take responsibility for the whole transfer. We studied this example in Exercise 5.

Whilst Exercise 5 may seem a little like a game, it is in fact an accurate description of how an `Account` object discharges its responsibility of responding to a message `transfer()` when there are sufficient finds for the transfer.

The receipt of the message `transfer()` results in the `Account` object, which is the receiver, sending two messages (without any human intervention).

1 The receiver sends a `debit()` message to itself. The argument to `debit()` is the amount specified as the second argument of the `transfer()` message.

2 The receiver sends a `credit()` message to the `Account` object that is specified in the first argument of the `transfer()` message. The argument to `credit()` is the same amount that was specified in the second argument in the original `transfer()` message and used in the `debit()` of step 1.

We say that this is an example of an object discharging its responsibility by *collaborating* with other objects by sending them messages. We also say that the objects that the receiver collaborates with are its helpers in this context.

Exercise 6

You have now seen two sequence diagrams, one in Subsection 4.3 and one in the solution to Exercise 5. Looking at these diagrams, what interesting similarities and differences can you see between the way the following two lines of code are carried out?

```
frog1.sameColourAs(frog2);
myAccount.transfer(herAccount, 400);
```

Solution

This is an open-ended question. Many answers are possible. One or two possible answers are noted below. Your answers may be different, but nevertheless correct.

Similarities

- Both involve collaboration between two objects.
- Both involve an object sending a message to itself.
- Both original responsibilities are discharged by the receiver by using exactly two messages.

Differences

- In the frog example, a message answer becomes a message argument.
- In the account example, the message arguments in both of the messages used to discharge the responsibility are taken directly from the message requesting the transfer.

5.4 Recap of terminology

At this point it is useful to review some of the terminology to do with messages that you have learnt.

The receiver is the object to which a message will be sent. The name of the message must be one of those contained in the protocol of the receiver's class, otherwise the message-send will not compile. The arguments are the pieces of information required by the message; some messages require no further information, and some require one or more pieces of information.

So, for example, in the case of the message-send

```
myAccount.credit(7)
```

`myAccount` is the receiver, `credit()` is the message name and `7` is the argument.

Finally, some messages result in a message answer being returned.

Exercise 7

Fill in the following table for the message-send

```
herAccount.transfer(myAccount, 200)
```

Receiver	
Message	
Message name	
Argument(s)	
Message-send	

Solution

Receiver	`herAccount`
Message	`transfer(myAccount, 200)`
Message name	`transfer()`
Argument(s)	`myAccount, 200`
Message-send	`herAccount.transfer(myAccount, 200)`

Exercise 8

Decide whether each of the following statements is true or false. If false, say why.

- All messages consist of a *name* and an *argument*.
- An object is associated with a particular *class*.
- A message is an instance of a class.

Solution

- False. Some messages have no arguments; some have more than one argument.
- True.
- False. Messages are not objects, and so they are not instances of a class.

SAQ 16

A message may:

(a) change the state of an object;

(b) make an object do something without changing its state;

(c) get back some useful information from an object;

(d) cause an object to send a message to another object;

(e) cause an object to send a message to itself.

Give an example of a message from this unit to illustrate each effect that a message can achieve in the list above.

ANSWER

Effect		Example
(a)	change the state of an object	`left()`
(b)	make an object do something without changing its state	`jump()`
(c)	get back some useful information from an object	`getColour()`
(d)	cause an object to send a message to another object	`sameColourAs(frog2)` – the receiver sends `getColour()` to the argument
(e)	be used by an object to send a message to itself	`frog1` changes its own `colour` when carrying out actions in response to `frog1.sameColourAs(frog2)`. The name of the message sent is `setColour()`.

SAQ 17

Give your understanding of the following terms and then look at the descriptions in the Glossary. (No answer to this question is given.)

argument, attribute, class, message answer, object, polymorphism, protocol, state, subclass.

6 Summary

After studying this unit you should understand the following key ideas.

- The values of an object's attributes constitute its state.
- An object's behaviour is determined by how it responds to the messages it understands (its protocol).
- The behaviour of an object in response to a message may be dependent on its state. An example of such state-dependent behaviour is the behaviour exhibited by `HoverFrog` objects when sent the `upBy()` message.
- Objects are organised into classes. Objects belonging to the same class (instances of the class) have the same set of attributes and respond to the same set of messages, responding to each message in an identical (but often state-dependent) manner. Classes are can be thought of as factories or templates for creating objects.
- When an object is created it has an initial state (its attributes have initial values). All instances of the same class have the same initial state.
- A class may have subclasses. An instance of a subclass has all the attributes and protocol of the parent class (the superclass), but the subclass may add to them.
- A subclass may modify the response to a particular inherited message, so that an instance of the subclass will respond to that message in a different way to an instance of the superclass. Furthermore, it is also common for instances of unrelated classes to have the same message in their protocols, and instances of those classes may or may not respond differently to that message. Such messages are termed polymorphic messages.
- Messages may return a message answer, and/or change the state of a receiver, or do neither. For example, the message `getColour()` merely returns the colour of the receiver whereas `setColour()` changes the value of the `colour` attribute. In the protocol of `Frog` objects, `jump()` neither changes the object's state nor returns a message answer.
- Some messages require arguments – extra information that is required for the message to make sense. For example, the `upBy()` message requires an `int` argument specifying what position the receiver should move to, as in the following message-send: `hoverFrog1.upBy(2)`.
- It is common for the protocol of an object to include pairs of messages to provide access to each of its attributes. A getter message and the corresponding setter message together form what is called a pair of accessor messages. For example, in the protocol of `Frog` objects, the message `getColour()` returns the value of the receiver's `colour` attribute, and `setColour()` sets (changes) the receiver's `colour` attribute.
- A message may cause the receiver to send a message to another object to help it carry out the behaviour required. For example, the message-send

  ```
  frog1.sameColourAs(frog2)
  ```

  results in `frog1` sending `frog2` a `getColour()` message in order to determine the colour of `frog2`. This behaviour is termed collaboration.
- A message may cause the receiver to send a message to itself. For example, the message-send

  ```
  frog1.sameColourAs(frog2)
  ```

  results in `frog1` sending itself the `setColour()` message.

LEARNING OUTCOMES

After studying this unit you should be able to:

- use appropriately the terms: object, message, protocol, state, attribute, argument, class, instance, receiver, message-send, message answer, message name, getter message, setter message, subclass, superclass, initialisation, polymorphism;
- use the microworlds in the Amphibian Worlds application to send messages to `Frog`, `HoverFrog` and `Toad` objects, including messages which require arguments;
- use the Accounts World programming tool, to create and send messages to `Account` objects, including messages which require arguments;
- inspect the state of objects;
- describe and explain the superclass/subclass relationship between the `Frog` and `HoverFrog` classes;
- describe the role of accessor messages in an object's protocol;
- discuss how two or more objects can collaborate to perform a task, and draw a sequence diagram to depict the interactions between the objects in the form of messages and message answers.

Glossary

accessor message The general term for either a **setter** or a **getter message**.

argument Extra information supplied with a **message**. For example, when requesting a `Frog` object to change its colour to that of another `Frog` object, it is necessary to provide that other `Frog` object as an argument. This is seen in the **message-send** `frog1.sameColourAs(frog2)`.

A message can have zero, one or more arguments.

There is no argument in the message `getColour()`.

The message `setColour(OUColour.PURPLE)` has one argument (namely `OUColour.PURPLE`) supplying information on which colour is to be chosen.

Two arguments (`yourAccount` and `50`) supply information in the message `transfer(yourAccount, 50)`.

attribute Some property or characteristic of an **object**, such as `position` for `Frog` objects, or `balance` for `Account` objects.

attribute value The current value of an **attribute**. For example, a `Frog` object has the attributes `colour` and `position`. The attribute `colour` of a particular object might have the value `OUColour.BLUE` and the attribute `position` might have the value `1`.

behaviour This term is used to describe the way an **object** responds to the **messages** in its **protocol**.

class A class is a template that serves to describe all instances (objects) of that class. It defines what **attributes** the objects should have and their **protocol** – what messages they can respond to.

Instances of the same class have the same attributes, which are initialised in the same way. They have the same **instance** protocol and respond in the same way to each message.

getter message A message that returns as its message answer the value of one of a receiver's attributes. See **setter message** and **accessor message**.

initialisation The **state** of an **object** when it is first created depends on its initialisation.

inspector An inspector is a tool used in M255 to look at the internal state of **objects** in a system. It lists the **attributes** of an object and displays their current values.

instance An **object** that belongs to a given **class** is described as an instance of that class.

message A message is a request for an **object** to do something. The only way to make an object do something is to send it a message.

For example, the position of a `Frog` object changes when it is sent the message `left()` or `right()`; to obtain information on the value of a `Frog` object's `colour` attribute, you send it the message `getColour()`.

message answer When a **message** is sent to an **object** then, depending on what the message is, a message answer may be returned. A message answer is a value or an object; it is not a message.

Sometimes a message answer is used, sometimes it is ignored. A message answer may be used subsequently as the receiver or argument of another message.

Enquiry messages (getter messages) often return the value of an **attribute**, as with the message `getColour()`, which returns a value such as `OUColour.GREEN`.

message name The name of a **message** does not include any arguments. For example, the name of the message `left()` is `left()`, and the name of the message `upBy(6)` is `upBy()`.

message-send The code that sends a message to an object – for example, `frog1.right()`, which consists of the **receiver** followed by a full stop and then the **message**.

object An object is a software component that has a unique identity and responds to **messages**. Each object has **state** and responds to a particular set of messages (its **protocol**). Thus a `Frog` object (which has little resemblance to a real-world frog) holds information on its position and colour as values of its **attributes** `position` and `colour`.

object-state diagram An object-state diagram represents an **object**. It shows the **class** of the object, its **state** in terms of attribute values, and its **protocol**.

parameter A synonym for **argument**.

polymorphism Any **message** to which **objects** of more than one **class** can respond is said to be polymorphic or to show polymorphism.

For example, both `Toad` and `Frog` objects respond to the message `left()`, but with different behaviours. They also respond to the message `green()`, with identical behaviours.

protocol The set of **messages** an **object** can respond to (understands).

receiver The **object** to which a **message** is sent.

sequence diagram A diagram that depicts the interactions between objects, in the form of **messages** and **message answers**.

setter message A message that sets the value of one of a receiver's attributes. See **getter message** and **accessor message**.

state The values of the **attributes** of an **object** constitute its state. The state of an object can vary over time as the values of its attributes change.

subclass A subclass is a new **class** defined in terms of an existing class (its **superclass**). Instances of a subclass have all the attributes that instances of the superclass have, but may have additional attributes. The protocol of the subclass includes (has at least all the messages of) the protocol of the superclass, but may define additional messages.

For example, `HoverFrog` is a subclass of `Frog`. The protocol of `HoverFrog` objects includes that of `Frog` objects and has, in addition, the messages `upBy()` and `downBy()` to which `Frog` objects cannot respond. `HoverFrog` objects have all the attributes of `Frog` objects (`colour` and `position`) and an additional attribute – `height`.

superclass If B is a **subclass** of A, then A is the superclass of B.

Index