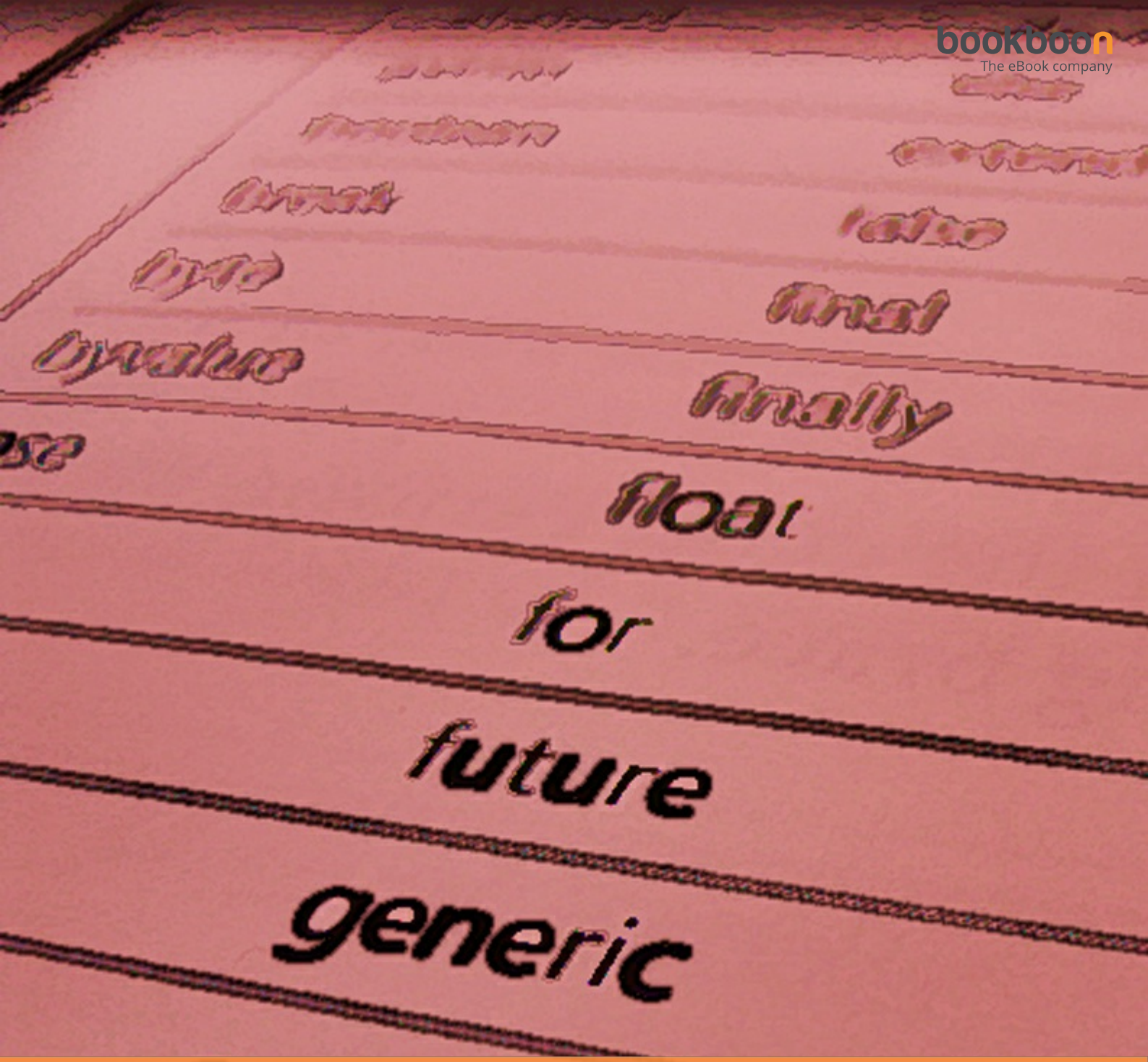


bookboon
The eBook company

Java: The Fundamentals of Objects and Classes

An Introduction to Java Programming

David Etheridge

David Etheridge

Java: The Fundamentals of Objects and Classes

– An Introduction to Java Programming

Download free eBooks at bookboon.com

Java: The Fundamentals of Objects and Classes
– An Introduction to Java Programming
© 2009 David Etheridge & Ventus Publishing ApS
ISBN 978-87-7681-475-5

Download free eBooks at bookboon.com

Contents

1.	**Object-Oriented Programming: What is an Object?**	**6**
1.1	Introduction to Objects	6
1.2	Comparison of OOP and Non-OOP	6
1.3	Object-Oriented Analysis and Design (OOA & D)	9
2.	**A First Java Programme: From Class Diagram to Source Code**	**21**
2.1	Introduction	21
2.2	The Class Diagram for the Member Class	21
2.3	The Java Source Code for the Member Class	22
2.4	Using Member Objects	30
2.5	Summary	35
3.	**Language Basics: Some Syntax and Semantics**	**44**
3.1	Introduction	44
3.2	Identifiers	44
3.3	Primitive Data Types	46
3.4	Variables	49
3.5	Operators	58
3.6	Summary	59

I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com



Click on the ad to read more
Download free eBooks at bookboon.com

4.	**Methods: Invoking an Object's Behavior**	**60**
4.1	How do we get Data Values into a Method?	60
4.2	How do we get Data Values out of a Method?	67
4.3	Method Overloading	68
4.4	The Structure of a Typical Class Definition	70
5.	**Classes and Objects: Creating and Using Objects**	**72**
5.1	Invoking an Object's Constructor	72
5.2	Object Construction and Initialisation of an Object's State	73
5.3	Overloading Constructors	75
5.4	Initialisation Blocks	77
6.	**Collecting Data I**	**78**
6.1	An Introduction to Arrays	78
6.2	Arrays as Data Structures	79
6.3	Declaring Arrays	81
6.4	Creating Arrays	81
6.5	Populating Arrays	82
6.6	Accessing Array Elements	87
6.7	Arguments Passed to the main Method	90



Click on the ad to read more
Download free eBooks at bookboon.com

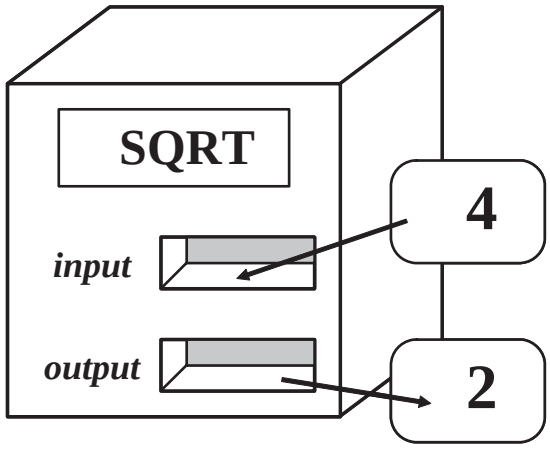
1. Object-Oriented Programming: *What is an Object?*

1.1 Introduction to Objects

While there is a study guide (available from Ventus) that focuses largely on objects and their characteristics, it will be instructive to the learner (of the Java programming language) to understand how the concept of an *object* is applied to their construction and use in Java applications. Therefore, Chapter One (of this guide) introduces the concept of an object from a language-independent point of view and examines the essential concepts associated with object-oriented programming (OOP) by briefly comparing how OOP and non-OOP approach the representation of data and information in an application. The chapter goes on to explain *classes*, *objects* and *messages* and concludes with an explanation of how a class is described with a special diagram known as a *class diagram*.

1.2 Comparison of OOP and Non-OOP

Despite the wide use of OOP languages such as Java, C++ and C#, non-OOP languages continue to be used in specific domains such as for some categories of embedded applications. In a conventional, procedural language such as C, data is sent to a procedure for processing; this paradigm of information processing is illustrated in Figure 1.1 below.



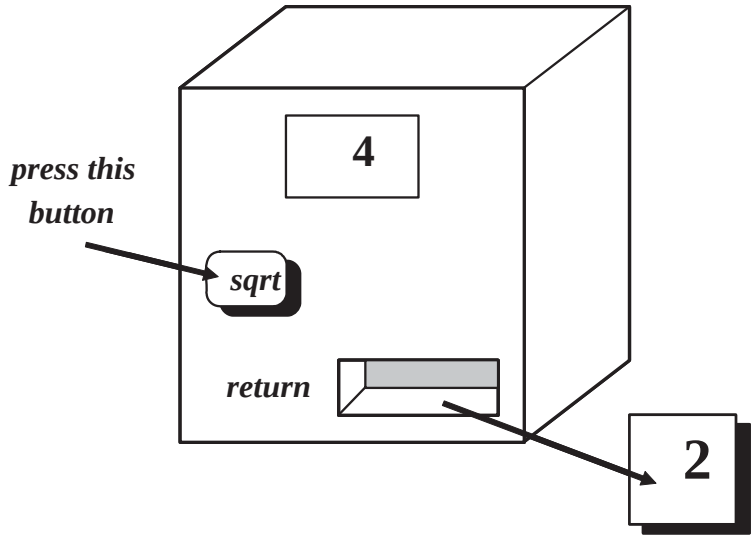
Source: R. A. Clarke, BCU.

Figure 1.1 Passing data to a procedure

The figure shows that the number 4 is passed to the function (SQRT) which is 'programmed' to calculate the result and output it (to the user of the procedure). In general, we can think of each procedure in an application as ready and waiting for data items to be sent to them so that they can do whatever they are programmed to do on behalf of the user of the application. Thus an application written in C will typically comprise a number of procedures along with ways and means to pass data items to them.

Download free eBooks at bookboon.com

The way in which OOP languages process data, on the other hand, can be thought of as the inverse of the procedural paradigm. Consider Figure 1.2 below.



Source: R. A. Clarke, BCU.

Figure 1.2 Passing a message to an object

In the figure, the data item – the number 4 – is represented by the box (with the label '4' on its front face). This representation of the number 4 can be referred to as *the object of the number 4.* This simple object doesn't merely *represent* the number 4, it includes a button labeled **sqrt** which, when pressed, produces the result that emerges from the slot labeled **return**.

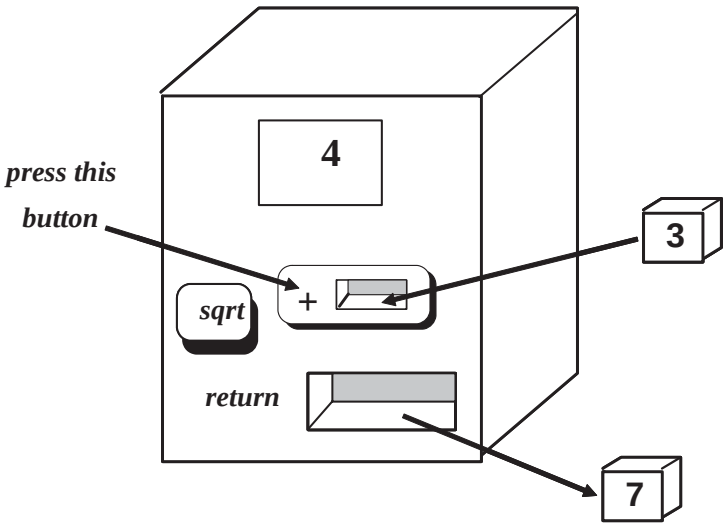
Whilst it is obvious that the object-oriented example is expected to produce the same result as that for the procedural example, it is apparent that the *way* in which the result is produced is entirely different when the object-oriented paradigm considered. In short, the latter approach to producing the result 2 can be expressed as follows.

Send the following message to the object 4: "press the sqrt button"

A *message* is sent to the object to tell it what to do. Other messages might press other buttons associated with the object. However for the present purposes, the object that represents the number 4 is a very simple one in that it has only one button associated with it. The result of sending a message to the object to press its one and only button 'returns' another object. Hence in Figure 1.2, the result that emerges from the 'return' slot - the number 2 – is an object in its own right with its own set of buttons.

Despite the apparent simplicity of the way in which the object works, the question remains: *how does it calculate the square root of itself?* The answer to this question enshrines *the* fundamental concept associated with objects, which is to say that objects carry their programming code around with them. Applying this concept to the object shown in Figure 1.2, it has a button which gives access to the programming code which calculates the square root (of the number represented by the object). This amalgam of data and code is further illustrated by an enhanced version of the object shown in Figure 1.3 below.

Download free eBooks at bookboon.com



Source: R. A. Clarke, BCU

Figure 1.3 The object with two buttons.

The enhanced object (representing the number 4) has *two* buttons: one to calculate the square root of itself – as before - and a second button that adds a number to the object. In the figure, a message is sent to the object to press the second button – the button labeled '+' – to add the object that represents the number 3 to the object that represents the number 4. For the '+' button to work, it requires a data item to be sent to it as part of the message to the object. This is the reason why the '+' button is provided with a slot into which the object representing the number 3 is passed. The format of the message shown in the figure can be expressed as follows.

Send a message that carries the object 3 to the object 4: "press the + button"

When this message is received and processed by the object, it returns an object that represents the number 7. In this case, the message has accessed the code associated with the '+' button. The enhanced object can be thought of as having two buttons, each of which is associated with its own programming code that is available to users of the object.

Extrapolating the principal of sending messages to the object depicted in Figure 1.3 gives rise to the notion that an object can be thought of as comprising a set of buttons that provide access to operations which are carried out depending on the details in the messages sent to that object.

In summary:

- in procedural programming languages, data is sent to a procedure;
- in an object-oriented programming language, messages are sent to an object;
- an object can be thought of as an amalgam of data and programming code: this is known as *encapsulation*.

Download free eBooks at bookboon.com

Whilst the concept of encapsulation is likely to appear rather strange to learners who are new to OOP, working with objects is a much more natural way of designing applications compared to designing them with procedures. Objects can be constructed to represent *anything* in the world around us and, as such, they can be easily re-used or modified. Given that we are surrounded by *things* or *objects* in the world around us, it seems natural and logical that we express this in our programming paradigm.

The next section takes the fundamental concepts explored in this section and applies them to a simple object. Before doing so, however, it is worth making the point that this section is not meant to be an exhaustive exploration of OO concepts: a separate study guide achieves this objective. Suffice it to say that the main purpose of this section is to explain the key concept of encapsulation.

Finally (in this section) it is also worth making the point that, in Java, data - such as the numbers discussed above - do not have to be represented by (Java) objects. They *can* be, but data such as integers are represented by primitive data types, much as in procedural languages. However representing data such as the number 4 as an object provides an opportunity to explain encapsulation.

The next section explores a simple object, in preparation to writing a first Java programme in Chapter Two.

1.3 Object-Oriented Analysis and Design (OOA & D)

1.3.1 What are my Objects?

As might be expected, given that the Java programming language is object-oriented, objects expressed in Java exhibit encapsulation of data values and operations on those values. Therefore because Java is object-oriented, elements of Java differ in their syntax compared with a procedural language such as C. Despite this difference, there are language elements in the Java code that Java objects carry about with them – as a consequence of encapsulation - that are common to other programming languages, whether they are object-oriented or not. Consequently as this guide begins to explore and apply the syntax of Java, some learners may recognize language elements in Java that are similar to their equivalents in other languages. Language elements such as the following may be familiar, depending on the programming experience of the learner:

- declaring and initializing primitive date types;
- manipulating variables;
- making decisions in an **if…then** type of construct;
- carrying out repetitions in **for…next** and **do…while** types of constructs;
- passing arguments to operations (known as *methods* in Java);
- working with arrays and other data structures;
- and so forth.

In fact, much of the semantics and syntax of Java is derived from languages such as C and C++, to the extent that learners with previous experience in non-OO or other OO languages are likely to be familiar with much of it. The principal difference, when using an OO language such as Java to write application logic, is that the language elements, such as those exemplified in the list above are encapsulated in an entity known as an object.

Download free eBooks at bookboon.com

Embarking on a course in Java will require a learner who is experienced in a procedural language to make the transition from a non-object-oriented language to an object-oriented one. Learners who have some experience of procedural languages should not be alarmed: this transition is not as difficult as it may seem! For the novice programmer, this guide begins with objects from the outset. In either case, once some of the essential concepts of object-oriented programming in Java have been grasped, they can be applied to almost any Java object. In short, the way that most objects are structured is common to them all. In other words, we can extrapolate from a relatively small number of concepts and apply them to any Java object.

The next few sub-sections work towards the description of a simple object in a language-independent way; actual Java code does not appear until Chapter Two. This approach is intended to make the point that OOA & D is *language-independent*. When the objects associated with an application are analysed, described and documented, including diagrammatic documentation, the analysis can be turned into *any* target OO language. In this guide, of course, the outcome of analysis and design will be translated into Java source code.

1.3.2 How do I know what my Objects are?

As has been established, object oriented analysis and design (OOA & D) models everything in the world around us in terms of software entities called objects. For example, we could model a banking application as comprising a number of objects including:



Click on the ad to read more
Download free eBooks at bookboon.com

- a *customer* object;
- a *current account* object associated with a particular customer object;
- a *savings account* object associated with the same customer object;
- the single *bank* object associated with customer objects;
- and so on.

Similarly for an on-line media store, analysis might show that the following objects exist:

- a registered member of the Media Store: there will be many of these objects;
- the Media Store itself: there will be only one of these objects;
- each member's virtual membership card;
- a DVD object: there will be many of these objects;
- a games object: there will be many of these objects;
- a CD object: there will be many of these objects;
- a book object: there will be many of these objects;
- and so forth.

An application that supports the business operations of such a media store will be used throughout this guide to illustrate how Java can be used to meet the requirements of a realistic business application and provide examples of concepts and language elements. Throughout the guide, the author's Media Store application will be referred to as the guide's 'themed application'.

In general, the outcome of the OOA & D process for a set of business requirements results in expressing the design as encapsulating data (*attributes*) and operations on these data (*behavior*) into objects. The details of OOA & D methodologies are outside the scope of this guide, apart from the use of a simple diagrammatic technique to describe objects; this chapter and will conclude with such a diagram for one of the objects of the themed application.

Returning, for a moment, to the bank example outlined at the beginning of this sub-section, let us assume that the current account object has an attribute called **`overdraftLimit`** and that its behaviour is used to set this attribute to a value of £500. Similarly, let us assume that the customer object has an attribute called **`name`** and that its behaviour is used to set the value of the attribute to "David Etheridge". Thus an object's attributes (or data) and hehaviour (or operations on these data) are closely linked. This linking or bonding of data and operations is, as we have already established, known as encapsulation.

There is a further implication of encapsulation that hasn't been explained as yet. The nature of the bond between data and operations is such that an object's data values are (usually) *only* manipulated by using the object's behaviour. In other words, an object's data values are not *directly* accessed; instead they are accessed via the object's behaviour. In short, a useful way of summarising the access to an object's data values is to think of an object as comprising *private* data values and *public* behaviour to manipulate these data values.

Download free eBooks at bookboon.com

Another consequence of the OOA & D approach is that the implementation details of an object's data are hidden from other objects that wish to use the data values of the object. This means that a *user* object only needs to know the behaviour that the *provider* object offers. Thus, we can think of the provider object's behaviour as a kind of a contract that the object offers to its user objects. As we will see in due course, the behaviour that an object offers to its user objects is known as its *interface*. All that a user object needs to know is *what* behaviour the provider object provides to manipulate its data values; user objects do not need to know *how* the provider object's behaviour is implemented. This means that implementation details can change, without changing the provider object's interface.

This property of objects is known as *information hiding*, another manifestation of encapsulation. This means that although an object may know *how* to communicate with another object, via the other object's *interface*, the object does not need to know how the attributes and behaviour of the other object are implemented: i.e. implementation details are hidden within the provider object. Consider an analogy: one might know how to drive a car without knowing how the internal combustion engine works! Or, in the example shown in Figure 1.3, the user object – the object for the number 3 – does not need to know how the provider object – the object for the number 4 – implements its '+' operation; all that it needs to know is that the operation is available to the outside world – i.e. it is public - and it needs a value to be passed to it in the message that asks the provider object to press its '+' button. This means that a further outcome of OOA & D is a model of the *communication* amongst objects. For example, the bank object might wish to send a message to the current account object to alter the value of **`overdraftLimit`**.

To summarize and, perhaps, simplify the OOA & D methodology, *any* application domain can be analysed and modelled in terms of the objects it comprises, where each object (in that domain) has *attributes* and *behaviour*.

1.3.3 Classes and Objects

Just when the learner thinks that they have grasped the, perhaps new, concept of an object, along comes a heading that introduces another new term: that of *the class*. The purpose of this section is to refine and define these two terms: they operate, as it were, in tandem.

Consider a simple analogy: David and Annette Etheridge's cat – called Jasmine - can be regarded as an object or *instance* of the *class* **`Cat`**, where the *instance name* is **`jasmine`** and where the class is a template or blueprint for *all* cats of the species of animal known as 'cat'. Thus, Mother Nature uses her class **`Cat`** as a template to create *every* domestic cat in existence. The distinction between a c*lass* and an *instance* or *object* of that class is an important one: a *class* is the blueprint for *all objects* (or *instances*) of that class. Similarly, Mother Nature uses her one and only **`Aardvark`** *class* to create all aardvarks, that is all instances of aardvarks that walk the earth.

> *A single class is used to create as many instances (or objects) of that class as are needed in an application.*

Download free eBooks at bookboon.com

For example, referring again to the bank application, we could use the class called **Customer** to create or *instantiate* as many customers as are needed, such that each customer object can subsequently be given values of the attributes defined in the class.

Similarly let us assume that one of the attributes of the class called **Cat** is called **mood** and that one of its behaviour elements is used to set the value of the attribute named **mood** of a particular cat. Thus we can use the template for a cat, i.e. the class called **Cat**, to create an instance of the class **Cat** with the name **jasmine** and make use of its behaviour to set the value of its attribute **mood** to "grumpy". Similarly, we can use the class **Cat** to create another instance of the class **Cat** with the instance name **florence** and use *its* behaviour to set the value of its attribute **mood** to "cool". This second instance of the class **Cat** has a different value of the attribute called **mood**. As we will discover in due course, two (or more) objects can have the same values of some or all of their attributes. However for the purposes of the present example, our two cat objects (named **jasmine** and **florence**) – created from the same class - differ in the value of their **mood** attribute. Thus, in terms of encapsulation, our two cat objects carry about with them the 'code' to express the value of their attribute named **mood** to be "grumpy" and "cool" respectively. Finally, it should be noted that our two objects of the class **Cat** are given *different* instance names to distinguish one from the other and to affirm their separate existence.

The next sub-section will analyse a simple class in order to show how its analysis is documented.

1.3.4 Analysis and Design of the Member Class

In this section, we will work with one of the classes of the themed application introduced in the previous section. The class is given the name (known as its *identifier*) **Member** to distinguish it from other classes in the application.

Based upon the discussion in the previous sub-section, we know that the class called **Member** can be used, in some way as yet to be explained, to create objects of the class **Member**. Before we work with the class called **Member**, let us return to our analogy. Remember that David and Annette Etheridge have a cat called Jasmine, created from the class called **Cat**. Thus the class is of *type* **Cat** and the particular *instance* of the class **Cat** is an object called **jasmine** (the reason for the lower case 'j' will be explained in a moment). David and Annette Etheridge used to have a cat called Florence: (Florence has gone through the great cat flap in the sky!) If both cats were alive today, David and Annette Etheridge would have two *instances* of the class **Cat** called **jasmine** and **florence** respectively. Thus the template for the two cats **jasmine** and **florence** is the class **Cat** (which has been used, by Mother Nature, to create two cats). Note that class type names begin with a capital letter: this is a convention used by the Java developer community. Thus we have the class **Cat**, *not* **cat**. Whilst class names *always* begin with a capital letter, instances of a class begin with a lower case letter. Thus we have **florence** and **jasmine**, *not* **Florence** and **Jasmine**, as identifiers for the two instances of the class **Cat**. The example that follows will further illustrate these naming conventions.

Download free eBooks at bookboon.com

Returning to the class called **Member**, our task is to define (some of) the attributes and behaviour of the class, so that when objects (or *instances*) of the class **Member** are created (or *instantiated*) we can give values to these attributes by making use of the behaviour of the class.

For the purposes of our example, we will identify (some of) the attributes and behaviour of the class named **Member** as follows.

Attributes of the Member Class

N. B. Attributes are categorized as one of a *type*: this can be a primitive data type or a class type as shown in the table on the next page.



Click on the ad to read more

Download free eBooks at bookboon.com

Attribute	Identifier	Type	Comments
first name	firstName	string	A string will be used to store the value of this attribute
last name	lastName	string	Similarly, a string is used
user name	userName	string	
password	password	string	
membership card	card	MembershipCard	The type is a *class*, because the member's card is an object in its own right: note, therefore, the capital 'M' in the type MembershipCard

The first column lists the properties or attributes of the class, expressed in a natural language. The second column implies that each attributes is given a name, known as its *identifier*. Note the convention for identifiers: for example, **firstName**, *not* **firstname**. In short, identifiers that comprise compound words begin with a lower case letter and all subsequent words in the compound word are capitalised. The third column gives the *type* of the attribute, i.e. what kind of data value it represents.

A critically important consequence of identifying types in an OOP language is that they can either be of the primitive data type, such as integer or string and so on as in a typical procedural programming language, or they can be a *class* types. For example, the list of attributes above includes one which is a class type. The reason for this is that, intuitively, a member's membership card is an object in its own right, with its own attributes and behaviour. We cannot represent such a complex entity in a procedural language by using primitive data types. Thus, the third column illustrates that OO design represents the non-primitive data types in an application as objects of one of a type. This feature is one of the key differences between a non-OO programming language and an OO programming language and gives that latter vastly superior flexibility compared to the former when it comes to identifying the attributes of the complex entities associated with an application.

Behaviour Elements of the Member Class

In order to represent a real-world instance of the class **Member**, we need to identify the behaviour that is used to manipulate the values of its attributes. The syntax that is used to describe behaviour in a language-independent way is as follows:

```
behaviourName( a comma-separated list of parameterName: parameterType ): returnType
```

where **behaviourName** is an arbitrary but meaningful name for the behaviour; note that it begins with a lower case letter.

Download free eBooks at bookboon.com

The terms *parameter* and *return type* are explained more fully later. For the time being, the **parameterName** can be thought of as an arbitrary (but meaningful) name of the **parameterType**, whose value is passed as an *argument* to the behaviour when a message is sent to the object to ask for the behaviour to be executed. The **returnType** is the type (if any) that the behaviour supplies when it completes its execution. In this sense, the behaviour is said to *return* a type when it is executed. For example, figures 1.2 and 1.3 show an object whose behaviour returns an object that represents a number.

In the context of behaviour, a *very* important consequence of the OO approach is that parameters and return types can be primitive data types *or* class types. Thus, behaviour can be designed so that primitive data types and/or objects can be passed to it as arguments. An argument that is an object is passed to the behaviour using the same mechanism as is used to pass a primitive data type to it. Similarly, behaviour can return an object or a primitive data type, but not both at the same time. For example, Figure 1.3 shows an object passed as an argument to the '+' behaviour. Similarly, the '+' behaviour could have been designed to accept an integer argument passed to it and even return an integer when the button is pressed to execute the behaviour. In practice, actual objects are not passed as arguments; instead, a *reference* to the object is passed. We will return to this concept in a later chapter.

Before we go any further, let us use the syntax for expressing behaviour to consider an example of using or *executing* behaviour by passing an argument to it. Let us assume that one of the behaviour elements of our **Member** class is defined as **setPassword(pword: string)** – note that there is no return type - where **setPassword** is the name of the behaviour, and **pword** is the identifier (i.e. the name) of its only parameter which is of the **string** data type. Thus, when the behaviour is executed, it will expect a value of the **string** type to be passed to it in the form of a message. A programming statement such as the following illustrates the concept of passing such a value to the behaviour when it is executed:

setPassword("abc999");

The programming statement above sends a message to the **Member** object and asks its run-time system to execute the behaviour and, in doing so, the statement is used to *pass the argument to the behaviour*. In such a statement, we can think of the pair of brackets ( ) as acting as a 'payload' for the behaviour in that it provides a simple mechanism to pass values of data or (references to) objects to the behaviour so that they can be used by the code associated with that behaviour.

Behaviour can be defined such that it does not expect arguments to be passed to it. In such a case, the payload is empty when the behaviour is executed. Consider, for example, the following programming statement:

getPassword( );

The message to the **Member** object to request the execution of the behaviour **getPassword** has been written so as not to expect value(s) of argument(s) to be passed to it. The behaviour **getPassword** is merely programmed to 'get' the current value of an attribute; it does not need any data or objects to do this.

We will use programming statements such as those above in the next chapter, when we write the class definition for the **Member** class.

Download free eBooks at bookboon.com

In the case of the behaviour **setPassword(pword: string)**, we have assumed, intuitively, that the behaviour does not return a value of a type when it is executed. On the other hand, let us assume that the behaviour **getPassword** *does* return a value when it is executed and that it is correctly described as **getPassword( ): string** to imply that the behaviour returns a value of the type **string**. Thus, the statement

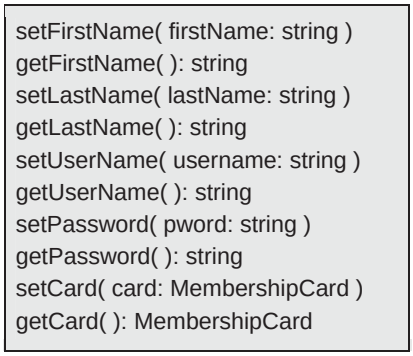
getPassword( );

actually produces a result in that it returns a **string** value that we should be able to output in some way.

The behaviour **setPassword(pword: string)** and **getPassword( ): string** (discussed above) leads to a fuller description of the behaviour of the class **Member**. The general syntax used to describe behaviour can be used to describe the specific behaviour of the **Member** class as shown on the next page, where the name of the behaviour is followed by its parameters in parenthesis and its return type (if any) following a colon.



Click on the ad to read more
Download free eBooks at bookboon.com



Note that as a general –but not universal – rule, each attribute has associated with it a pair of behaviour elements **setXxx** and **getXxx**, where **Xxx** is the capitalized name of the attribute. Broadly speaking, the reason for this is so that we have sufficient behaviour to be able to *set* the value of an attribute and also to *get* (i.e. find out) the value of the attribute at any point in a programme.

Note, again, the way that behaviour names are written: they begin with a lower case letter and can be compound words, where words other than the first begin with a capital letter.

Referring to the list above, the **setFirstName** behaviour can be used to *pass an argument* of type **string** with an identifier **firstName** to an object of the class **Member** with the purpose of setting the value of the object's **firstName** attribute to the value of the argument. The **getFirstName** behaviour can be used to find out the current value of the attribute **firstName** in that the behaviour is defined to return the value of the attribute **firstName** as a **string**. A similar analysis applies to the next six behaviour elements in the list.

The purpose of **setCard** is to pass the object reference **card**, of the class type **MembershipCard**, as an argument to the method in order to set the value of the attribute with the identifier **card** to refer to a **MembershipCard** object. Invoking this method sets the value of the attribute **card** to the reference to a previously-created **MembershipCard** object passed as the only argument to the behaviour. In effect, this behaviour element *associates* a member of the Media Store with a membership card. We will see, as the themed application develops in later chapters, that the behaviour **setCard** is used to give a member his or her (virtual) membership card.

The purpose of **getCard** is to return the current value of the attribute **card** as an object reference of the **MembershipCard** type. Invoking **getCard**, therefore, returns a reference to a **MembershipCard** object. In effect, this behaviour retrieves the member's membership card so that transactions can be carried out with it.

It should be noted that the list of behaviour elements of the class **Member**, shown above, illustrates that a behaviour element can work with values of primitive data types and/or class types when passed as arguments. Similarly a behaviour's return type can be a primitive data types or a class types. In short, the use of objects as attributes, arguments and return types gives OO programming languages an enormous advantage over non-OO programming languages in that the former are more easily and readily used to represent actual things or entities in the world around us compared to the latter. In practice, as we will find out in due course, an OO language can also represent *abstract* entities in an application.

Download free eBooks at bookboon.com

The list of behaviour elements of the **Member** class shows that the (arbitrary but meaningful) identifiers of the parameters associated with behaviour elements *may* be the same as those of the attributes that the behaviour manipulates. We will address this apparent clash when the OOA & D of our **Member** class is translated into Java source code in the next chapter. However it should be noted that it is not mandatory for identifiers of parameters and attributes to be the same.

Let us define some terms at this point.

- *Parameter names* are arbitrary but meaningful identifiers for the types that are passed to behaviour. Behaviour may or may not have parameters.
- *Arguments* represent actual values *passed* to behaviour when it is executed so that it can use them for some specific purpose within the implementation of that behaviour. The type of an argument can be one of the primitive data types, defined in the implementation language of the class, or it can be a class type.
- The *return type* is the type that the behaviour returns when the behaviour is executed. The type can be one of the primitive data types, defined in the implementation language of the class, or can be a class type. On the other hand, behaviour may have no return type; i.e. it does not return a value when it is executed.

It can be seen that the first behaviour shown in the list for the **Member** class has no return type, the second returns a primitive type and the last returns a class type. It seems reasonably intuitive that the **setFirstName** behaviour is likely to need an argument to do its work and that the **getPassword** behaviour is not likely to need an argument but it *is* likely to return a value – in this case, a **string** value.

Remember that the attributes and behaviour in the lists above are a *language-independent* way of describing these two aspects of the class **Member**. However they are merely lists: what we need now is a diagram that helps us to design the class of our **Member** objects.

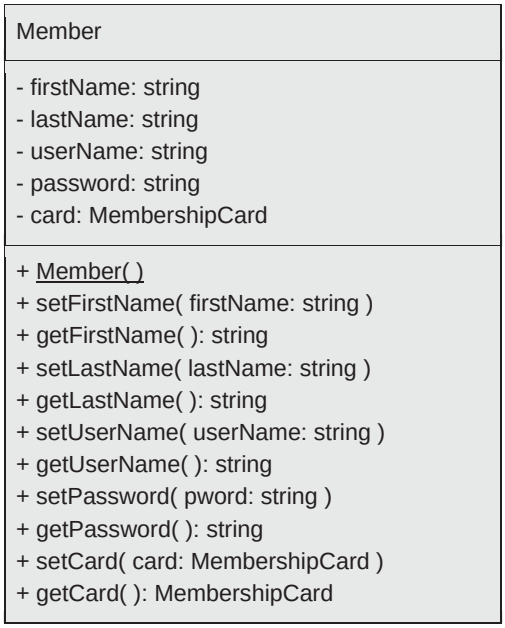
The next section summarises the attributes and behaviour of the class **Member** in terms of a diagram known as a *Class Diagram*. Class diagrams are part of the Unified Modeling Language methodology (UML) for OOA & D and are a language-independent way of describing a class.

1.3.5 The Class Diagram of the **Member** Class

Up to this point in the chapter, we have side-stepped any discussion about exactly *how* an object is created from its class. Just as Mother Nature knows how to create objects of the class **Cat** from her template for the species that we call ‘cat’, the OOP run-time system needs a way of making or *constructing* objects of the class **Member** and storing them in some convenient place in (computer) memory. An OOP language uses an entity known as a *constructor* to construct objects of a class. Thus in addition to attributes and behaviour, one or more constructors form part of a class diagram. We will explore constructors in more detail in later chapters. For the time being, we will make the constructor for the **Member** class straightforward.

Download free eBooks at bookboon.com

The first section of a class diagram contains the class name, the second section lists the attributes and the third section lists constructors and behaviour. Thus, the class diagram of the **Member** class derives directly from the attributes and behaviour discussed earlier in this chapter, with the addition of a no-arguments constructor: it is shown below.



Note that constructors are underlined in class diagrams.

In the diagram, the qualifier '-' mean *private* and the qualifier '+' means *public*, so that access to data values conforms to the principal of encapsulation discussed earlier.

A further point about class diagrams should be borne in mind at this point: a class diagram identifies types, parameter and attribute in a language-independent way; its purpose is not to give implementation details of behaviour – this aspect of OOA & D is language specific and it outside the scope of a class diagram. However, as we will se in the next chapter, a class diagram contains sufficient information to enable the programmer to *declare* attributes, constructors and behaviour elements. The details about the implementation of the elements of a class are obtained from other aspects of OOA & D and the business requirements of the application.

The next chapter takes the class diagram above and explains how the information in it translated into Java source code.

Download free eBooks at bookboon.com

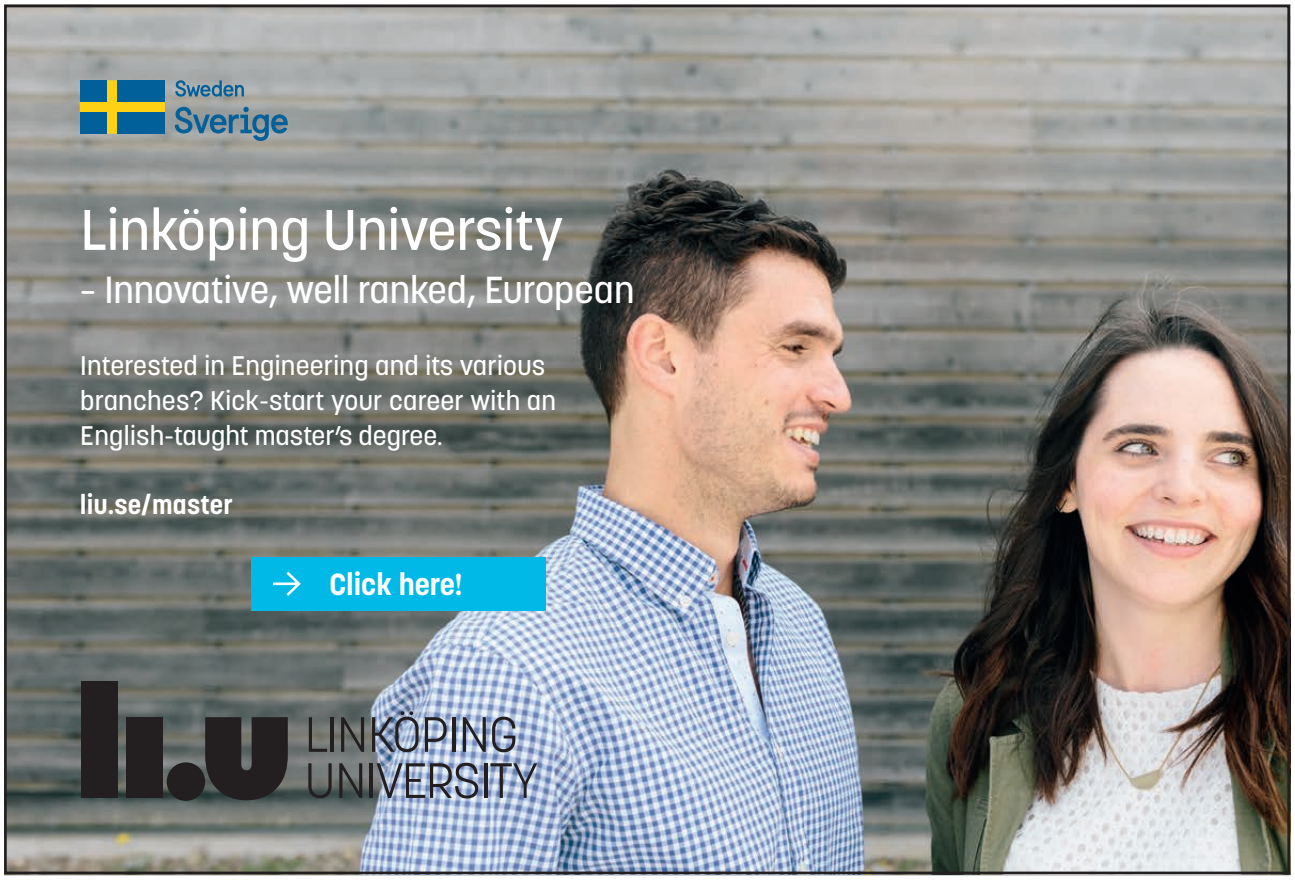
2. A First Java Programme: From Class Diagram to Source Code

2.1 Introduction

The aim of Chapter Two is to take the simple class diagram shown at the end of Chapter One and explain how it is translated into Java source code. The code is explained in terms of its attributes, constructor and behaviour and a test class is used to explain how its constructor and behaviour elements are used.

2.2 The Class Diagram for the Member Class

Before we embark on our first Java programme, let us recall the class diagram with which we concluded Chapter One. The class diagram is reproduced in Figure 2.1 below, with the omission of the constructor: this is to keep the code simple to begin with. We will replace the constructor in the class diagram and provide code for it later in this chapter.



Click on the ad to read more

Download free eBooks at bookboon.com

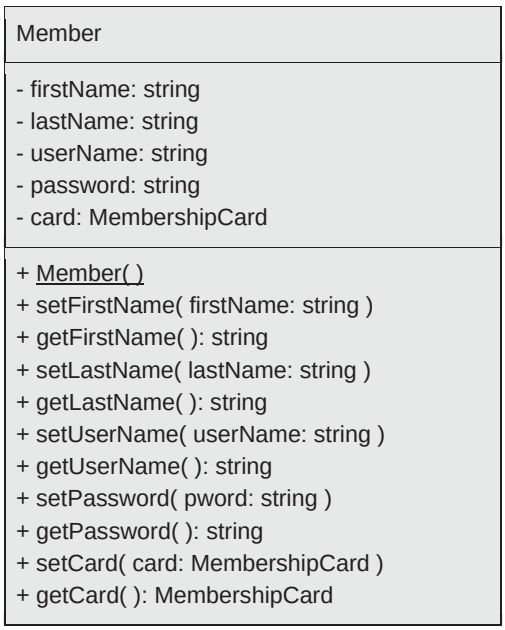


Figure 2.1 Class diagram for the Member class

In Figure 2.1, let us be reminded that the qualifier '-' means *private* and the qualifier '+' means *public*. The purpose of these qualifiers will be revealed when we write the code for the class.

The next section explains how the information in the class diagram shown in Figure 2.1 is translated into Java source code.

2.3 The Java Source Code for the Member Class

Remember that, in general, a class definition declares attributes and defines constructors and behaviour. The Java developer concentrates on writing types called classes, as a result of interpreting class diagrams and other elements of the OOA & D of an application's domain. The Java developer also makes extensive use of the thousands of classes provided by the originators of the Java language (Sun Microsystems Inc.) that are documented in the Java Applications Programming Interface (API).

> The API can be downloaded from:
> http://java.sun.com/javase/downloads/index.jsp

We have established that classes typically comprise attributes and the behaviour that is used to manipulate these data. Attributes are implemented, in Java, as *variables*, whose value determines the condition or *state* of an object of that class and behaviour elements are implemented using a construct known as a *method*. When a method is executed, it is said to be *called* or *invoked*.

> Collectively, variables and methods of a class are often referred to as *members* or *fields* of that class.

Download free eBooks at bookboon.com

As has been mentioned earlier, an instance of a class is also called an object, such that, perhaps somewhat confusingly, the terms *instance* and *object* are interchangeable in Java. The requirement to create an instance of a class from the definition of the class gives rise to a fundamental question: *how do we actually create an instance of a class so that its methods can be executed?* We will address this question in this section.

One of the components of a class, which we haven't explained fully so far in the discussion of the **Member** class, is its constructor. A constructor is used to create or construct an instance of that class. Object construction is required so that the Java run-time environment (JRE) can respond to a call to an object's constructor to create an actual object and store it in memory. An instance does not exist in memory until its constructor is called; only its class definition is loaded by the (JRE). We will meet the constructor for the **Member** class later.

Broadly, then, we can think of the Java developer as writing Java classes, from which objects can be constructed (by calling their constructors). Classes are to objects as an architect's plan is to a house, i.e. we can produce many houses from a single plan and we can construct or *instantiate* many instances from a single template known as a class. Given that objects can communicate with other objects, this gives the developer the means to re-use classes from one application in another application. Therefore, with Java object technology, we can build software applications by combining re-useable and interchangeable objects, some of which can be standardised in terms of their interface. This is probably the single-most important advantage of object-oriented programming (OOP) compared with non-OOP in application development.

We are now at the stage when we can translate the class diagram for the **Member** class into Java source code, often shortened to 'code'. The code that follows is the class definition of the class named **Member** but includes only some of the attributes and methods that do not involve object types: this is to keep the example straightforward. The reason for this restriction is that if we were to declare attributes or parameters of the **MembershipCard** class type in the class **Member,** as required by the class diagram, the Java compiler would look for the class definition of the class **MembershipCard**. In order to keep the example straightforward, we will only write the class definition for the class **Member** for the time being; we will refer to the class definition of the class **MembershipCard** in a later chapter. Thus, in this section, we will work with a single class that includes only primitive data types; there are no class types included in the simplified class diagram.

In order to make the example code even more straightforward, the class diagram is further simplified as shown in the next diagram. The class diagram that we will translate into Java code declares two variables and their corresponding 'setter' (or *mutator*) and 'getter' (or *accessor*) methods, as follows.

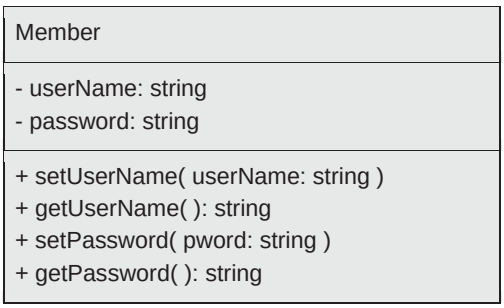


Figure 2.2 Simplified class diagram for the Member class

Download free eBooks at bookboon.com

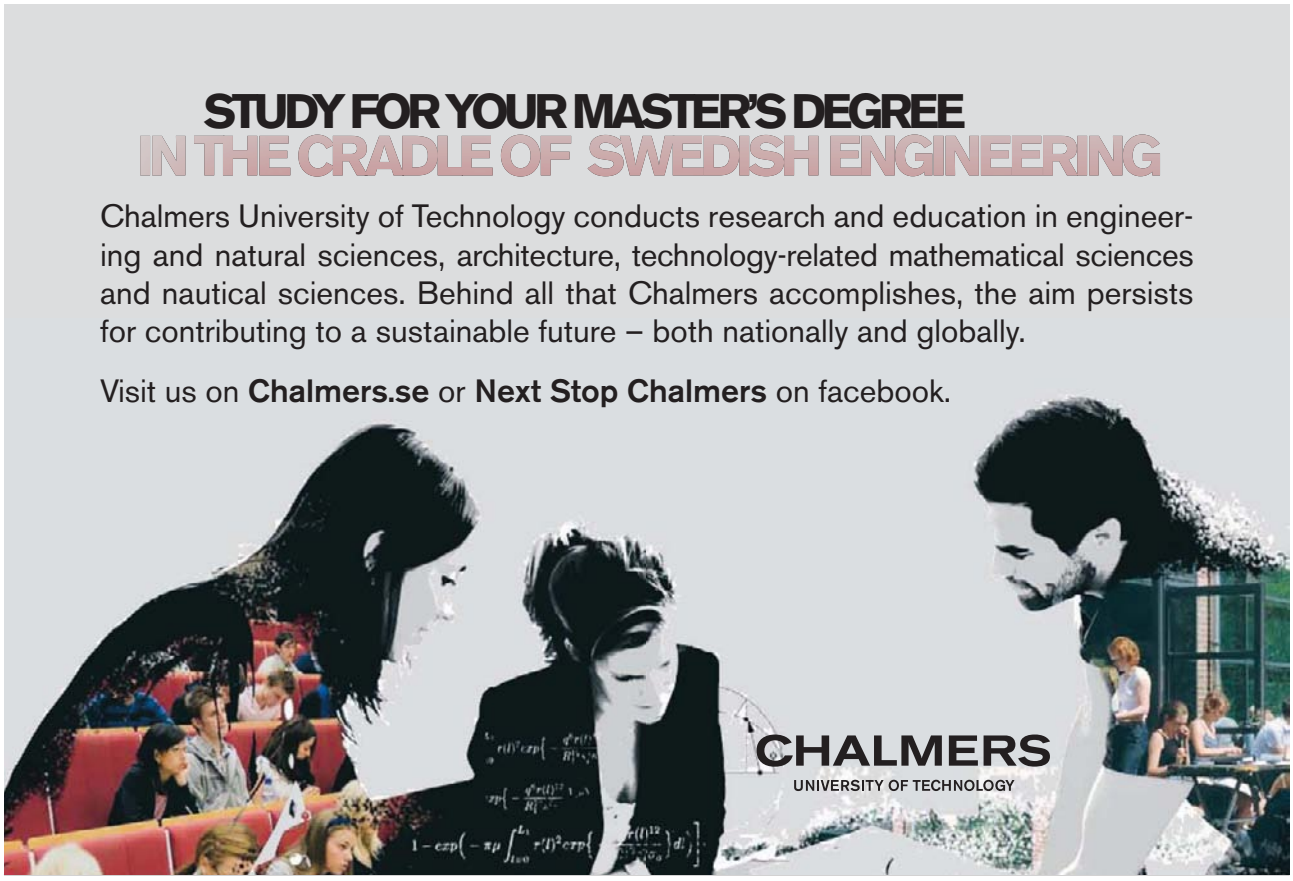
The reason for the simplification (of the full class diagram) is so that the class definition can be more easily understood, compared to its full definition. In short, we well keep our first Java programme as simple as possible.

In the class definition that follows below, ‘ // ‘ is a single-line comment and ‘ /** … */ ‘ is a block comment and, as such, are ignored by the Java compiler. For the purposes of the example, Java statements are written in bold and comments in normal typeface.

```
// Class definition for the class diagram shown in Figure 2.2. Note that the name of
// the class starts, by convention, with a capital letter and that it is declared as public.
// The first Java statement is the class declaration. Note that the words public and
// class must begin with a lower case letter.

public class Member { // The class declaration.

    // Declare instance variables first. Things to note:
    // String types in Java are objects and are declared as ‘String’, not ‘string’.
    // The qualifier 'private' is used for variables.
    // 'String' is a type and 'userName' and ‘password’ are variable names, also
    // known as identifiers. Thus, we write the following:
```



Click on the ad to read more
Download free eBooks at bookboon.com

```
private String userName;
private String password;
// Note that, in the two statements above, we have only declared these two
// variables, they have not been initialized. In fact they will, at first, be
// initialized to their default values when the constructor is invoked. If the
// constructor sets the variables, their default values will be overwritten
// accordingly.

// Constructors are defined next in a class definition. In this example, the
// class includes a no-argument constructor that does nothing; i.e. the code
// block of the constructor is empty (of Java statements). N. B.  The name of
// an object's constructor is the same as its class.  Note the way that it is
// documented.
/**
* The default constructor.
* Note that there are no arguments passed to this constructor.
* Constructors do not declare a return type.  When its invocation completes, a
* constructor returns a reference to the instance created in memory.
*/
public Member( ) {

        // Typically, one of the tasks of a constructor is to initialize variables.
        // In this case, the absence of code results in an instance created and
        // stored in memory, whose variables are initialized to their default
        // values when this constructor is called.

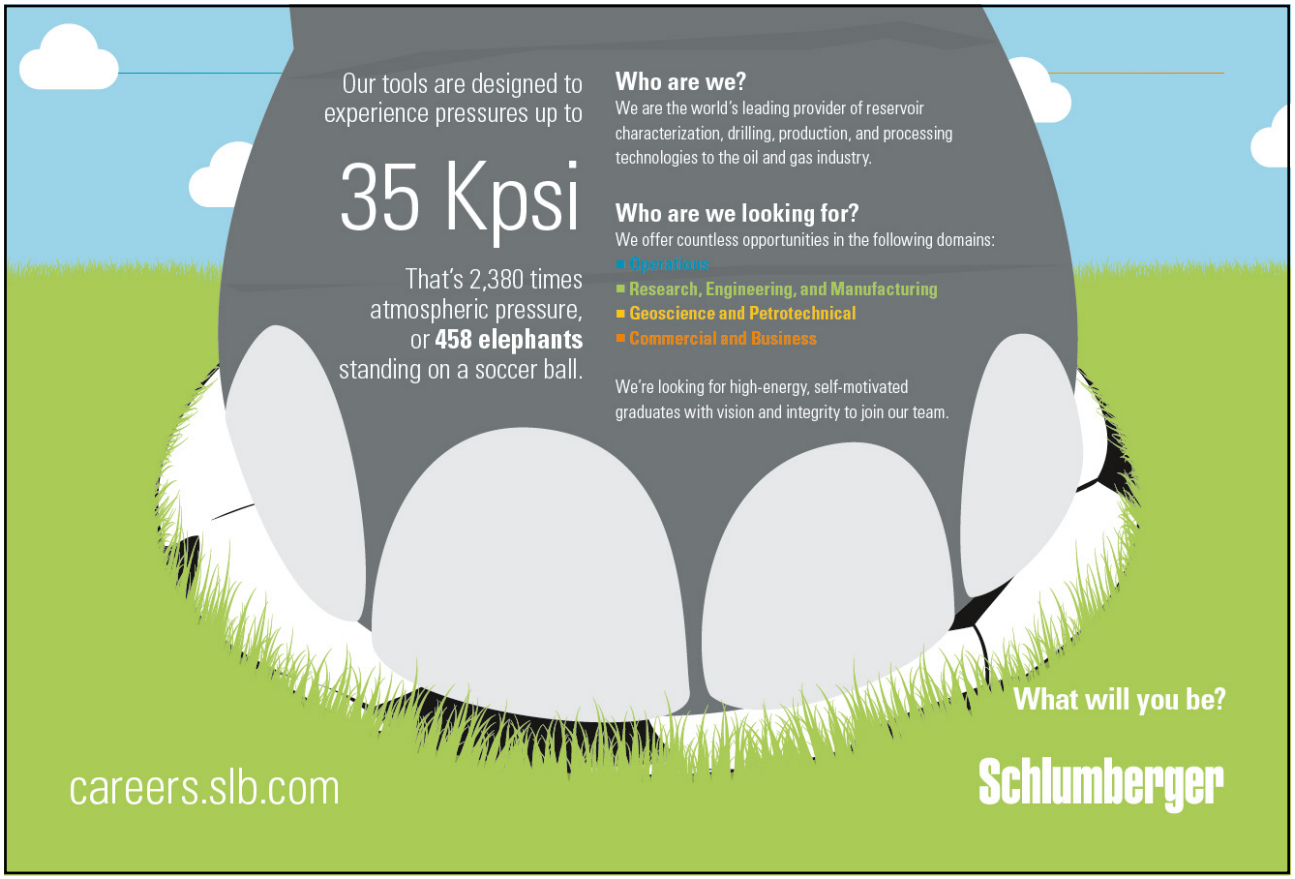
} // End of constructor.

// Method implementations are next in a class definition.
// Things to note:
// Methods are public; the return type must be stated and, is declared as void
// if there is no return type.
// The comma-separated list of parameters (if any) are in parenthesis.
// A method definition begins with the method declaration, followed by the
// body of the method's implementation.
// Note the convention for method names: they begin with a lower case letter
// and are meaningful in that they should reflect the function of the method.
// Multiple words can be used, where all but the first word begins with a
// capital letter.  Note how methods are documented, including the use of '@"
// tags.  We will find out why these tags are useful in due course, when we
// explore documentation more fully in a later chapter.
```


Download free eBooks at bookboon.com

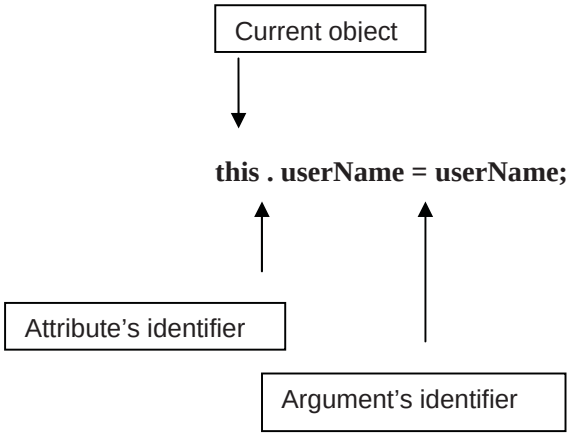
```
/**
 * Mutator for the variable userName.
* @param  userName  The member's user name.
*/
public void setUserName( String userName ) {

        // The purpose of calling setUserName, from an object that uses an
        // object of the class Member, is to assign the value of the method's
        // argument called userName to the value of the attribute called
        // userName.
        // The 'this' refers to the current object and the ' . ' selects the variable
        // called userName from the current object.
```




Click on the ad to read more

Download free eBooks at bookboon.com




```
        // Note that, in this method definition, the parameter's identifier is the
        // same as that of the variable it is modifying.  It is permissible to do
        // this in Java, but it is important that you know which 'userName' is
        // which.  The compiler knows which is which.

    }  // End of definition of setUserName.

    /**
    * Accessor for the variable userName.
    * @return  userName  The value of the variable userName.
    */
    public String getUserName( ) {

        // The next statement returns the current value of userName when the
        // method is called.
        return userName;

    }  // End of definition of getUserName.

    /**
    * Mutator for the variable password.
    * @param  pword  The member's  password.
    */
    public void setPassword( String pword ) {

        // In this method definition, the name of the parameter is different
        // from the name of the variable that it sets.  Therefore, in the next
        // statement, we don't need to refer to the current object with 'this'.
        password = pword;

    }  // End of definition of setPassword.
```


Download free eBooks at bookboon.com

```
    /**
    * Accessor for the variable password.
    * @return  password  The value of the variable password.
    */
    public String getPassword( ) {

        return password;

    }  // End of definition of getPassword.

}  // End of class definition of Member.
```

Inspection of the source code (above) shows that it conforms to its class diagram in that it implements (i.e. provides the code for) two pairs of methods associated with setting and getting the value of each of the variables **userName** and **password**. Clearly, this version of the class definition of the **Member** class is a very simple one, but it is, nevertheless, a true class in that it has variables and methods: it will also have a constructor in a moment.

It should be noted that the two setter methods and the two getter method of the class **Member** are very straightforward in that the former sets the value of a variable and the latter gets the *current* value of a variable using a single Java statement. As we will see in due course, the body of a typical method will usually comprise many more than one Java statement.



Click on the ad to read more

Download free eBooks at bookboon.com

At this point, it will be instructive to compare the class diagram for the class **Member** with the source code in an attempt to see how Java programming statements correspond to elements of the diagram.

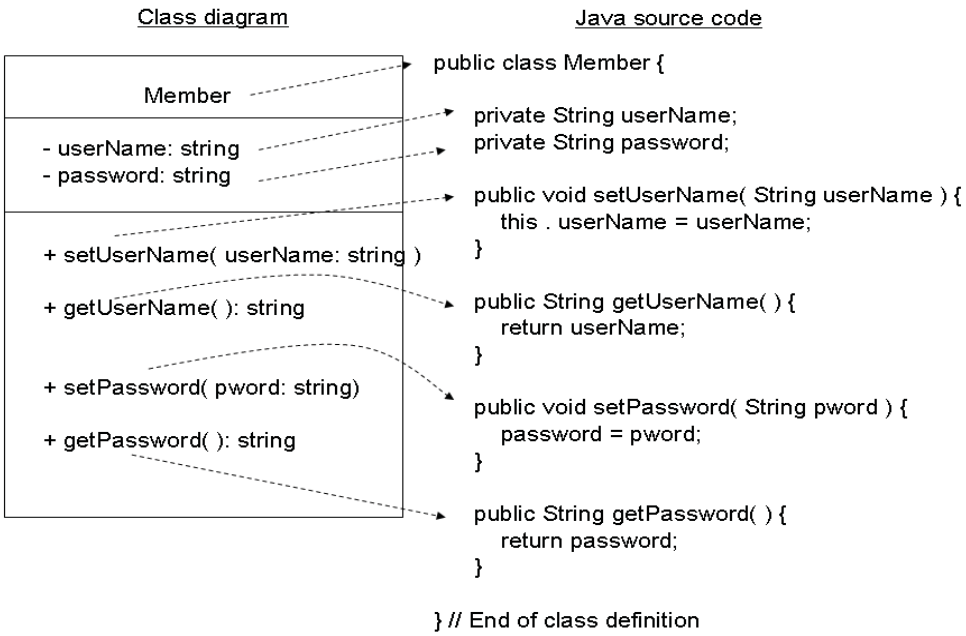


Figure 2.3 Mapping the class diagram to the class definition

The constructor for the class **Member** has been omitted from Figure 2.3, for the sake of simplifying the example. In practice, however, an object's constructors are included in the class diagram. Accordingly, the modified class diagram follows next.

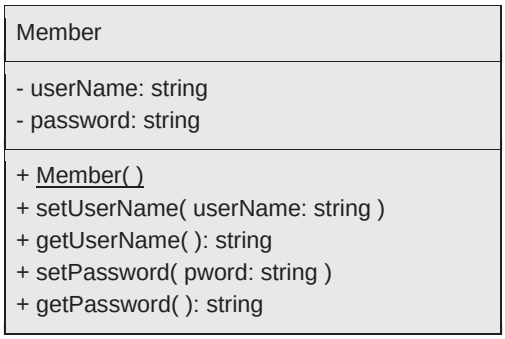


Figure 2.4 The class diagram that includes the constructor

Now that we have written the source code for a simplified version of our **Member** class, we can progress and find out how we invoke its methods. This is the subject of the next section.

Download free eBooks at bookboon.com

2.4 Using Member Objects

So far we have interpreted a simplified version of the full class diagram for the class **Member**, in order to illustrate how to write a class definition. However, we have not explored how we actually *use* a class definition.

The class **Member** can be considered as a *provider* class, in that it provides or offers – as it were – a number of public methods for execution. If we assume that one or more of these methods can be called by any other class – let us call them *user* classes – we can think of the set of methods as the *interface* of the provider class, where the term 'interface' is used, in this context, to describe the methods of the provider class. As we will see later, the code for a user class is likely to include Java statements that, firstly, calls (one of the) object's constructors (passing arguments to it if required by the class diagram) and, secondly, calls methods and passes arguments to them if required by the class diagram. In the case of the latter type of call, the user object needs to know the provider object's interface. In other words, a user object only needs to know the following details about each method of the interface of a provider object:

- the name of the method;
- it parameter list, if any;
- its return type, if any.

A user object does not need to know *how* methods are implemented; user objects only need to know the name, parameters and return type of each method of a provider object's interface. Thus a provider object implements the details of what a method does. In short, a provider object knows *how* a method is implemented, whilst a user object knows *what* its provider object implements.

In order to use – i.e. invoke or call - methods of an object of the provider class **Member**, we usually write a user class that constructs one or more instances of the class **Member** and calls their methods using these named instances. The code for the user class, let us call it TestMember (by convention, the word 'Test' precedes the class name), would include a call to the constructor of **Member** in order to place an instance of the class **Member** in memory so that its methods can be called by referring to the instance by its identifier, known as an *object reference*.

The code for the class **Member** in Section 2.3 includes the definition of its *default constructor*, i.e. its no-arguments, no code constructor. In practice, if we only need to call an object's default constructor, there is no need to include it in the code for the class definition: the Java compiler inserts a default constructor into the class definition for us. Clearly, if we need a constructor that takes arguments and includes Java statements in its code block, this must be provided by the developer: the JRE is only able to insert a default constructor if the developer has not provided any constructors. Although the JRE inserts the default constructor, the example in this chapter includes the default constructor explicitly in order to reinforce the purpose of constructors in Java.

Download free eBooks at bookboon.com

Before we can call the (default) constructor for **Member** in our user object, we need (to declare) a variable of the type **Member** in our user class. A variable of a class type is used as an object reference for an object of that class when it is constructed and stored in memory. When we have declared a variable of the type **Member**, we can initialise it by calling the constructor for the **Member** class.

The result of a call to the constructor of **Member** means that we have a reference to an actual object that we can use to call methods of that class. The following code snippet shows how an instance of **Member** is constructed in our user class and how one of its methods is called.

```
// The next statement declares a variable of the type Member, with an identifier
// called member.  Note, again, that class types start with a capital letter and
// identifiers start with a lower case letter.
Member member;

// In the next statement, the variable is initialised to the object reference to the
// instance created when the default constructor of the class Member is called.  The
// entity Member( ) is the no arguments constructor of the class Member.  Thus, the
// types on both sides of the association symbol ‘ = ‘ are compatible.  N. B.  Java is a
// strongly-typed language.
member = new Member( );
```

The first of the two statements above declares a variable called **member** of the **Member** (class) type; the second statement calls the default constructor of the **Member** class. The use of the keyword ‘new’ instantiates an object of the class **Member** and stores it in memory.

Note that the two Java statements above can be combined into one, as follows:

```
Member member = new Member( );
```

We now have an object reference, called **member**, to an instance of the **Member** class stored in memory. This object reference gives us a way to ‘address’ or find our object in memory; without this reference to our object, we cannot call methods defined in the class definition.

We can now call methods of the class **Member** by using the object reference called **member**. Thus, method calls from a user object look like this.

```
// A call to the setUserName method, with a value of the expected type passed to it
// between brackets.  Note that String values are placed between quotation marks.
member . setUserName( "Dylan" );
```


Download free eBooks at bookboon.com

```
// Similarly, a call to the setPassword method.
member . setPassword( "abc999" );

// And a call to the getPassword method.
member . getPassword(  );
```

The three statements above use the object reference to send a message to the object to call a method of the **Member** class. The first statement can be read from right to left thus: pass a String argument “Dylan” to the method **setUserName** and call this method ‘on’ the instance of the class **Member** called **member**, using the ‘dot’ (‘ **.** ‘) selector to identify the specific object (stored in memory) to use for this method call. According to the class definition of the class called **Member**, the method **setUserName** uses the argument passed to it when it is called to execute the Java statement of the method’s implementation. Thus, the method does what it is designed to do as a result of a call to it.

We have established that we need to know an object’s reference so that we can select it when invoking its methods. Thus, the statement at the top of the next page

```
member . getPassword( );
```



Click on the ad to read more
Download free eBooks at bookboon.com

will compile, whereas the statement

```
memberOne . getPassword( );
```

will generate a compiler error as follows: *cannot find symbol memberOne*

The compiler generates this error because we have not declared a variable with the identifier **memberOne**.

We now have some idea how to invoke methods from a user class, the question remains: *how do we execute our user class?* Do we have to write a class definition for it; how do we actually start our user class? Fortunately, this is not as problematic as it may seem at first sight.

Java statements that make method calls can be included in a special method called **main** that is included in the class definition of the user class of the provider class **Member**. When the **main** method of the user class is executed by the Java Virtual Machine (JVM), the statements within **main** are executed. Typically **main** is used to instantiate objects and call methods on those objects. Most Java applications usually contain several classes, at least one of which must include **main**. The **main** method is the entry point of a Java application; the JVM looks for it in order to start the application. We will use a **main** method to test the logic of some of the methods of the provider class.

A simple version of the code for the user class for the provider class Member follows.

```
// Class declaration of the test class.
public class TestMember {

    // The declaration of main.
    public static void main( String[  ] args ) {

    // We will examine the syntax of the declaration of main in due course.   For
    // the time being, the inclusion of the modifier ‘static’ can be taken to mean
    // that we do not have to instantiate an object of the class TestMember.

    // A constructor call and some method calls are next.

    // Declare a variable of the type Member and initialise it by instantiating an
    // object of the class type Member with a call to its constructor.
    Member member = new Member(  );

    // Invoke one of the ‘set’ methods of Member by using the object reference
    // of the instance of the class Member called member.  Pass an argument to
    // the method using the expected data type.
    member . setUserName( "Dylan" );

    }  // End of definition of main.

}  // End of class definition of TestMember.
```


Download free eBooks at bookboon.com

Close inspection of the code shows that all that **main** does is call **setUserName**, the affect of which will depend on the code in the body of the method. However, if a call to **getUserName** were to be included in the code for **TestMember**, we ought to be able to observe the result of this method call because it returns a value. Note that this call is included in a print statement, whose syntax we will examine in due course. The amended code for **TestMember** follows: note the order of the two method calls.

```
// Code for amended test class.
public class TestMember {

        public static void main( String[  ] args ) {

                Member member = new Member(  ); // As before.
                member . setUserName( "Dylan" ); // As before.
                // Call the 'get' method of Member in a print statement.  The method
                // call returns a value.
                System.out.println( "The member's user name is: " +
                        member . getUserName(  ) );

        }  // End of definition of main.

}  // End of class definition of TestMember.
```

In the amended class definition of **TestMember**, the purpose of the ' + ' operator in the argument passed to the **println** method is to concatenate the literal String with the String that is returned from the call to **getUserName**.

Compilation of **TestMember** and execution of **main** produces the following output:

```
The member's user name is: Dylan
```

Inspection of the code for **TestMember** shows that it does not need to know how the variables and methods of **Member** are implemented; rather, **TestMember** only needs a reference to an object of the **Member** class in order to invoke its methods, which it does so by knowing the interface provided by the object. Thus the user class **TestMember** invokes **setUserName** and **getUserName** in that order merely to illustrate that the provider class **Member** does what its interface is contracted to do. In this respect, the **TestMember** class has been used to test some of the functionality of the **Member** class.

If we were to write the body of main as follows:

```
Member member = new Member( );
System.out.println( "The member's user name is: " +
                member . getUserName(  ) );
member . setUserName( "Dylan" );
```

the result of executing main is: The member's user name is: null

Download free eBooks at bookboon.com

Can you work out why this is the case?

You may have noticed that we haven't instantiated an object of the class **TestMember**. This is because its only member – the **main** method – is *static*, which means – in this context - that we don't have to instantiate an object of the **TestMember** type. We will examine static class members in due course.

2.5 Summary

In summary, we have created a simple Java application that comprises two classes: a provider class called **Member** and a user, test class called **TestMember**. The former has variables, a constructor and a number of methods and the latter constructs an instance of the former and calls two of its methods to access and manipulate the data value of one of the object's variables.

Despite its simplicity, the example discussed in this chapter illustrates the following concepts:

- encapsulation (of data and behaviour);
- coding class definitions for provider and test classes in a Java application;
- object construction;
- invoking methods;
- execution of a **main** method to carry out some simple testing.



Click on the ad to read more
Download free eBooks at bookboon.com

Before we move on to Chapter Three, let us examine a visual representation of an actual **Member** object in memory in order to illustrate, visually, how it is created and used in order to show that the visual representation gives the same result as executing **main** as we did in Section 2.4.

The screen shot below shows the **Member** class in a Java development environment known as BlueJ; the purpose of this illustration is to show what happens when an instance of the class is constructed.

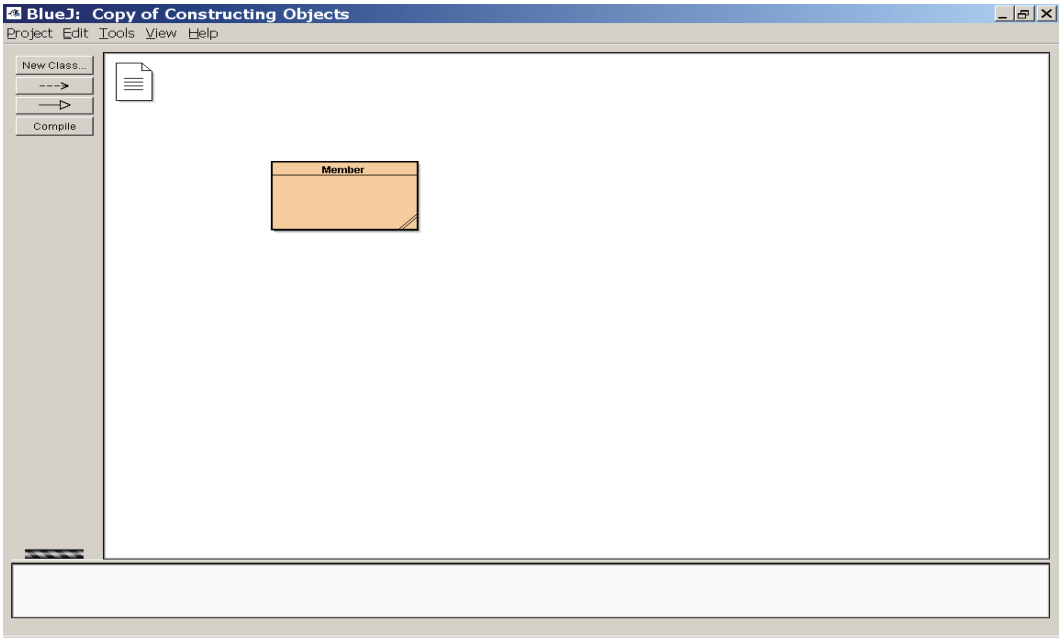


Figure 2.5 The Member class

The single icon in Figure 2.5 (on the previous page) represents the class definition for the **Member** class and implies that it has been loaded.

Download free eBooks at bookboon.com

The next screen shot shows that the constructor (**`Member( )`**) is available for invocation.

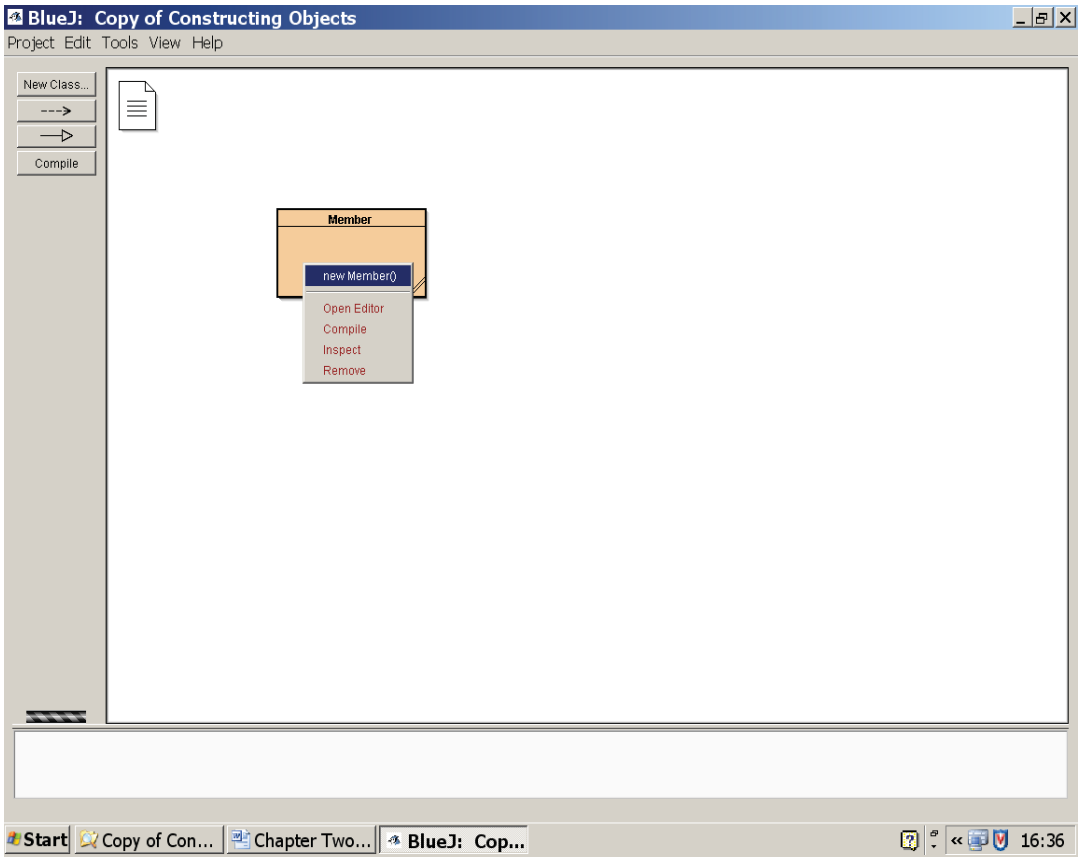


Figure 2.6 Access to the constructor

Download free eBooks at bookboon.com

The constructor is invoked by pressing Ok in the dialogue box shown in the next screen shot.

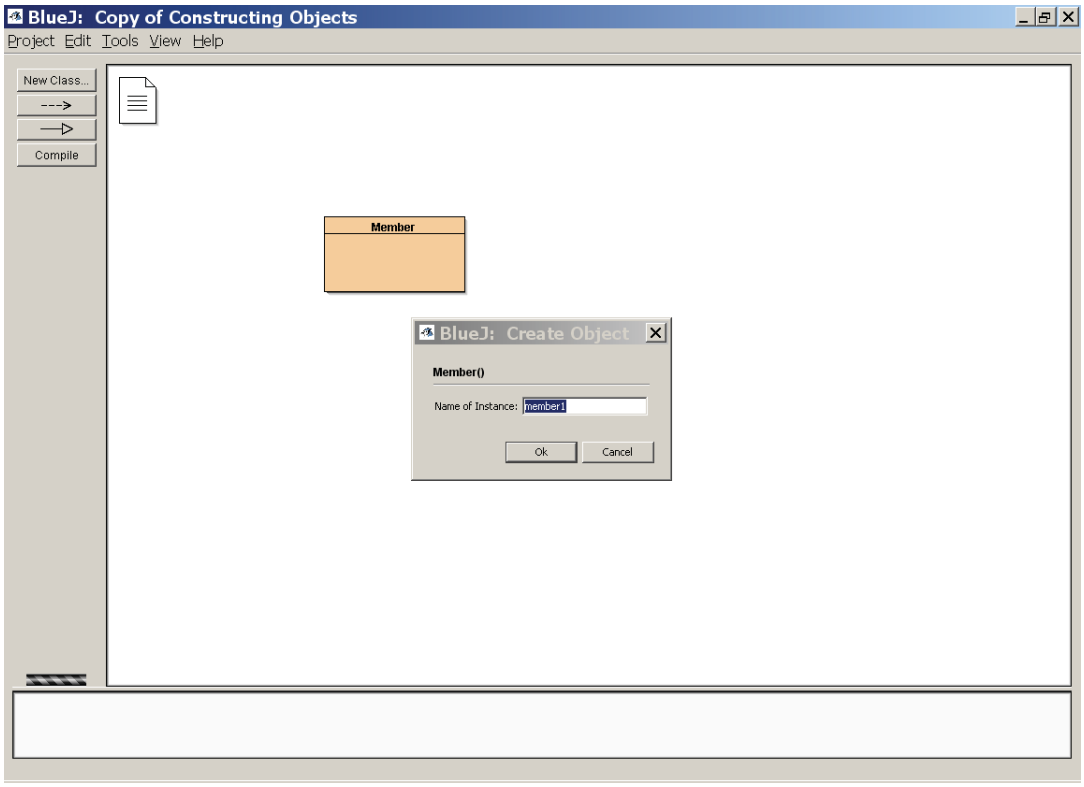


Figure 2.7 A call to Member( )

Figure 2.7 shows that the Java development environment calls the default constructor for **`Member`** and provides a default (editable) object reference to the instance created. In the figure above the object reference is given the identifier **`member1`**.

Following the call to the (default) constructor, the Java development environment represents the actual object stored in memory as a rectangle at the foot of the screen. The next screen shot shows that this object – with the reference **`member1`** – has been instantiated and stored in memory.

Download free eBooks at bookboon.com

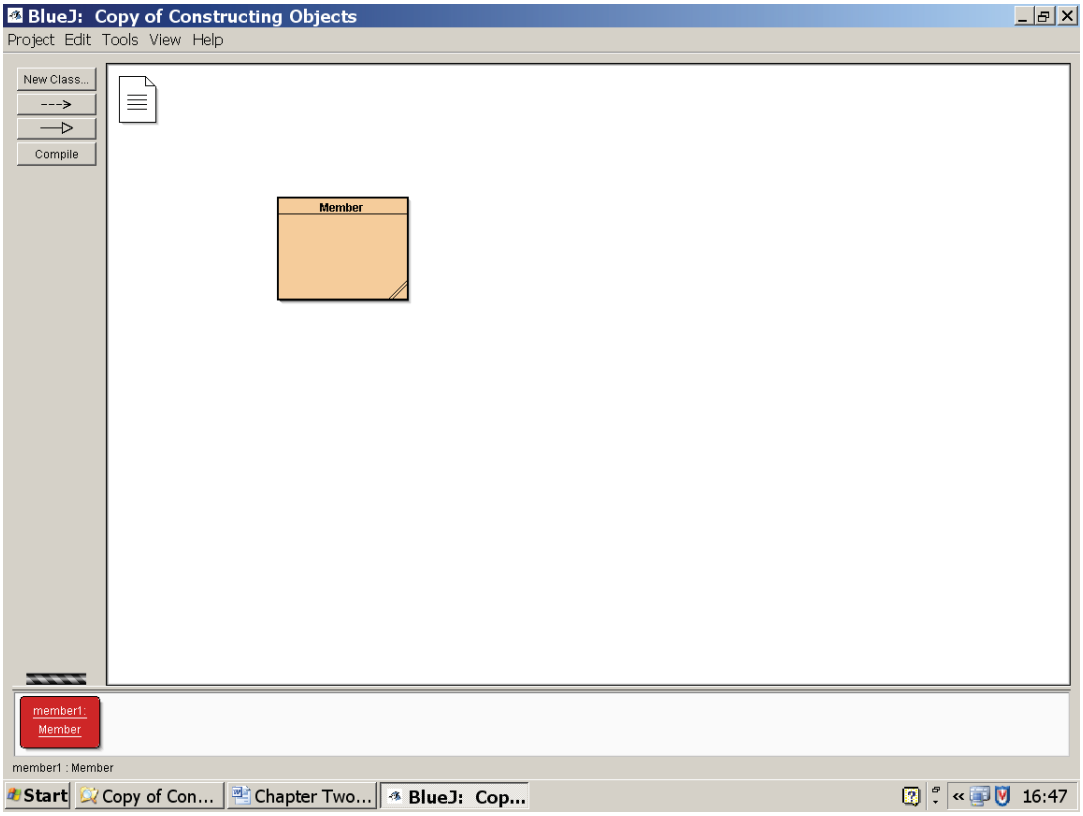


Figure 2.8 The instance with an object reference of member1



Click on the ad to read more

Download free eBooks at bookboon.com

The next screen shot shows the four methods of the object; it can be seen that they comply with the class diagram.

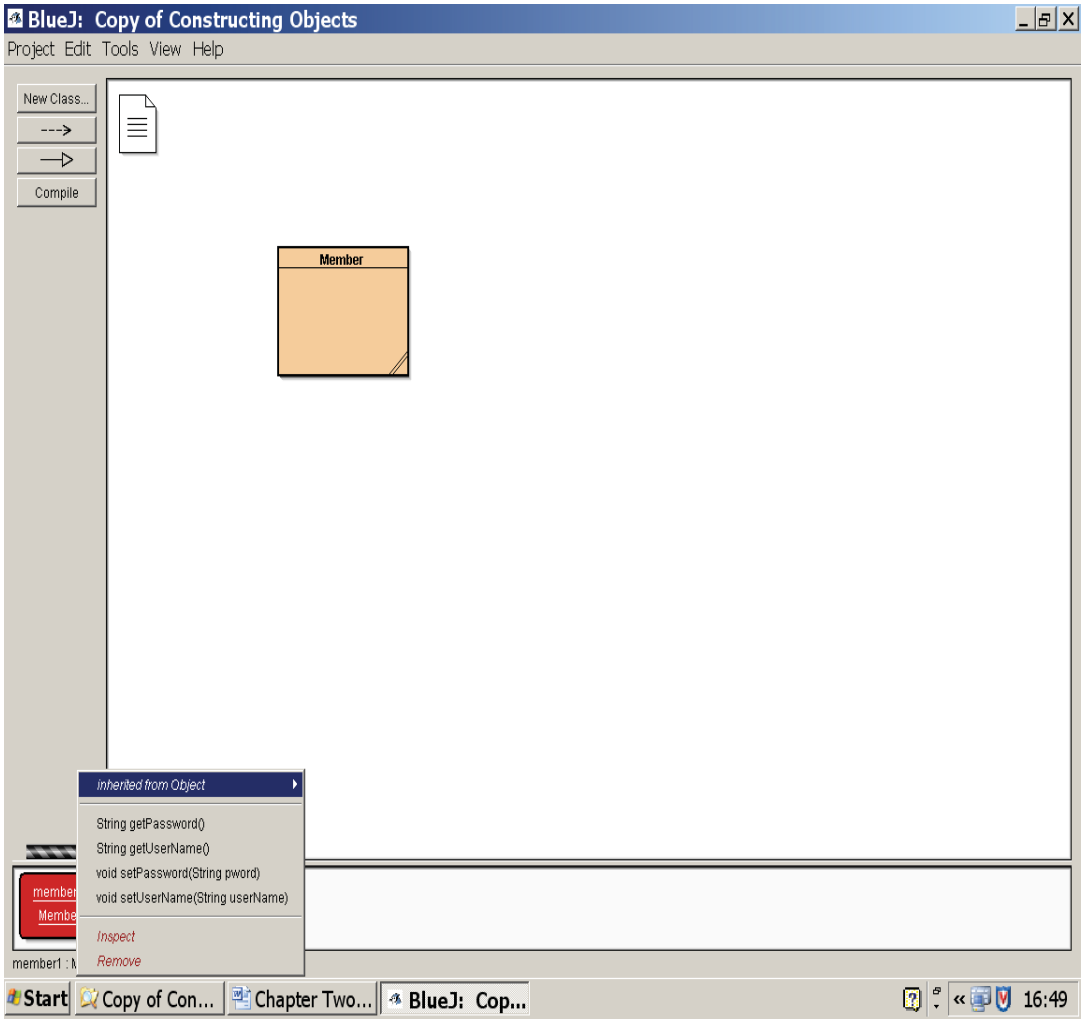


Figure 2.9 The object's interface, in terms of its public methods

The purpose of the series of screen shots above is to show, in a visual way, how a **Member** object is constructed and used, without the need to write a **main** method. Any of the four methods of the object stored in memory can be tested using the Java development environment. For example, the next screen shot shows a call to **setUserName**.

Download free eBooks at bookboon.com

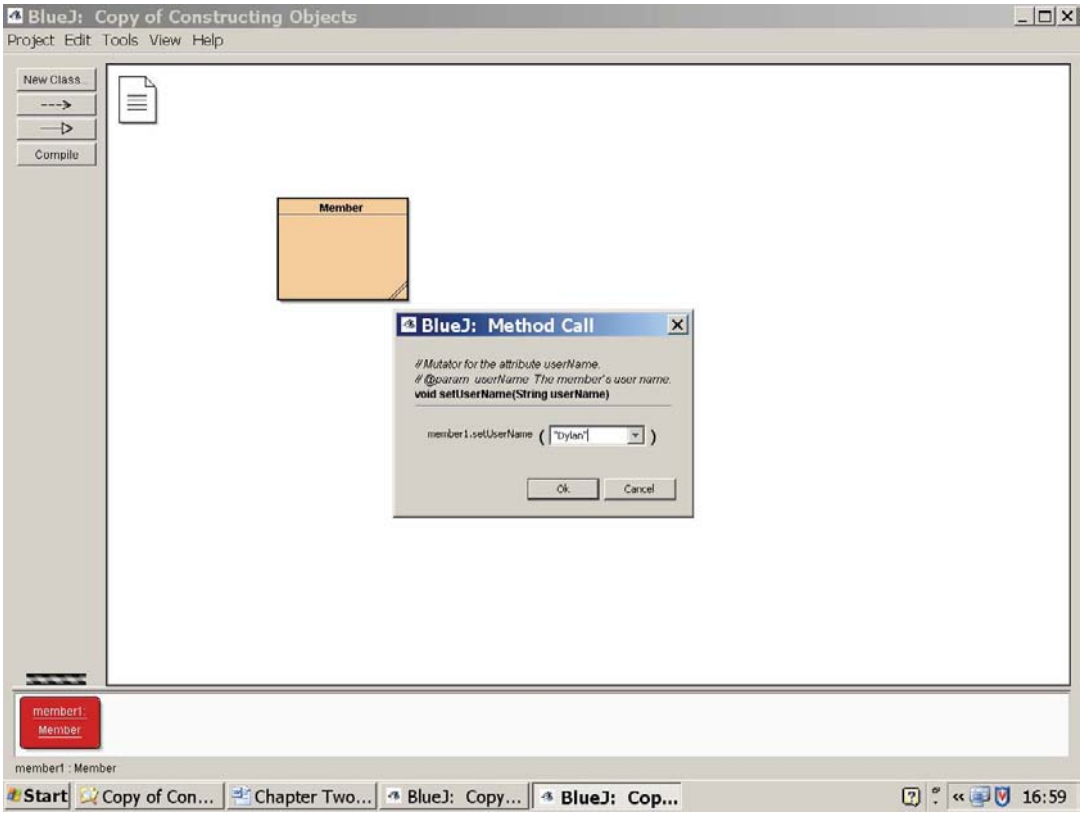


Figure 2.10 A call to setUserName

Figure 2.10 shows that the call to **setUserName** expects a **String** argument, as required by the class definition.

The next screen shot shows what happens when the value of the variable **userName** is inspected.

Download free eBooks at bookboon.com

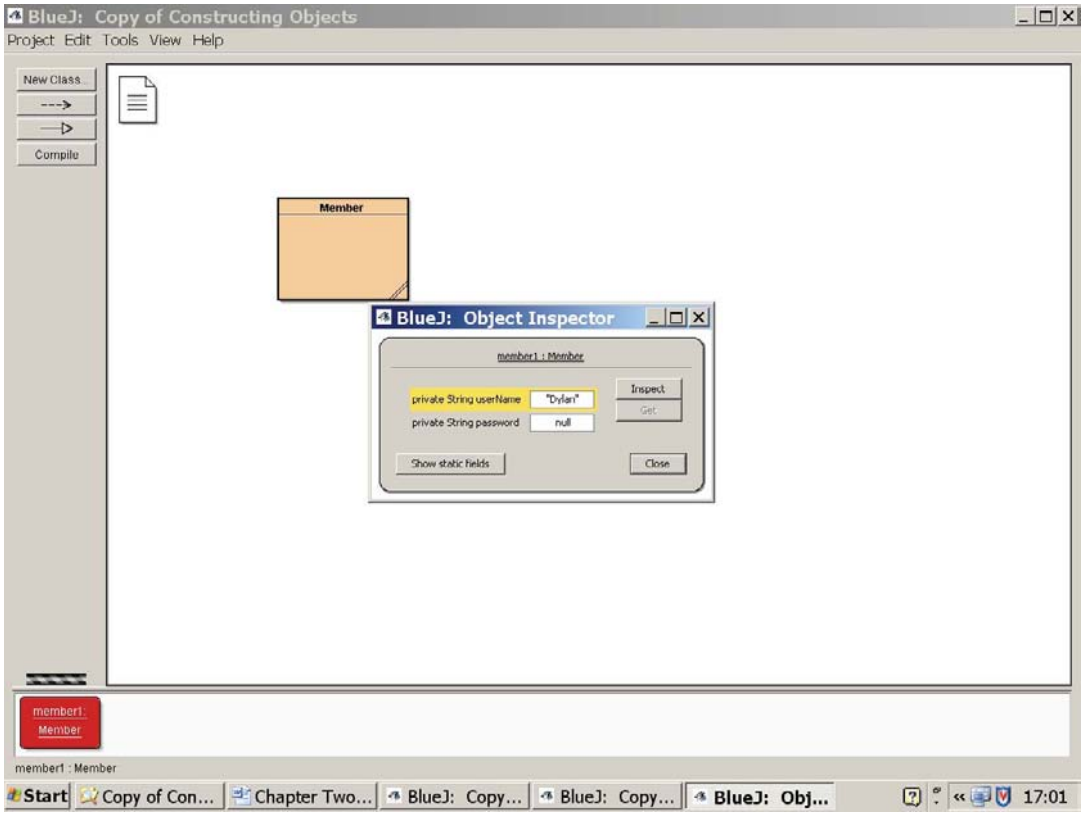


Figure 2.11 The value of the variable userName

Figure 2.11 shows that the value of the variable **userName** is "**Dylan**", as expected. However, we can see that the value of the variable **password** is **null**, because its set method has not been called and the effect of calling the default constructor initialises all instance variables to their default values. Since initialisation, the **setUserName** method has been called. Hence, its value has been changed from **null** to "**Dylan**", whereas the value of the variable password remains as null, because the default value of a Java String object is an object reference of value **null.**

Default object references have the value **null**.

A call to getUserName is shown in the final screen shot.

Download free eBooks at bookboon.com

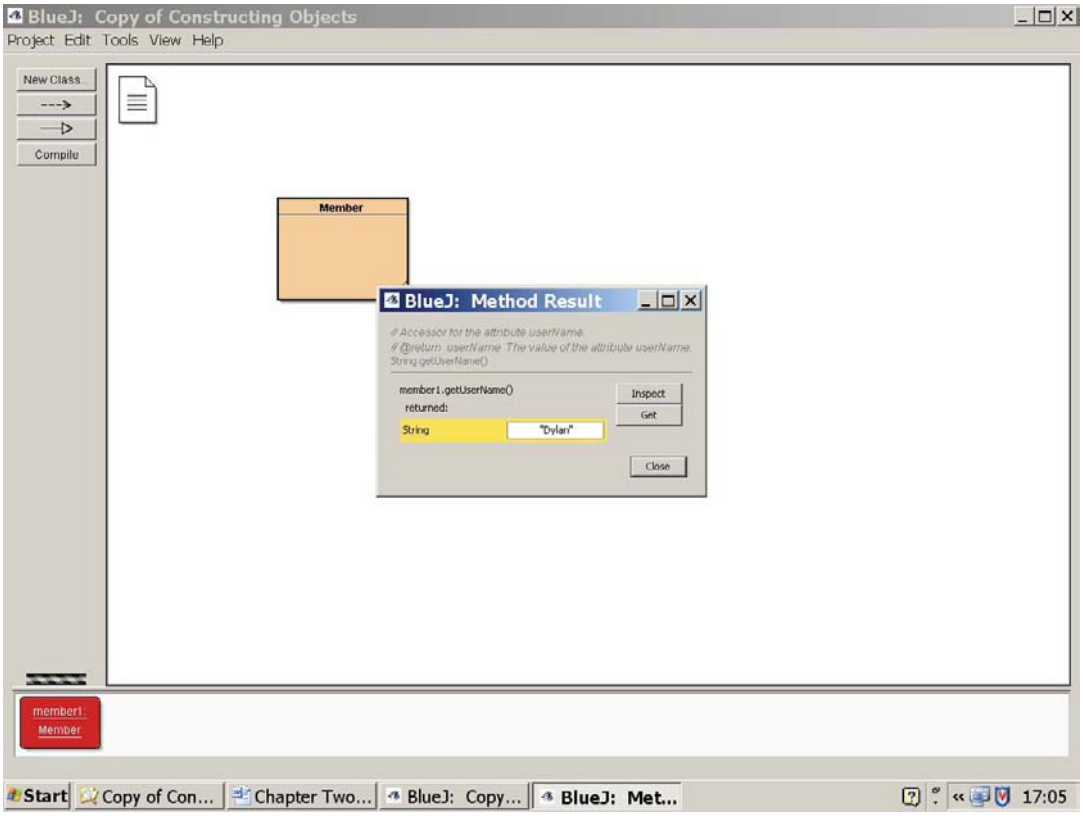


Figure 2.12 A call to getUserName

Invocation of **getUserName** returns the value "**Dylan**", the same result that we observed when the main method in Section 2.4 is executed.

The screen shots show how a constructor of a class has been used to instantiate and store, in memory, an object so that it is available by using its object reference to call its methods. The purpose of the illustration is to replicate the function of the **main** method outlined in Section 2.4, in order to visualise how we can create and use a **Member** object and, at the same time, visualise some of the fundamental concepts of classes and objects that are covered in chapters 1 and 2.

In practice, we usually write one or more test classes to test an object's methods. When we are satisfied with the results of our test strategy, we can reject all but one of our test classes so that we retain a single **main** method that starts the application in the correct place. When we use a graphical user interfaces (GUI) to 'drive' an application, the **main** method usually displays the GUI. We will explore GUIs towards the end of this guide.

Before we add some classes to the themed application, the next chapter explores some of the basic elements of the Java language.

Download free eBooks at bookboon.com

3. Language Basics: Some Syntax and Semantics

3.1 Introduction

In Chapter Two, we see that class attributes are implemented in Java programmes as *variables*, whose values determine the *state* of an object. To some extent Chapter Two addresses the question of how we *name* variables; this question is explored further in this chapter.

Chapter Three explores some of the basic elements of the Java language. Given the nature of this guide, it is not the intention to make this chapter exhaustive with respect to all of the basic elements of the Java language. Further details can be found in the on-line Java tutorial.

> The Java tutorial that covers language basics is found at:
> http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html

We see in Chapter Two that the two broad categories of Java *types* are *primitives* and *classes.* There are eight of the former and a vast number of classes, including several thousand classes provided with the Java language development environment and an infinitude of classes written by the worldwide community of Java developers. This chapter examines aspects of both categories of types.

3.2 Identifiers

An identifier is a meaningful name given to a component in a Java programme. Identifiers are used to name the class itself – where the name of the class starts with an upper case letter – and to name its instances, its methods and their parameters. While class identifiers *always* – by convention – start with an upper case letter, everything else is identified with a word (or compound word) that starts with a lower case letter. Identifiers should be made as meaningful as possible, in the context of the application concerned. Thus compound words or phrases are used in practice.

Referring to elements of the themed application, we can use the following identifiers for variables in the **`Member`** class:

lastName; firstName; MembershipCard

but not

last name; membershipCard

because we wouldn't name a class **`membershipCard`** and spaces are not permitted in identifiers.

We could have declared other variables in the class definition as follows:

last_name; firstName; $password;

Download free eBooks at bookboon.com

We cannot use what are known as *keywords* for identifiers. These words are reserved and cannot be used solely as an identifier, but can be used as part of an identifier. Thus we cannot identify a variable as follows:

private int new; // not permitted because **int** is a keyword

but we could write

private int newInt;

The table below lists the keywords in the Java language.

abstract	Continue	for	New	switch
assert***	Default	goto*	Package	synchronized
boolean	Do	if	Private	this
break	Double	implements	Protected	throw
byte	Else	import	Public	throws
case	enum****	instanceof	Return	transient
catch	Extends	int	Short	try
char	Final	interface	Static	void
class	Finally	long	strictfp**	volatile
const*	float	native	super	while

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

Source: http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html

Table 3.1 The Java language keywords

Java is case-sensitive: this means that we cannot expect the following statement to compile:

return newint;

if we have not previously declared the identifier **newint**. On the other hand, if we write

return newInt;

as the last statement of the **getNewInt** method, it *will* compile because the identifier named **newInt** has been declared previously.

Download free eBooks at bookboon.com

Similarly we cannot expect the compiler to recognise identifiers such as the following

new_int;
new_Int;
New_int;

if they have not been declared before we refer to them later in our code.

3.3 Primitive Data Types

In one of the declarations in Section 3.2, we declared a variable with the identifier **newInt** to be of the **int** type, in the following statement:

private int newInt;

Let us deconstruct this simple statement from right to left: *we declare that we are going to use an identifier named* `newInt` *to refer to integer values and ensure that access to this variable is private.*

This kind of declaration gives rise to an obvious question: *what primitive data types are there in the Java language?* The list on the next page summarises the primitive data types supported in Java.



Click on the ad to read more

Download free eBooks at bookboon.com

Data Type	Language Element	Comments
logical	boolean	values: true or false
text	char	use 16 bit Unicode characters; literal value enclosed between ‘ ‘, as in ‘u’
text	String	not a primitive, a String of characters is a Java class that is loaded with the class loader; literal values enclosed between “ “, as in “hi”
integral	byte	1 byte
	short	2 bytes
	int	4 bytes
	long	8 bytes
real	double	8 bytes
	float	4 bytes

Table 3.2 Primitive data types

3.3.1 Conversion Between Primitive Data Types

Before we move on to discuss assignment of actual values to variables, it will be instructive to find out if Java can convert between types automatically or whether this is left to the developer and if compile-time and run-time rules for conversion between types are different.

In some situations, the JRE implicitly changes the type without the need for the developer to do this. All conversion of primitive data types is checked at compile-time in order to establish whether or not the conversion is permissible.

Consider, for example, the following code snippet:

```
int i = 10;
double d = i; // assign an int type to a double type
```

A value of 10.0 is displayed when `d` is output.

Evidently the implicit conversion from an `int` to a `double` is permissible.

Consider this code snippet:

```
double d = 10;
int i = d;
```


Download free eBooks at bookboon.com

The first statement compiles; this means that the implicit conversion from an **int** to a **double** is permissible when we assign a literal integer value to a **double**. However the second statement does not compile: the compiler tells us that there is a *possible loss of precision*. This is because we are trying to squeeze, as it were, an eight byte value into a four byte value (see Table 3.2); the compiler won't let us carry out such a *narrowing* conversion.

On the other hand, if we write:

```
double d = 10;
int i = ( int )d; // the cast ( int ) forces d to be an int; we will examine the concept of casting
                  // or explicit conversion later in this section
```

Both statements compile and a value of 10 is displayed when **i** is output.

The general rules for implicit assignment conversion are as follows:

- a boolean cannot be converted to any other type;
- a non-boolean type can be converted to another non-boolean type provided that the conversion is a *widening* conversion;
- a non-boolean type cannot be converted to another non-boolean type if the conversion is a *narrowing* conversion.

INNOVATIVE
LIKE YOU.

If you're hoping for a truly modern education, one where you're encouraged to speak your mind and to think long-term, both when it comes to your own future and the future of the planet. Then the University of Gothenburg is the place for you.

Study a Master's programme in Gothenburg, Sweden | www.gu.se/education

UNIVERSITY OF GOTHENBURG

Click on the ad to read more

Download free eBooks at bookboon.com

Another kind of conversion occurs when a value is passed as an argument to a method when the method defines a parameter of some other type.

For example, consider the following method declaration:

```
public void someMethod( double someDoubleValue ) {

    // do something with the argument

} // end of method
```

The method is expecting a value of a **double** to be passed to it when it is invoked. If we pass a **float** to the method when it is invoked, the **float** will be automatically converted to a **double**.

Fortunately the rules that govern this kind of conversion are the same as those for implicit assignment conversion listed above.

3.3.2 Explicit Conversion Between Primitive Date Types

The previous sub-section shows that Java is willing to carry out widening conversions implicitly. On the other hand, a narrowing conversion generates a compiler error. Should we actually *intend* to run the risk of the possible loss of precision when carrying out a narrowing conversion, we must make what is known as *an explicit cast*. Let us recall the following code snippet from the previous sub-section:

```
double d = 10;
int i = ( int )d; // the cast ( int ) forces d to be an int
```

Casting means explicitly telling Java to force a conversion that the compiler would otherwise not carry out implicitly. To make a cast, the desired type is placed between brackets, as in the second statement above, where the type of **d** – a **double** - is said to be *cast* (i.e. flagged by the compiler to be converted at run-time) into an **int** type. Casting is, typically, carried out when a narrowing conversion is required and, as such, is never carried out implicitly; an explicit cast must be provided by the Java developer to inform the compiler that the risk of losing precision is acceptable.

Conversion and casting of object references is beyond the scope of this chapter, but will be addressed in a later chapter.

3.4 Variables

Broadly-speaking, there are five categories of variables associated with classes: instance variables; class variables; constants; local variables; parameters.

Download free eBooks at bookboon.com

3.4.1 Instance Variables

Instance variables are data elements or object references that can change in value. Instance variables are usually declared at the beginning of a class definition. Recalling our **Member** class, if we ignore the modifier **private** for the time being, it declares the following instance variables:

```
String firstName; // i.e. the type, followed by the identifier, followed by a semi-colon
String lastName;
String userName;
String password;
MembershipCard card;
```

The first four types are Java **String** class types and the fourth is one of the class types of the themed application. The four variables declared above are the *instance variables* of the **Member** class. Instance variables are also known as *non-static fields*. Objects store their individual state in instance variables, so called because their values are unique to the object. As the declarations above imply, instance variables must be declared *before* they are used.

We can initialise an instance variable when it is declared, as in the next statement:

```
private String password = "xxxxxxxx";
```



Click on the ad to read more
Download free eBooks at bookboon.com

In the example of the **Member** class, such initialisation is not very useful: it is shown above merely as an example. However there may be circumstances when initialisation at the same time as declaration of an instance variable *is* useful. For example, for business reasons, we might wish to give all new members of the Media Store a default password, which will be overwritten when the user provides his or her own. Thus, the statement

private String password = "mediastore";

declares and initialises the instance variable.

In practice, instance variables are often initialised by means of a constructor call. For example, one of the constructors of the **Member** class might be:

```
public Member( String password ) {

    this . password = password;

} // end of constructor
```

which when invoked as follows

Member member = new Member ("xxxxxxxx");

assigns the value "xxxxxxxx" to the variable **password** of the object whose reference is **member**.

A method call is typically used to change the value of an instance variable in an application. We know that, from its class diagram, the **Member** class includes a method with the identifier **setPassword**, which when invoked as in the following statement

member . setPassword("xxxxxxxx");

provides the **setPassword** method with an argument to work with. If we recall the body of the method, it assigns the argument to the variable **password**, as follows:

password = pword;

The effect of this statement, which is executed when **setPassword** is invoked, assigns the value "xxxxxxxxx" to the variable **password**.

In summary, there are a number of ways to assign a value to an instance variable:

- when it is declared;
- in the body of a constructor, by using an argument (or arguments) passed to the constructor when it is called so that its invocation initialises one or more instance variables;
- following its declaration, later in the code: typically in a method body.

Download free eBooks at bookboon.com

The next sub-section considers *class variables*.

3.4.2 Class Variables

We use instances variable when we want the fields in an object to have unique values, in other words to have object-specific values. The set of values of an object's instance variables is known as its *state*. Thus, instance variables can be inintislised and, subsequently, modified so that the state of an object can be the same or different from other objects of the class.

On the other hand, there are circumstances when we might want the value of a field to be shared, i.e. to be the *same*, for all instances of the class. This category of variable is known as a *class variable*, a variable specific to the class and not to instances of the class. One of the consequences of declaring class variables is that if a Java programme changes the value of a class variable, its value changes for all instances. This will be illustrated by an example in a moment.

Class variables, also known as *static fields*, are so called because the modifier *static* means that there is only one copy of the value of the variable, irrespective of how many instances of the class are in existence: individual objects do not have their own copy. The concept of a static variable begs a question: *why would we want to declare class variables?*



Click on the ad to read more
Download free eBooks at bookboon.com

For example, in the themed application, we could declare a static variable in one of its classes that allocates and keeps track of membership numbers. This class - it has the identifier **MediaStore** - keeps a record of the next available membership number. We don't want to record this value with every **Member** object; instead, we keep a single record of it as a class variable in the **MediaStore** object.

The code that follows aims to illustrate some of the properties of class variables.

```
//  Simplified version of the MediaStore class.
public class MediaStore {

    // Declare a class variable.  The first member will have a membership number
    // of 1000.
    private static int nextAvailableMembershipNumber = 1000;

    // Define a method that registers a new member with the Media Store.
    public void addMember( String firstName, String lastName,
                                    String userName, String password ) {
        // The constructor for Member requires four parameters that are meant to be
        // self-evident.  The fifth parameter is the next available membership number.
        // Hence, the call to getNextAvailableMembershipNumber is next.
        Member newMember = new Member( firstName, lastName,
                userName, password, getNextAvailableMembershipNumber( ) );
        // Increment the class variable.
        nextAvailableMembershipNumber++;

    } // End of addMember.

    public int getNextAvailableMembershipNumber( ) {

        // Get the last member from the file of members.  This object is given the
        // reference lastMember.  The Member class includes a method that
        // returns a member's membership number: getMembershipNumber.
        // Get the membership number of the last member and increment it by 1.
        int lastMembersMembershipNumber =
                lastMember.getMembershipNumber( );
        nextAvailableMembershipNumber =
                        ++lastMembersMembershipNumber;
        // Thus the static member, nextAvailableMembershipNumber, has been
        // set to the incremented value of the most recent member who joined
        // the Media Store.  Return the value of the next available membership number.
        return nextAvailableMembershipNumber;

    } // End of getNextAvailableMembershipNumber.

} // end of class definition
```


Download free eBooks at bookboon.com

When new **Member** objects are created and inspected, it can be shown that their membership numbers are in the sequence 1001, 1002, 1003 and so on.

Before we examine variables that are local to methods, we briefly consider *constant* types in the next sub-section.

3.4.3 Constants

Constants are data items whose value cannot change. They are declared like this:

```
static final discount int = 10; // the standard discount for members of the Media Store is 10 %
                                // if they have a good record of returning borrowed items
```

The term *final* means that the value cannot change.

3.4.4 Local Variables

A variable that is declared in the body of a method is said to be *local* to that method. When the method completes its execution, all local variables are destroyed and are, therefore, said to be *out of scope*. Local variables are only visible to the method in which they are declared; they are not accessible to the other members of the class.

Unlike instance variables, local variables *must* be initialised by the Java developer; otherwise, the compiler will complain. For example, consider a version of the **setPassword** method of the **Member** class:



Click on the ad to read more
Download free eBooks at bookboon.com

```
public void setPassword( String pword ) {

        String defaultPassword; // declare a local String variable
        password = pword;  // as before
        // add the String defaultPassword to the value of password
        password = password + defaultPassword;

        // remaining code

} // end of method definition
```

The method does not compile; the compiler's message is: *variable defaultPassword might not have been initialized*. However, the following version *does* compile:

```
public void setPassword( String pword ) {

        String defaultPassword = null; // declare and initialise a local String variable to its
                                       // default value
        password = pword;  // as before
        // add the String defaultPassword to the value of password
        password = password + defaultPassword;

        // remaining code

} // end of method definition
```

The local variable in the method is initialised to its default value *before* it is used.

> Remember that the default value of a **String**, or *any* object reference, is **null**.

> Local variables must be initialised when they are declared.

3.4.5 Parameters

Parameter variables are listed in the declaration of constructors, methods and **try...catch** blocks. (**try...catch** blocks are examined in Chapter Four in *An Introduction to Java Programming 2: Classes in Java Applications*.)

The declaration of a parameter consists of a type and an identifier. For example, an enhanced version of the themed application includes a constructor for the **Member** class shown on the next page.

Download free eBooks at bookboon.com

```
/**
 * Constructor for objects of class Member.
 *
 * @param fName  The member's first name.
 * @param lName  The member's last name.
 * @param userName  The member's user name.
 * @param password  The member's password.
 *
 */
public Member( String fName, String lName, String uName, String pWord ) {

        firstName = fName;
        lastName = lName;
        userName = uName;
        password = pWord;

} // End of constructor.
```

The constructor declares four parameters, which are replaced at run-time with computable values known as *arguments* when the constructor is called, typically from a **main** method, as follows:

```
Member memberOne = new Member( “David”, “Etheridge”, “Dylan”,  “xxxxxxxx” );
```

The arguments passed to the constructor, when it is invoked, match the parameter list in its declaration. Otherwise the compiler alerts the developer to an error.

One of the methods of the **Member** class is defined as follows:

```
/** This method gives a membership card to a member of the Media Store.
 *
 * @return result  A boolean to state whether the addition of a card is allowed.
 * @param card  A parameter of the MembershipCard type.
 */
public boolean setCard( MembershipCard card ) {

        boolean result = true; // initialise a local variable
        // remainder of method body
        return result;

} // End of setCard.
```

The method would be invoked as follows:

```
// Let us assume that the class definition of the MembershipCard class has been written and
// compiled.  Create a membership card
```


Download free eBooks at bookboon.com

```
MembershipCard card = new MembershipCard( );
// and give it to the Member created above.
memberOne . setCard( card );
```

We could now retrieve the member's membership card so that we could carry out transactions with it, as follows:

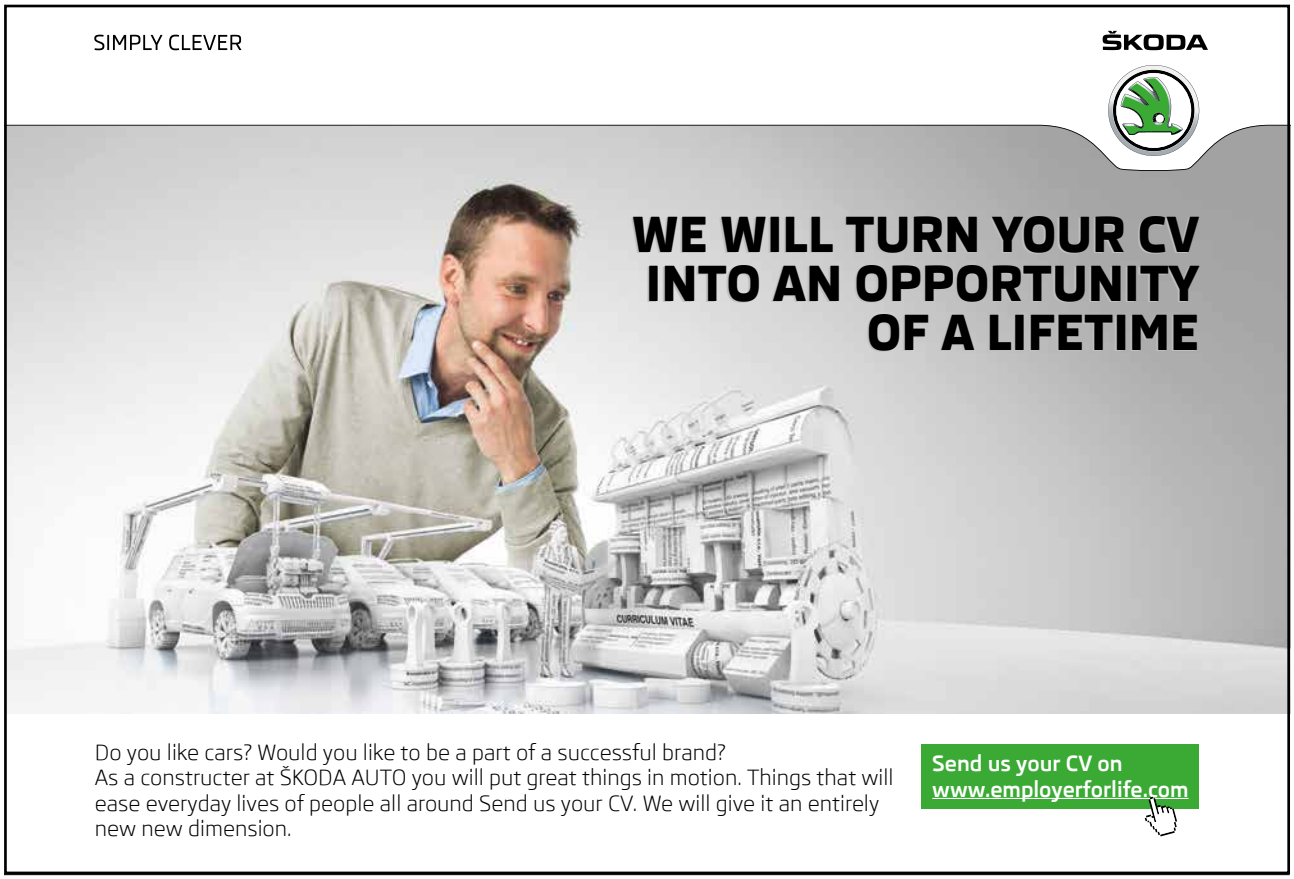
```
memberOne . getCard( );
```

The problem with this method call is that is doesn't actually achieve anything useful as it stands! We know, from the class diagram in Chapter One, that this method returns an object of the type **MembershipCard**, so the statement should – more usefully – read as follows:

```
// Declare a local variable of the MembershipCard type.
MembershipCard card = memberOne . getCard( );
```

so that both 'sides' of the second statement are *compatible* in terms of types. In other words, the class type assignment is compatible with the return type resulting from the call to **getCard**.

> Java is a *strongly-typed* language. This means that the compiler checks for class type compatibility and prevents incompatible assignments at compile time.



Click on the ad to read more
Download free eBooks at bookboon.com

Now that we have access to an actual object of the **MembershipCard** type, it is in fact the card given to the member in the statement

memberOne . setCard(card);

we can, if we wish, call methods of the **MembershipCard** object; for example:

card. getNoOnLoan( ); // this method returns an **int**, so it can be called in a print statement

Parameters cannot be explicitly initialised in the declaration of a method or a constructor because they are implicitly initialised to the arguments passed to the method or constructor when it is called. The parameter variables of a method or a constructor cease to exist, i.e. they go out of scope, when the method or constructor completes its execution. In the case of a method, its completion will *return* a value of type identified in its declaration; in the case of a constructor, its completion returns an object reference to the instance created.

> *Parameters* are the variables declared in a method or a constructor that are replaced at run-time by actual values, known as *arguments*, when the method or constructor is invoked. When the method or constructor is invoked, the arguments *must* match the parameters in type and order, otherwise a compiler error ensues.

3.5 Operators

The following list summarises the operators available in the Java language.

Simple Assignment Operator

=	simple assignment operator

Arithmetic Operators

+	additive operator (also used for String concatenation)
-	subtraction operator
*	multiplication operator
/	division operator
%	remainder operator

Unary Operators

+	Unary plus operator; indicates positive value
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical compliment operator; inverts the value of a boolean

Equality and Relational Operators

==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Download free eBooks at bookboon.com

Conditional Operators

&&	Conditional-AND
||	Conditional-OR

Type Comparison Operator

instanceof	compares an object to a specified type; its use returns a boolean

Source: http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html

Rather than give an example of *every* operator, it is probably more instructive for the learner to encounter operators in context; most of them are likely to be reasonably intuitive.

However it *will* be instructive to give an example of the use of the **instanceof** operator, given that its inclusion in the list implies that it is used to compare class types. We will encounter examples of using this operator in the themed application in later chapters. For the present purposes, we can use this operator in the following simple example, where we wish to find out the class of an object. Consider the following code snippet:

```
// retrieve an object reference for a member of the Media Store; find out its type
if ( memberOne instanceof Member )
{
   // do something with the object whose reference memberOne is of the Member type
}
else
{
   //  do something else
}
```

It can be argued that this example is rather contrived because we *know* that **memberOne** refers to a **Member** object. Nevertheless the aim of the example is to show that if we are working with an object reference and we are not sure of its type, the use of the **instanceof** operator is extremely valuable in that we can find out the type before we embark on subsequent processing using the object that we are not sure about. The code snipped above merely confirms that we are definitely working with a **Member** object by using the **instanceof** operator in an **if...else** construct.

3.6 Summary

The principal focus of this chapter is on categories of variables; Chapter Four forms a pair with it in that it focuses largely on methods.

Download free eBooks at bookboon.com

4. Methods: Invoking an Object's Behavior

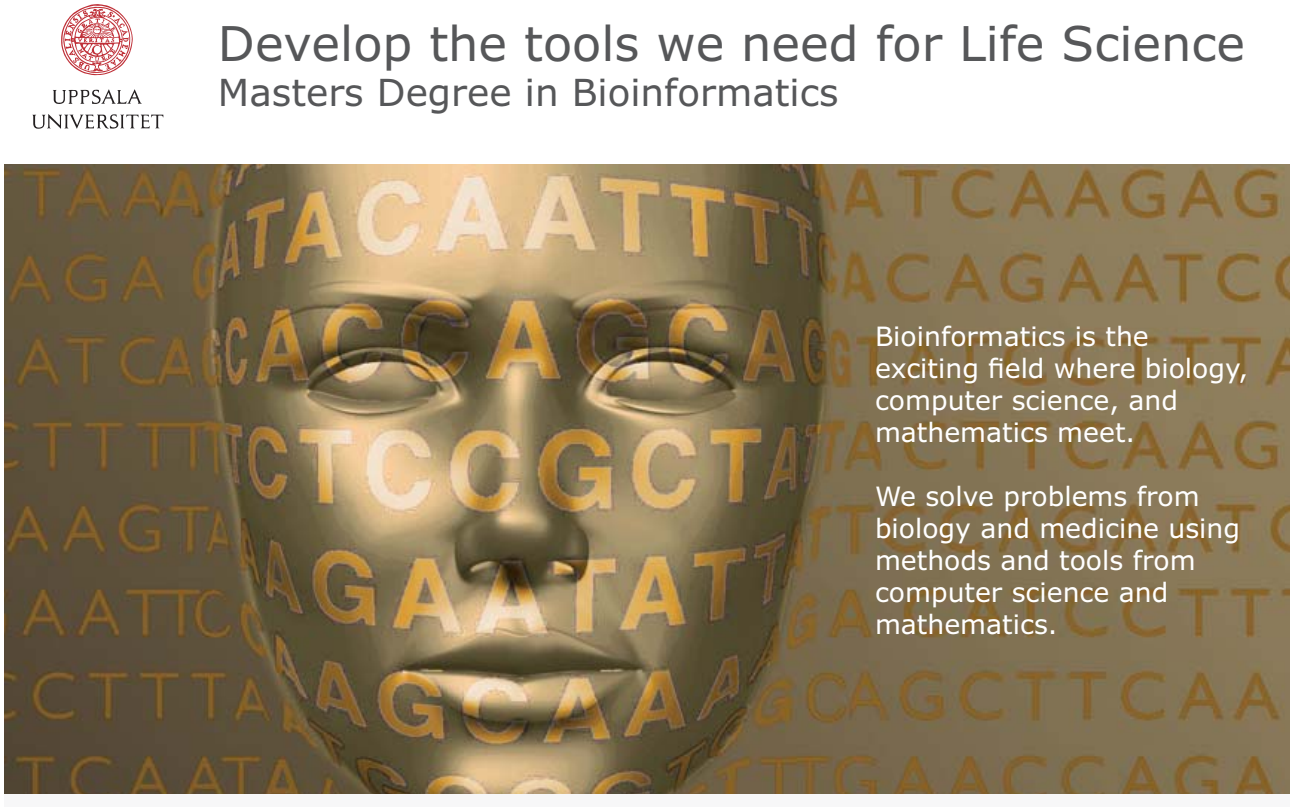
By now the learner will be familiar, to some extent, with *method invocation* from earlier chapters, when objects of the **Member** class in the themed application are used to give some examples of passing arguments to methods. Chapter Four goes into more detail about methods and gives a further explanation about how methods are defined and used. Examples from the themed application are used to illustrate the principal concepts associated with an object's methods.

4.1 How do we get Data Values into a Method?

Chapter Three examines an object's variables, i.e. its *state* or *what it knows what its values are*. An object's methods represent the behaviour of an object, or *what is knows what it can do*, and surround, or encapsulate, an object's variables. This section answers the question about how we get computable values into methods.

As we know from previous chapters, a method is invoked by selecting the object reference for the instance required. The general syntax of a method invocation can be summarised as follows.

```
object_refernece . methodName( param_1, param_2, …, param_X );
```



Click on the ad to read more
Download free eBooks at bookboon.com

Referring, again, to the **Member** class of the themed application, we could instantiate a number of **Member** objects (in a **main** method) and call their methods as in the following code snippet.

```
// Instantiate three members; call the no-arguments constructor for the Member class.
Member memberOne = new Member( );
Member memberTwo = new Member( );
Member memberThree = new Member( );

// Call one of the set methods of these objects.
memberOne . setUserName( "Bob" );
memberTwo . setUserName( "Elvis" );
memberThree . setUserName( "Joni" );

// Call one of the get methods of these objects in a print statement.
System.out.println( The member's username is: + memberOne . getUserName( );
System.out.println( The member's useranme is: + memberTwo . getUserName( );
System.out.println( The member's username is: + memberThree . getUserName( );
```

The screen output from executing this fragment of **main** is:

```
The member's username is: Bob
The member's username is: Elvis
The member's username is: Joni
```

In short, we must ensure that we know which method we are calling on which object and in which order.

In the code snippet above, it is evident that **setUserName** expects a **String** argument to be passed to it; this is because its definition is written as:

```
public void setUserName( Sting username ) {

    // body of method

} // end of method definition
```

The single parameter is replaced by a computable value, i.e. an argument, when the method is invoked.

Download free eBooks at bookboon.com

- The general syntax of a method's declaration is modifier return_type method_name(parameter_list) exception_list

- The method's definition is its declaration, together with the body of the method's implementation between braces, as follows:

```
modifier  return_type  method_name( parameter_list )  exception_list
{
        // body of method
        // return statement (if any) is last
}
```

- The method's *signature* is its name and parameter list.

It is in the body of a method where application logic is executed, using statements such as:

- invocations: calls to other methods;
- assignments: changes to the values of fields or local variables;
- selection: cause a branch;
- repetition: cause a loop;
- detect exceptions, i.e. error conditions.

If the identifier of a parameter is the same as that of an instance variable, the former is said to *hide* the latter. The compiler is able to distinguish between the two identifiers by the use of the keyword 'this', as in the following method definition that we met in Chapter One:

```
public void setUserName( String userName ) [

        this . userName = userName; // 'this' refers to the current object

} // end of setUserName method definition
```

If, on the other hand, we wish to avoid hiding, we could write the method definition as follows:

```
public void setUserName( String uName ) [

        userName = uName;

}
```

where the identifier of the parameter is deliberately chosen to be different from that of the instance variable. In this case, the keyword 'this' *can* be included but it is not necessary to do so.

Download free eBooks at bookboon.com

In both versions of the method **setUserName**, the value of the parameter's argument has scope only within the body of the method. Thus, in general, arguments cease to exist when a method completes its execution.

A final point to make concerning arguments is that a method cannot be passed as an argument to another method or a constructor. Instead, an object reference is passed to the method or constructor so that the object reference is made available to that method or constructor or to other members of the class that invoke that method. For example, consider the following code snippet from the graphical version of the themed application shown on the next page.

UNIVERSITY OF COPENHAGEN



Copenhagen Master of Excellence



Copenhagen Master of Excellence are two-year master degrees taught in English at one of Europe's leading universities

Come to Copenhagen - *and aspire!*

Apply now at
www.come.ku.dk

Click on the ad to read more

Download free eBooks at bookboon.com

```
// The constructor of the GUI takes a parameter of the MediaStore type, passed to it from
// main.
public MediaStoreGui ( MediaStore mediaStore ) {

    // Initalise the instance variable mediaStore of the MediaStoreGui class.
    this.mediaStore = mediaStore;
    // the object reference with the identifier mediaStore is available to members of
    // the MediaStore class.

} // end of definition of constructor
```

The examples and discussion in this section are meant to raise a question in the mind of the learner: *are arguments passed by value or by reference?* This question is addressed in the next sub-section.

4.1.1 Pass by Value versus Pass by Reference

All arguments to methods (and constructors) are, in Java, *passed by value*. This means that a *copy* of the argument is passed in to a method (or a constructor) call.

The example that follows aims to illustrate what pass by value semantics means in practice: detailed code documentation is omitted for the sake of clarity.

```
public class SomeClass {

    // declare an instance variable
    private String greeting;

    public void changeValue( int x ) {
        x = x + 1; // the body of the method changes the value of parameter x
        System.out.println( "x is " + x ); // outputs the new value of x
    }

    public void setGreeting( String newGreeting ) {
        greeting = newGreeting;
    }

    public String getGreeting( ) {
        return greeting;
    }

    public void setReference( String greeting ) {
        // change the value of the argument greeting to refer to a different String
        // object
        greeting = "Hello there!";
        this.greeting = greeting;
```


Download free eBooks at bookboon.com

```
    }

    public String getReference( ) {
        return greeting;
    }

} // end of class definition
```

The test class follows.

```
public class TestSomeClass {

    public static void main( String [ ] args ) {

        // create an instance of SomeClass
        SomeClass sc = new SomeClass( );
        int x = 1234; // initialise a local variable
        sc.changeValue( x ); // pass a copy of x into the method
        System.out.println( "x is " + x ); // output the value of local variable x;
                                           // it should still be 1234
        // create a new String object
        String greeting = new String( "Bonjour" );
        // pass it to setGreeting
        sc.setGreeting( greeting );
        System.out.println( sc.getGreeting( ) );
        // pass it to setReference
        sc.setReference( greeting );
        System.out.println( sc.getReference( ) );
        System.out.println( greeting );

    } // end of main

} // end of class definition
```

The output from executing main is:

```
x is 1235  // the method changeValue changes the value of x
x is 1234  // the original value of x in main is unchanged
Bonjour  // the method calls one of the methods of SomeClass that changes the value
         // of one of its instance variables
Hello there!  // the value of the argument refers to a different object as a result of the
              // method call
Bonjour  // but the original reference declared in main refers to the original object
```


Download free eBooks at bookboon.com

The method **changeValue** changes the value of the argument passed to it – *a copy of* x – but it does not change the original value of **x**, as shown by the output. Thus the integer values 1235 and 1234 are output according to the semantics of pass by value as they apply to arguments.

When a parameter is an object reference, it is *a copy* of the object reference that is passed to the method. You can change which object the argument refers to inside the method, without affecting the original object reference that was passed. However if the body of the method calls methods of the original object – via the copy of its reference - that change the state of the object, the object's state is changed for the duration of its scope in a programme.

Thus, in the example above, the strings "Bonjour" and "Hello there!" are output according to the semantics of pass by value as they apply to object references.

A common misconception about passing object references to methods or constructors is that Java uses *pass by reference* semantics. This is incorrect: pass by reference would mean that *if* used by Java, the *original* reference to the object would be passed to the method or constructor, rather than a copy of the reference, as is the case in Java. The Java language passes object references by value, in that a copy of the object reference is passed to the method or constructor.

> The Java language uses *pass by value* semantics, in that copies of arguments are passed to methods and constructors when arguments are any of the primitive data types or a class type.



Click on the ad to read more

Download free eBooks at bookboon.com

The statement in the box isn't true when objects are passed amongst objects in a distributed application. However, such applications are beyond the scope of this guide. For the purposes of the present guide, the learner should use the examples above to understand the consequences of Java's use of pass by value semantics.

4.2 How do we get Data Values out of a Method?

In previous chapters, we have encountered a number of references to a method's *return type*. In the definition of a method, the return type is declared as part of the method's declaration and its value is returned by the final statement of the method.

For example, consider the following definition for one of the methods of the **DvdMembershipCard** class of the themed application.

```
/**
* This method searches the array of available DVDs for a DVD by its catalogue number. If the
* catalogue number doesn't exist, a suitable message is output.
*
* @param catNo  A String representing the DVD's catalogue number.
* @return dvd  A reference to a Dvd object.
*/
public Dvd findDvd( String catNo ) {

        // Scan the array of DVDs for the one required.
        for ( int i = 0; i < dvds.length; i++ )
        {
                if ( ( dvds[ i ].getCatNo( ) ).equals( catNo) )
                {
                        dvd = dvds[ i ];
                        System.out.println( "Found DVD: " + dvd.getCatNo( ) + " " +
                                dvd.getTitle( ) );
                } // End of if block.
        } // End of for loop.

        if ( dvd != null )
        {
                return dvd; // return statement
        } // End of if block.
        else
        {
                System.out.println( "No such catalogue number." );
                return null; // return statement
        } // End of else block.

} // End of findDvd method definition.
```


Download free eBooks at bookboon.com

It isn't necessary for the reader to understand how the method works in detail. Rather, the purpose of the example is to illustrate that the method's *return type* – the **Dvd** class type – matches the final *return statement*, a reference to an object of the **Dvd** class. While it may appear, at first sight, that there are two return statements, only one of them will execute because they are in an **if...else** construct. Thus, the return statement will either return a non-**null** object reference or a **null** object reference.

A method's return type can be one of three types:

- 'void', if the method does not return a value
- one of Java's primitive types
- a class type

The return statement in the second and third case must match the return type specified in the declaration of the method in that it must return a value of the *same* type as the return type. There is no return statement if the method is 'void' in that it does not return a value.

Some of the most common compiler errors encountered by learners of the Java language are to omit the return statement if there *is* a return type, or include it if the method is 'void', or erroneously provide a mismatch between the value of what is returned in the method's final statement and the type declared in the method's declaration. Rather than give an example of each compilation error, it is left to the learner to experience these errors – and correct them – when defining methods. In other words, dealing with this kind of compiler error is best understood by experiencing it in context.

4.3 Method Overloading

Methods with the same name but with different parameter lists are said to be *overloaded*. The term *overloading*, as it applies to methods, means that the method name has more than one meaning. This concept begs a question: *why would we want to overload methods?* There aren't any overloaded methods in the themed application, so an example from the Java API is used to address this question.

Amongst the methods of the **PrintStream** class are the following variants of the **print** method shown on the next page:

Download free eBooks at bookboon.com

void	**print**(boolean b) Prints a boolean value.
void	**print**(char c) Prints a character.
void	**print**(char[] s) Prints an array of characters.
void	**print**(double d) Prints a double-precision floating-point number.
void	**print**(float f) Prints a floating-point number.
void	**print**(int i) Prints an integer.
void	**print**(long l) Prints a long integer.
void	**print**(Object obj) Prints an object.
void	**print**(String s) Prints a string.

Trust and responsibility

NNE and Pharmaplan have joined forces to create NNE Pharmaplan, the world's leading engineering and consultancy company focused entirely on the pharma and biotech industries.

Inés Aréizaga Esteva (Spain), 25 years old
Education: Chemical Engineer

– You have to be proactive and open-minded as a newcomer and make it clear to your colleagues what you are able to cope. The pharmaceutical field is new to me. But busy as they are, most of my colleagues find the time to teach me, and they also trust me. Even though it was a bit hard at first, I can feel over time that I am beginning to be taken seriously and that my contribution is appreciated.



NNE Pharmaplan is the world's leading engineering and consultancy company focused entirely on the pharma and biotech industries. We employ more than 1500 people worldwide and offer global reach and local knowledge along with our all-encompassing list of services. **nnepharmaplan.com**

nne pharmaplan®

Click on the ad to read more

Download free eBooks at bookboon.com

It would be very inconvenient to invent a separate identifier for each variant of the **print** method; instead, the same name is used and the methods are overloaded by dint of their differing parameter lists. The Java compiler distinguishes amongst overloaded methods on the basis of their parameter lists: the method's return type is not considered when methods are overloaded. Thus the two method declarations that follow are not overloaded because they have the same signature; they will generate a compiler error:

```
public void myPrintMethod( int someValue ) {  }
```

and

```
public int myPrintMethod( int someOtherValue ) {  }
```

The compiler highlights the second method declaration and outputs the message: *myPrintMethod already defined*. In other words, the compiler cannot distinguish between the two methods.

As the list of **print** methods shown above implies, method overloading is typically used when variants of a method accept data passed to it in various forms.

It is worth mentioning, at this point in the chapter, that constructors can also be overloaded. Constructors are examined in the next chapter.

4.4 The Structure of a Typical Class Definition

The class definitions used as examples in this and previous chapters suggest that a typical class definition comprises the following outline structure:

```
public class ClassName {

    // Firstly, declare instance variables.
    // Secondly, define constructors.
    // Thirdly, define methods.
}
```

While this structure may appear, on the face of it, to be simplistic, it is used by the worldwide Java developer community such that the three categories of class members, namely: instance variables; constructors; methods are usually defined in that order by convention.

We can use this simple structure to summarise the main points made in this chapter, i.e. how do we invoke an object's methods. This is illustrated in Figure 4.1 below.

Download free eBooks at bookboon.com

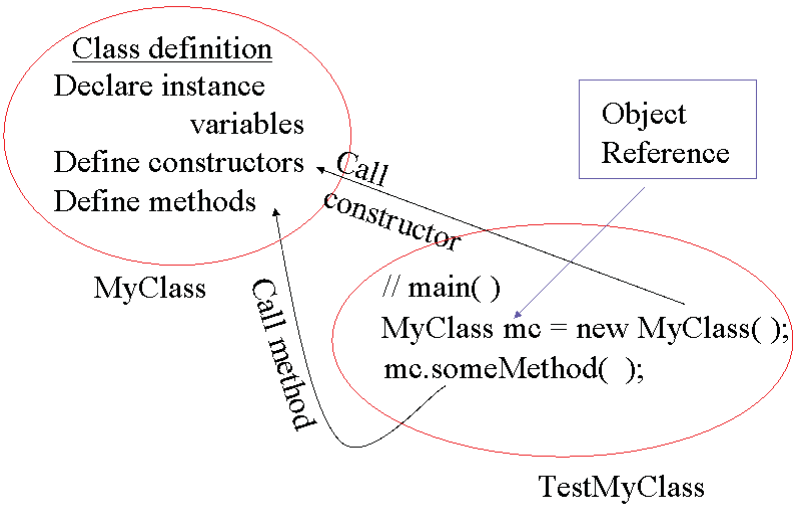


Figure 4.1 Calling an object's constructor and methods

Figure 4.1 shows a code snippet of a simple test class for the class whose definition is called **MyClass**. If we assume that one of the constructors of objects of the type **MyClass** is a no-argument constructor and that the class diagram of **MyClass** includes a no-argument method with the identifier **someMethod**, the two statements of the code snippet for **main that follow**

```
MyClass mc = new MyClass( );  // instantiate an object of the MyClass type
mc . someMethod( );  // call a method by referring to the object reference declared and
                     // initialised in the first statement
```

return us to the introduction to the main theme of this chapter, namely: that we instantiate an object so *that we can invoke its methods.*

The next chapter explains how we create objects by invoking their constructors.

Download free eBooks at bookboon.com

5. Classes and Objects: Creating and Using Objects

5.1 Invoking an Object's Constructor

There are several examples in previous chapters that illustrate how constructors are used to instantiate objects of a class. Let us recall the overall technique before we bring together a number of features of constructors in this chapter.

One of the constructors for **Member** objects in the themed application is as follows:

```
/**
* Constructor for objects of class Member.
*
* @param fName  The member's first name.
* @param lName  The member's last name.
* @param userName  The member's user name.
* @param password  The member's password.
*/
public Member( String fName, String lName, String uName, String pWord ) {
```



Download free eBooks at bookboon.com
Click on the ad to read more

```
// initialise instance variables
            firstName = fName;
            lastName = lName;
            userName = uName;
            password = pWord;
            // instances have access to static members
            // set the membership number by incrementing the next available membership number.
            membershipNumber = ++Member.nextAvailableMembershipNumber;

    } // End of constructor.
```

> An object's constructors have the same name as the class they instantiate.

To access an object of the class **Member** in an application, we first declare a variable of the **Member** type in a **main** method in a test class as follows:

```
Member member;
```

The statement above does not *create* a **Member** object; it merely declares a variable of the required type that can subsequently be initialised to refer to an instance of the **Member** type. The variable that refers to an object is known as its *object reference*. The object that an object reference refers to must be created explicitly, in a statement that instantiates a **Member** object as follows.

```
member = new Member( "David", "Etheridge", "Dylan", "xxxxxxxx" );
```

The two statements above can be combined as follows.

```
Member member = new Member( "David", "Etheridge", "Dylan", "xxxxxxxx" );
```

When the **Member** object is created by using 'new', the type of object required to be constructed is specified and the required arguments are passed to the constructor. The JRE allocates sufficient memory to store the fields of the object and initialises its state. When initialisation is complete, the JRE returns a reference to the new object. Thus, we can regard a constructor as returning an object reference to the object stored in memory.

While objects are explicitly instantiated using 'new', as shown above for a **Member** object, there is no need to explicitly destroy them (as is required in some OO run-time systems). The Java Virtual Machine (JVM) manages memory on behalf of the developer so that memory for objects that is no longer used in an application is automatically reclaimed without the intervention of the developer.

5.2 Object Construction and Initialisation of an Object's State

In general, an object's fields can be initialised when they are declared or they can be declared without being initialised. For example, the code snippet on the next page shows part of the class declaration for a version of the **Member** class:

Download free eBooks at bookboon.com

```
// Declare instance variables.
private String firstName;
private String lastName;
private String userName;
// Initialise the next variable to a literal value.
private String password = “defaltPassword”;
private int membershipNumber;
// Initialise a class variable so that the first membership number is 100.
private static int nextAvailableMembershipNumber = 100;
// Declare an array used to store references to a member's two virtual membership cards.
// (Arrays are explored in the next chapter.)
private MembershipCard [ ] cards = new MembershipCard[ 2 ];
// The next variable keeps track of the next available space in the array of cards.
private int noOfCards;
```

The code snippet illustrates an example where some of the instance variables are initialised and some are only declared. In the case of the latter type of declaration, the instance variable is initialised to its default value when the constructor returns an object reference to the newly-created object. For example, the instance variable **noOfCards** is initialised to 0 when the object is created.

Declaring and initialising none, some or all instance variables in this way if often sufficient to establish the initial state of an object. On the other hand, where more than simple initialisation to literals or default values is required and where other tasks are required to be performed, the body of a constructor can be used to do the work of establishing the initial state of an object. Consider the following part of the constructor for the **Member** class.

```
public Member( String fName, String lName, String uName, String pWord ) {

    // initialise instance variables
    firstName = fName;
    lastName = lName;
    userName = uName;
    password = pWord;

    // remaining code of the constructor.

} // end of definition of constructor
```

This constructor is used when simple initialisation of **Member** objects is insufficient. Thus, in the code block of the constructor above, the arguments passed to the constructor are associated with four of the fields of the **Member** class. The effect of the four statements inside the constructor’s code block is to initialise the four fields before the constructor returns a reference to the object.

Constructors can, like methods, generate or *throw* special objects that represent error conditions. These special objects are instances of Java’s in-built **Exception** class. We will explore how to throw and detect

Download free eBooks at bookboon.com

Exception objects in Chapter Four in *An Introduction to Java Programming 2: Classes in Java Applications*.

It is worthwhile being reminded at this point in the discussion about constructors that the compiler inserts a default constructor if the developer has not defined any constructors for a class.

> The *default constructor* takes no arguments and contains no code. It is provided automatically only if the developer has not provided any constructors in a class definition.

5.3 Overloading Constructors

We saw in the previous chapter that methods can be overloaded. Constructors can be similarly overloaded to provide flexibility in initialising the state of objects of a class. For example, the following class definition includes more than one constructor.

```
public class SetTheTime {

        private int hours;
        private int minutes;
        private int seconds;

        public SetTheTime( ) {
                // this constructor does nothing, so hours, minutes and seconds will be set to
                // their default value of 0
        }

        public SetTheTime( int h, int m ) {
                // initialise two instance variables
                hours = h;
                minutes = m;
        }

        public SetTheTime( int h, int m, int s ) {
                // call one of the other constructors
                this( h, m );
        }

        public void getTheTime( ) {
                System.out.println( "The time is " + hours + ":" + minutes );
        }

} // end of class definition
```


Download free eBooks at bookboon.com

The test class is as follows:

```
public class TestSetTheTime {

    public static void main( String [ ] args ) {

    // call the no arguments constructor
    SetTheTime time1 = new SetTheTime( );

    // this will create an object whose time will be set to 1245
    SetTheTime time2 = new SetTheTime( 12, 45, 30 );

    time1.getTheTime( );
    time2.getTheTime( );

    } // end of main

} // end of class definition
```

The output when **main** is executed is:

```
The time is 0:0      // as a result of calling the no-argument constructor
The time is 12:45    // as a result of calling one constructor from another
```

FOSS

Sharp Minds - Bright Ideas!

Employees at FOSS Analytical A/S are living proof of the company value - First - using new inventions to make dedicated solutions for our customers. With sharp minds and cross functional teamwork, we constantly strive to develop new unique products - Would you like to join our team?

FOSS works diligently with innovation and development as basis for its growth. It is reflected in the fact that more than 200 of the 1200 employees in FOSS work with Research & Development in Scandinavia and USA. Engineers at FOSS work in production, development and marketing, within a wide range of different fields, i.e. Chemistry, Electronics, Mechanics, Software, Optics, Microbiology, Chemometrics.

We offer
A challenging job in an international and innovative company that is leading in its field. You will get the opportunity to work with the most advanced technology together with highly skilled colleagues.

Read more about FOSS at www.foss.dk - or go directly to our student site www.foss.dk/sharpminds where you can learn more about your possibilities of working together with us on projects, your thesis etc.

The Family owned FOSS group is the world leader as supplier of dedicated, high-tech analytical solutions which measure and control the quality and production of agricultural, food, pharmaceutical and chemical products. Main activities are initiated from Denmark, Sweden and USA with headquarters domiciled in Hillerød, DK. The products are marketed globally by 23 sales companies and an extensive net of distributors. In line with the corevalue to be 'First', the company intends to expand its market position.

Dedicated Analytical Solutions
FOSS
Slangerupgade 69
3400 Hillerød
Tel. +45 70103370
www.foss.dk



Click on the ad to read more

Download free eBooks at bookboon.com

The example class – **SetTheTime** – is a simple illustration of a class which provides more than one constructor. The example also shows that a constructor can be called from the body of another constructor by using the 'this' invocation as the *first* executable statement in the constructor. Thus, in the example above, the two argument constructor is called in the first statement of the three argument constructor.

5.4 Initialisation Blocks

Complex initialisation of fields can be achieved by using what is known as an *initialisation block.* An initialisation block is a block of statements, delimited by braces, that appears near the beginning of a class definition outside of any constructor definitions. The position of such a block can be generalised in the following simple template for a typical class definition:

```
public class MyClass {

        // declare instance variables first

        // declare initialisation blocks next
        {
                // code for such a block goes here
        } // end of initialisation block

        // define constructors next

        // define methods next

} // end of class definition
```

An initialisation block is executed as if it were placed at the beginning of every constructor of a class. In other words, it represents a common block of code that *every* constructor executes.

Thus far, in this study guide, we have only been able to work with *single* values of primitive data types and object references. In the next chapter, we will find out how we can associate multiple values of types with a single variable so that we can work with multiple values of primitives or object references in an application.

Download free eBooks at bookboon.com

6. Collecting Data I

Chapter Six explores one of the ways in which we can collect data values or object references into a single data structure known as an *array*. Other data structures are explored in Chapter Two (Collecting Data II) in *An Introduction to Java Programming 3: Graphical User Interfaces.*

6.1 An Introduction to Arrays

Up to this point in the guide, we have encountered many examples of instance variables that refer to a *single* value, using statements such as:

```
private int value1 = 1;
private int value2 = 2;
private int value3 = 3;
```

and

```
private Member memberOne = new Member( );
private Member memberTwo = new Member( );
private Member memberThree = new Member( );
private Member memberFour = new Member( );
```

Whilst this kind of declaration and initialisation is well understood, there are situations (in an application) when we might want to manipulate a logical group or collection of primitive data values or object references as a whole. For example, we might want to store the four **Member** objects instantiated above in a file. In such an event we would have to copy the **Member** objects to the file using four Java statements, one for each object. In order to make it easier to satisfy such a requirement, we need some kind of *data structure* that holds our group of object references in such a way that the group can be referred to by a single identifier.

For example if we could group the four object references for the **Member** objects instantiated above in some way and give the group an identifier as if it were an instance variable, we will have met our general requirement of manipulating multiple values of primitive data types or object references with a *single* variable. This concept is illustrated in Figure 6.1 on the next page.

Download free eBooks at bookboon.com

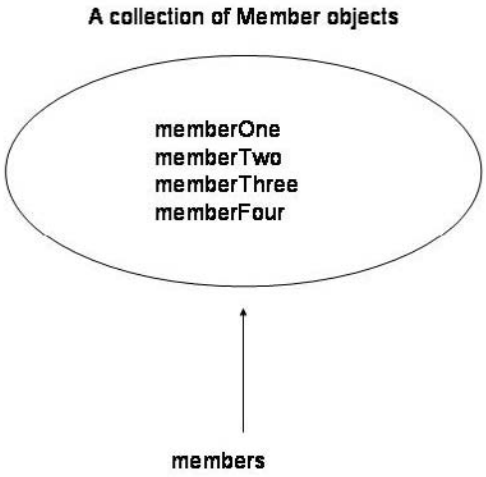


Figure 6.1 A collection of Member objects

In the figure, the variable **members** refers to the group of four references to **Member** objects that have been previously instantiated.

Fortunately in Java, as in most programming languages, a data structure that meets our requirement exists in the form of an *array*.

> An array is a data structure that holds multiple values of the *same* type.

The fact that Java supports arrays raises a question: *how do we work with arrays?* The remainder of this chapter aims to address this question.

6.2 Arrays as Data Structures

The previous section introduces arrays as concrete data structures that can be used to store a homogeneous collection of the values of the same primitive data type or object references to instances of the same class type. We can visualise an array as a collection of containers, into each of which we can place an element of the type that the array is expecting.

Figure 6.2 shows an empty array and Figure 6.3 shows the array populated with elements of the same type.



Figure 6.2 A visual representation of an empty array

Download free eBooks at bookboon.com



An array contains elements that are of the *same* type

Figure 6.3 The array populated with elements of the same type

The array shown in Figure 6.4 is not supported in Java because one of the types is different from the others.



An array cannot contains elements of different types

Figure 6.4 A heterogeneous array

In the sections that follow, we will find out how we can place elements in an array and how, subsequently, we can gain access to them.



Download free eBooks at bookboon.com
Click on the ad to read more

6.3 Declaring Arrays

The general syntax for declaring an array is as follows:

type[ ] identifier;

Thus, we can have the following declarations:

```
private Pineapple[ ] pineapples;
private Banana[ ] bananas;
private int[ ] someValues;
private String[ ] dvdTitles;
private Member[ ] members;
```

Statements such as those above do not create the array, they declare a variable and give it an identifier just as if we are declaring an instance variable; the only difference is the inclusion of the pair of square brackets [ ] to indicate that the variable is an array.

6.4 Creating Arrays

Arrays are created by using '**new**', just as if we are instantiating an object. In fact, an array *is* an object so it is perhaps not surprising that arrays are created by using '**new**'.

The general syntax for creating an array is:

type[ ] identifier = new type[size];

where **size** is the number of elements specified to be reserved (in memory) for the array. This attribute of *any* array provides a public final variable of an array with the identifier **length**. Once the size of an array has been specified, as in the statement in the box above, its length cannot be changed.

To create an array, we write a statement such as the following:

```
char[ ] alphabet = new char[ 26 ]; // 'new' returns a reference (alphabet) to an array of char
                                   // values
String[ ] dvdTitles = new String[ 1000 ];
Pineapple[ ] pineapples = new Pineapple[ 7 ]; // the array in Figure 6.3
Member[ ]  members = new Member[ 500 ]; // an array of references of the Member type.
```

The value placed between [and] *must* be provided by the Java developer.

Download free eBooks at bookboon.com

6.4.1 Initialising Arrays

All elements in an array are initialised to their default value until actual values are placed in array positions. The next section explains how an array is populated.

6.5 Populating Arrays

When an array is populated with values, we can visualise it as shown in figures 6.4 and 6.5 on the next page.



Click on the ad to read more
Download free eBooks at bookboon.com

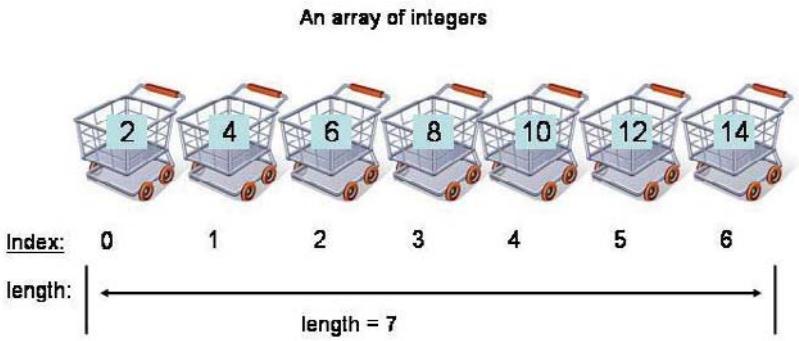


Figure 6.4 An array of integers of length = 7

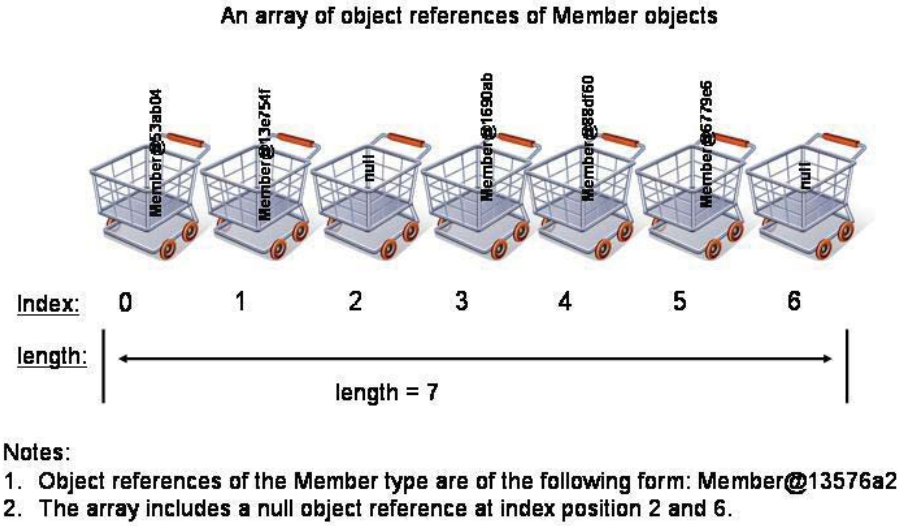


Figure 6.5 An array of references to Member objects

Figure 6.6 redraws Figure 6.5 in such a way that the object references are more clearly seen as the elements of the array.

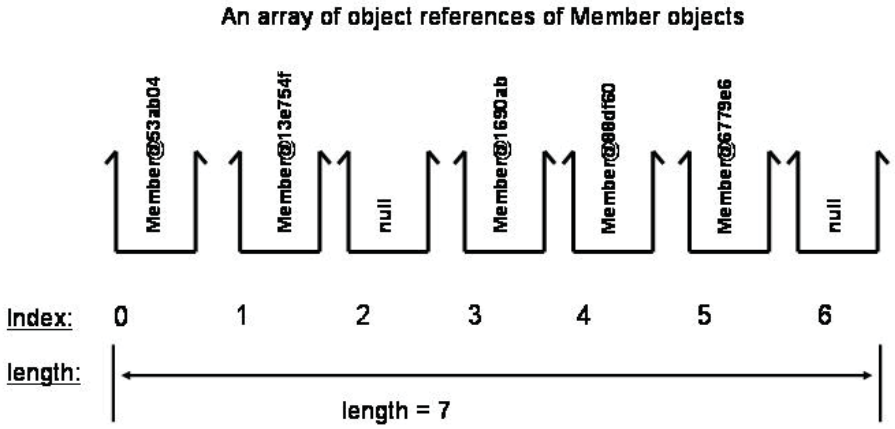


Figure 6.6 null and non-null object references to Member objects in an array

Download free eBooks at bookboon.com

In broad terms, there are three commonly-used ways to populate an array.

Population on Declaration

There is no need to use '**new**' when this technique is used. The size of the array is determined from the number of elements placed between { and }. For example, the statement

Member[ ] members = { memberOne, memberTwo, memberThree };

creates an array of length = 3 and populates it with object references. In such a statement, if any of the object references placed between { and } are **null**, a **null** object reference is placed in the array as shown in Figure 6.6 above.

Using a Loop

A loop can often be used to populate an array. A later chapter (Chapter Two in *An Introduction to Java Programming 2: Classes in Java Applications*) includes an explanation of using loops in Java, its use in the code snippets that follow is meant to be intuitive.

Consider the following example.

```
int[ ] counter = new int[ 10 ];
for( int i = 0; i < counter.length; i ++ )
{
        counter[ i ] = i; // the length of the array is 10
}
```

The effect of the three statements between the (and) of the **for** loop executes the loop ten times, such that the first value of **i** used within the { and } of the **for** loop is 0 and the final value is 9. When the code is included in **main**, the values of the array elements output in a print statement are as follows:

0 1 2 3 4 5 6 7 8 9

Before we move on to look another simple example, it will be instructive to deliberately introduce a logic error in the **for** loop above. Consider the modified code snippet.

```
int[ ] counter = new int[ 10 ];
for( int i = 0; i <= counter.length; i ++ )
{
        counter[ i ] = i; // the length of the array is still 10
}
```


Download free eBooks at bookboon.com

In the modified code snippet, the effect of the statements between the (and) of the **for** loop executes the loop *eleven* times, because the first value of **i** (used within the { and } of the **for** loop) is 0 and the final value is 10. When the code is included in **main**, the output that follows indicates that there is an error when attempting to populate the array position whose index is 10.

0 1 2 3 4 5 6 7 8 9 java.lang.ArrayIndexOutOfBoundsException: 10

The length of the array is unchanged – it is still 10 – but the loop tries to place an integer with value 11 into the 11th position in the array; this position does not exist for this array and an error ensues. The error occurs at *run-time* and is one of the most common errors experienced by learners when using arrays for the first time.

When using a **for** loop to populate an array, the logic of the loop must be carefully checked in order to avoid an **ArrayIndexOutOfBoundsException** type of error.

A **for** loop could be used to populate an array with object references, as shown in the next code snippet.

```
Member[ ] members = new Member[ 10 ];
for( int i = 0; i < members.length; i ++ )
{
        members[ i ] = new Member( ); // populate each array position with a reference to a
                                      // new Member object
}
```

The statement

i < members.length

between the pair of brackets ( ) of the **for** loop ensures that the final value of **i** does not exceed the value of the last index position in the array – i.e. it does not exceed 9 in this case.

The statement

members[i] = new Member( );

in the code block of the **for** loop could have been written as follows:

```
Member member = new Member( ); // i.e. the object reference is explicit
Members[ i ] = member;
```

The outcome is the same in both cases: *we have placed a new Member object in the ith position of the array*. Given that this is done inside a **for** loop, we can re-use the identifier **member** each time the loop is executed. Whichever way we look at it, we do not need to know the identifier of the object references in the array. It is only when we *retrieve* an object reference from an array that we need to give it a reference of the correct type so that we can invoke its methods. We will examine how we access array elements later in this section.

Download free eBooks at bookboon.com

Populate an Array When Required

Individual array positions can be populated as and when required, as exemplified in the statements on the next page.

```
Member[ ]  members = new Member[ 10 ];
members[ 0 ] = new Member( );
members[ 1 ] = new Member( );
```

A shortened version of the **MediaStore** class of the themed application adds a new member to the Media Store by calling one of the constructors of the Member class in a method, as shown in the code below.

```
public class MediaStore {

    // Declare an array to store references to Member objects.
    private Member [ ] members;
    // The number of members who have joined the Media Store.
    private int noOfMembers;

    // Initialise the size of the array of members for testing purposes.
    members = new Member[ 10 ];
```



Click on the ad to read more

Download free eBooks at bookboon.com

```
    /**
    * This method adds a member to the array.
    *
    * @param fName  The member's first name.
    * @param lName  The member's last name.
    * @param uName  The member's user name.
    * @param pWord  The member's password.
    */
    public void addMember( String fName, String lName, String
                                                    uName, String pWord ) {

        members[ noOfMembers ] = new Member( fName,
                                                    lName, uName, pWord );
        noOfMembers++;

    } // end of addMember

    // remainder of the class definition

} // end of class definition
```

The instance variable **noOfMembers**, whose initial value is 0, is used to keep track of the next available index value of the next empty position in the array. The effect of invoking **addMember** ten times places the reference to the newly-created **Member** object in the next position in the array until it is full.

The use of the three techniques described above raises a question: *how do we access array elements?* This will be addressed in a moment.

6.5.1 The Bounds of Array Indices

The examples and figures presented in previous sections aim to show that an array is *bounded* by the index value 0 and (length – 1). As we have seen already, using an out of bounds index gives rise to a run-time error in the guise of an **ArrayIndexOutOfBoundsException** object. (Chapter Four in *An Introduction to Java Programming 2: Classes in Java Applications* explains how **Exception** objects are dealt with.)

6.6 Accessing Array Elements

At the end of Section 6.2, a question is raised: *how do we access array elements?* We have seen that placing a value in an array is straightforward. However retrieving values (from an array) is less than straightforward because we either need to know the index value of the position in the array of the element we are seeking or, if we don't know this value, we have to scan the array until we find what we are looking for. (There are other data structures available in Java that makes it easier to find elements without the need to know *where* in the data structure the element sought has been placed. Chapter Two in *An Introduction to Java Programming 3: Graphical User Interfaces* explores some of these classes.)

Download free eBooks at bookboon.com

Section 6.5 includes these statements

```
counter[ i ] = i; // place the value of i into the (i + 1)th position in the array
members[ i ] = new Member( );  // place the reference to the new Member object into the
                              // (i + 1)th position in the array
members[ 0 ] = new Member( ); // place the reference to a new Member object in the first
                              // position in the array
members[ 1 ] = new Member( ); // place the reference to a new Member object in the second
                              // position in the array
members[ noOfMembers ] = new Member( ); // place the reference to a new Member object
                                        // in the position in the array whose index is
                                        // given by the current value of the variable
                                        // noOfMembers
```

It is evident from their use (in Section 6.5) that an array element is populated by placing the index value between [and] and associating the array position with a value. Accessing an array element is achieved in a similar way. For example, the statement

```
members[ noOfMembers ];
```

accesses the object reference of the **Member** object whose index is given by the current value of **noOfMembers**. If the value of **noOfMembers** is equal to 5, the statement accesses the sixth element in the array.

Similarly, the statement

```
someArray[ 2 ];
```

accessed the third element in the array whose identifier is **someArray**.

A common logic error often made by learners is to write statements such as

```
members[ noOfMembers ];
```

and

```
someArray[ 2 ]; // an array of integers
```

and wonder why nothing happens. In order to access *and* retrieve an element from an array, we declare a variable of the compatible type and associate the array element with it, as follows for the two statements above:

```
Member member = members[ noOfMembers ];
```


Download free eBooks at bookboon.com

and

```
int someValue = someArray[ 2 ];
```

In the case of the first statement, we now have a variable that refers to the array element so that we can work with this **Member** object and, for example, invoke its methods.

While it may seem that accessing an array element is obvious in that it requires only one line of code, it should be remembered that the line of code above only make sense when we know where in the array to find the element we are looking for. In some cases, we *will* know the value of the index position to access. More often that not, though, we do not know this value and we have to search the array for the element we are seeking. A simplified code snippet from one of the methods of the themed application provides an example when such a search is required.

```
// One of the buttons of the GUI is the login button for existing members.
// The purpose of this code block is to find the member in the array of members.
// Capture the existing member's user name and password from the GUI and store these data in
// the variables named username and password.  Concatenate the user name and password.
String searchString = userName + password;

// Search the array of members for the combined user name and password.
// First, get the array of members.
Member[ ] existingMembers = mediaStore.getMembers( );
// Search the array of existing members and compare the search string with each member's
// combined user name and password.
for ( int i = 0; i < mediaStore.getNoOfMembers( ); i ++ )
{
        existingMember = existingMembers[ i ];
        String existingUserName = existingMember.getUserName( );
        String existingPassword = existingMember.getPassword( );
        String combinedNameAndPassword = existingUserName + existingPassword;
        if ( searchString.equals( combinedNameAndPassword ) )
        {
                // Found existing member in the array of members.
                // Exit the for loop.
        } // end of if block
        else
        {
                // do something else
        } // end of else block

  }  // end of the for loop
```


Download free eBooks at bookboon.com

The statement

if (searchString.equals(combinedNameAndPassword))

uses the **equals** method of Java's **String** class. The relevant page of the Java API is summarised below.

java.lang
Class String

java.lang.Object
└**java.lang.String**

equals

public boolean **equals**(Object anObject)

Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object.

Overrides:
`equals` in class `Object`

Parameters:
`anObject` - The object to compare this `String` against

Returns:
`true` if the given object represents a `String` equivalent to this string, `false` otherwise

The use of the **equals** method in the **for** loop enables the code to compare the two strings for each iteration of the loop until an exact match is found.

We will explore the documentation that Java developers provide for their classes and the documentation that is provided by Sun Microsystems Inc. – as exemplified by the extract from the API for the **String** class shown above – in the next chapter.

6.7 Arguments Passed to the main Method

Before we move on to the next chapter, it is worthwhile reflecting that we now know enough about arrays to be able to analyse fully the signature of **main** at this juncture. We have encountered the declaration of **main** a number of times in previous chapters: it is

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

The declaration of **main** can be deconstructed as shown on the next page.

Download free eBooks at bookboon.com

public	**main** has the modifier **public** so that the JVM can access it and use it as the starting point of an application
static	defines **main** to be a class member
void	**main** does not return values of types
String[ ]	the single parameter is an array of **String** objects

The example code snippets that include **main** in previous chapters do not actually *use* the parameter **args**. This raises a question: *what is args used for?*

The parameter **args** can be used to take an argument that is an array of **String** objects. For example, if we were to execute a Java programme by using a batch file that is run by double-clicking a screen icon, we are – in effect – executing a DOS command in the batch file such as the following:

C:\ > java MyClass

where the **java** part of the command executes a programme called **java.exe**.



Click on the ad to read more
Download free eBooks at bookboon.com

java.exe will run **main**, if there is a **main** method in the class **MyClass**. Thus, we can call **main** from the DOS prompt.

On the other hand, we can include arguments in the DOS command and pass values directly into **main**, as exemplified by the next command:

C:\ > java MyClass 192.168.1.2

The result of this command is to place the IP address into **args[0]** so that somewhere in the body of **main** we can write:

String ipAddress = args[0];

to retrieve the **String** value stored in **args[0]**.

The example shows how we can use the parameter defined for the **main** method to pass **String** arguments *directly* into **main**. This is often useful when we need to pass values of, typically, fixed data into **main** so that these data are made available to classes in an application. The example above shows how we can pass the IP address of, for example, a server's location into a client component of a client/server application. This technique is used for some categories of *distributed* applications written in Java in cases where the IP address of the server is fixed and can be hard-coded into a DOS command as shown above. If the IP address of the server changes, the disadvantage of using the parameter **args** is that the batch file would have to be re-written to take account of the new IP address.

Download free eBooks at bookboon.com