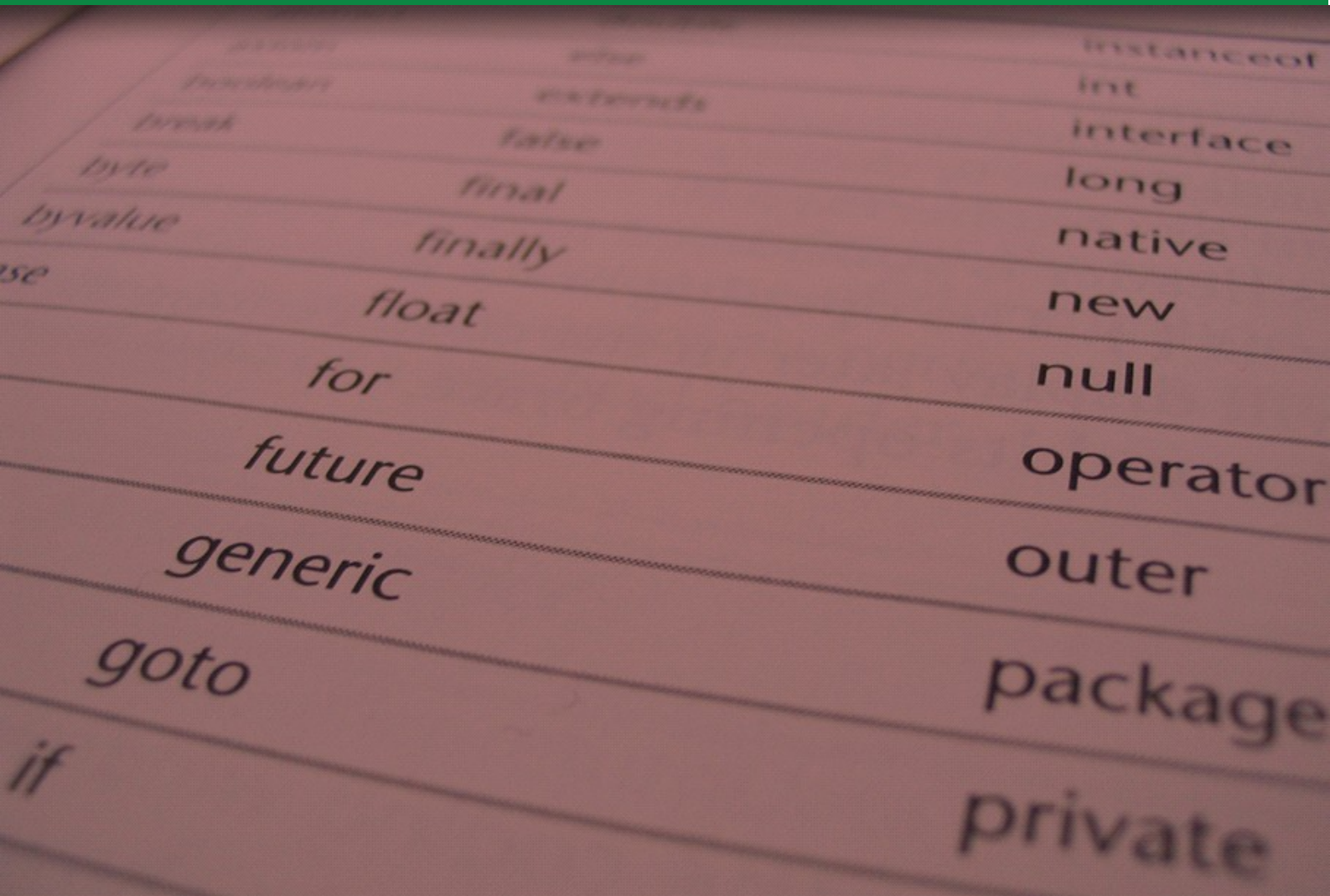


JAVA: THE FUNDAMENTALS OF OBJECTS AND CLASSES

DAVID ETHERIDGE



in association with

TEXTBOOK TORRENTS

David Etheridge

Java: The Fundamentals of Objects and Classes

– An Introduction to Java Programming

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

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

Please click the advert


WHAT'S MISSING IN THIS EQUATION?

You could be one of our future talents

MAERSK INTERNATIONAL TECHNOLOGY & SCIENCE PROGRAMME

Are you about to graduate as an **engineer** or **geoscientist**? Or have you already graduated?
If so, there may be an exciting future for you with A.P. Moller - Maersk.

www.maersk.com/mitas



MAERSK

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

Please click the advert



CHALLENGE YOURSELF

WWW.STUDYINSWEDEN.SE

Today's job market values ambitious, innovative, perceptive team players. Swedish universities foster these qualities through a forward-thinking culture where you're close to the latest ideas and global trends.

Whatever your career goals may be, studying in Sweden will give you valuable skills and a competitive advantage for your future. www.studyinsweden.se

Si.
Swedish Institute

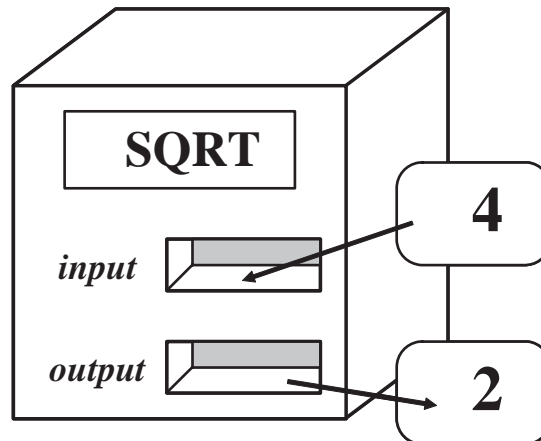
1. Object-Oriented Programming: *What is an Object?*

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.

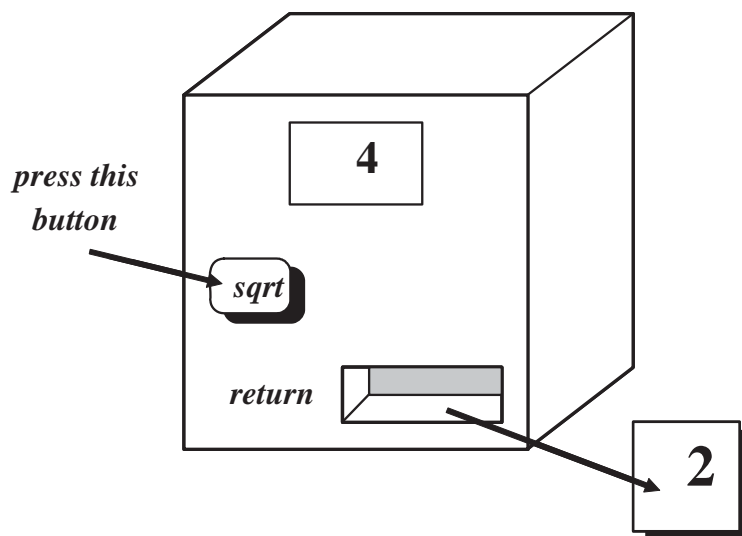


Source: R. A. Clarke, BCU.

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.

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

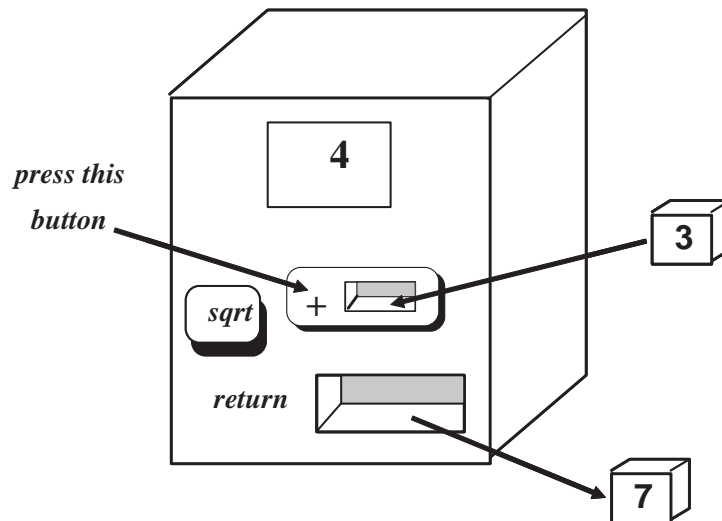
In the figure, the data item – the number 4 – is represented by the box (with the label ‘4’ on its front face). This representation of the number 4 can be referred to as *the object of the number 4*. This simple object doesn’t merely *represent* the number 4, it includes a button labeled **sqrt** which, 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.



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*.

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 data 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.

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:



www.job.oticon.dk

oticon
PEOPLE FIRST

- 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 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 behaviour (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.

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 *class* 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.

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.

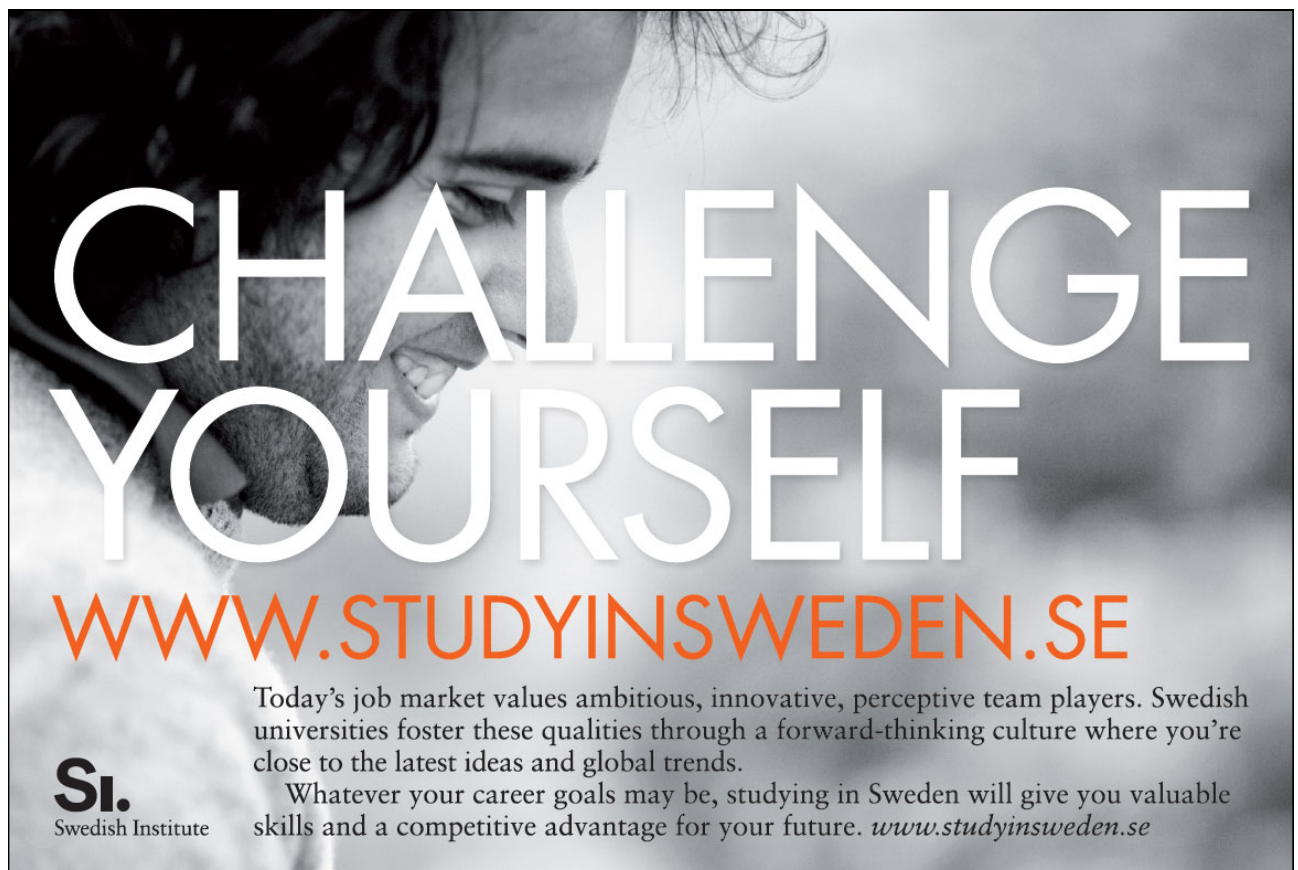
Returning to the class called **Member**, our task is to define (some of) the attributes and behaviour of the class, so that when objects (or *instances*) of the class **Member** are created (or *instantiated*) we can give values to these attributes by making use of the behaviour of the class.

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.

Please click the advert



**CHALLENGE
YOURSELF**

WWW.STUDYINSWEDEN.SE

Si.
Swedish Institute

Today's job market values ambitious, innovative, perceptive team players. Swedish universities foster these qualities through a forward-thinking culture where you're close to the latest ideas and global trends.

Whatever your career goals may be, studying in Sweden will give you valuable skills and a competitive advantage for your future. www.studyinsweden.se

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 <i>class</i> , 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.

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(pwd: string)** – note that there is no return type – where **setPassword** is the name of the behaviour, and **pwd** 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.

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.

Please click the advert



Always aiming for higher ground.
Just another day at the office for a Tiger.

Join the Accenture High Performance Business Forum

On Thursday, April 23rd, Accenture invites top students to the High Performance Business Forum where you can learn how leading Danish companies are using the current economic downturn to gain competitive advantages. You will meet two of Accenture's global senior executives as they present new original research and illustrate how technology can help forward thinking companies cope with the downturn.

Visit student.accentureforum.dk to see the program and register

Visit student.accentureforum.dk

- Consulting • Technology • Outsourcing

accenture
High performance. Delivered.

© 2009 Accenture. All rights reserved.

```
setFirstName( firstName: string )  
getFirstName( ): string  
setLastName( lastName: string )  
getLastName( ): string  
setUserName( username: string )  
getUserName( ): string  
setPassword( pword: string )  
getPassword( ): string  
setCard( card: MembershipCard )  
getCard( ): MembershipCard
```

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.

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.

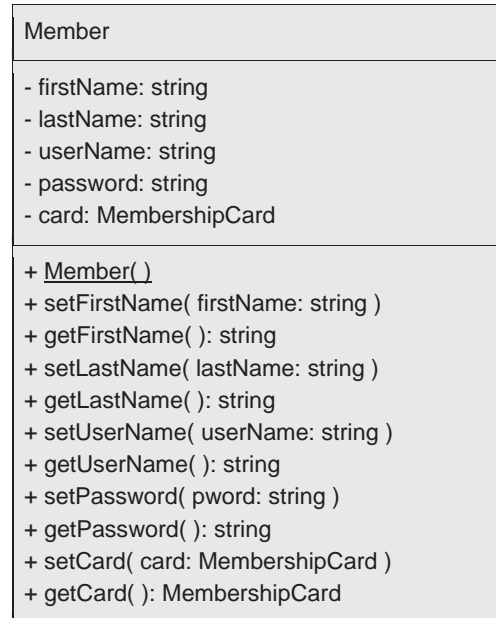
Remember that the attributes and behaviour in the lists above are a *language-independent* way of describing these two aspects of the class **Member**. However they are merely lists: what we need now is a diagram that helps us to design the class of our **Member** objects.

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.

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 see 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.