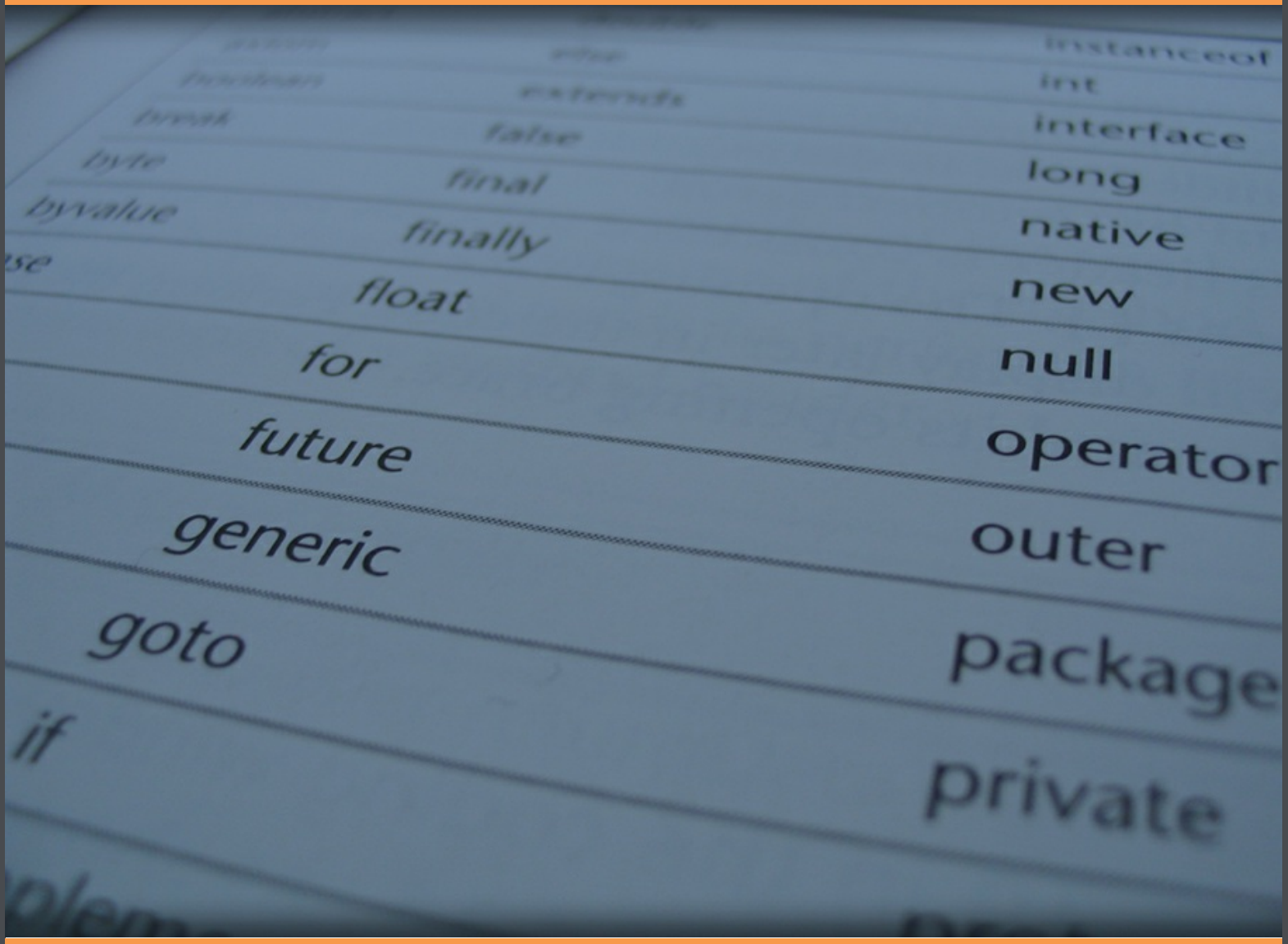
bookboon.com

Java: Classes in Java Applications

An Introduction to Java Programming

David Etheridge



Download free books at
bookboon.com

David Etheridge

Java: Classes in Java Applications

– An Introduction to Java Programming

Download free eBooks at bookboon.com

Java: Classes in Java Applications – An Introduction to Java Programming
© 2009 David Etheridge & Ventus Publishing ApS
ISBN 978-87-7681-495-3

Download free eBooks at bookboon.com

Contents

1.	**Using the Java Application Programming Interface**	**7**
1.1	Documentation in Developer-Written Java Classes	7
1.2	Documentation in the Java Application Programming Interface	15
2.	**Flow Control**	**19**
2.1	Introduction to Flow Control	19
2.2	Sequential Flow	19
2.3	Conditional Flow	19
2.4	Making Decisions	20
2.5	Controlling the Repetition of Blocks of Code	33
2.6	Deciding Which Construct to Use	41
2.7	Branching Statements	42
2.8	Handling Exception Objects	43
3.	**Extending Classes by Means of Inheritance**	**45**
3.1	What Does Inheritance Mean?	45
3.2	Overriding and Hiding Methods in a Subclass	53
3.3	Invoking a Parent Class Constructor from a Subclass Constructor	56
3.4	final and abstract Classes	57
3.5	What Does Type Compatibility Mean?	60

Download free eBooks at bookboon.com

Click on the ad to read more

3.6	Virtual Method Invocation	64
3.7	Controlling Access to the Members of a Class	65
3.8	Summary of Inheritance	68
4.	**Errors in Java Programmes**	**70**
4.1	Categories of Error	70
4.2	What are Unexpected Error Conditions?	71
4.3	Checked Exceptions	73
4.4	try … catch … finally Blocks	82
4.5	Throwing Exceptions	84
4.6	Exceptions in the Themed Application	87
4.7	Summary of Exceptions	92
5.	**Java Interfaces**	**93**
5.1	What is a Java Interface?	93
5.2	Defining and Implementing a Java Interface	97
5.3	The Role of Interfaces as a Means to Introduce Behaviour to a Class	101
5.4	Interfaces as Types	103
5.5	Summary of Java Interfaces	106
6.	**Grouping Classes Together in a Java Application**	**107**
6.1	An Introduction to Java Packages	107

Download free eBooks at bookboon.com

Click on the ad to read more

6.2	Creating Packages	107
6.3	Naming Convention	111
6.4	Packages in the Java Language	111
6.5	Using and Accessing Package Members	114
6.6	Compiling and Running Package Members	117

SIMPLY CLEVER

ŠKODA

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

Download free eBooks at bookboon.com

Click on the ad to read more

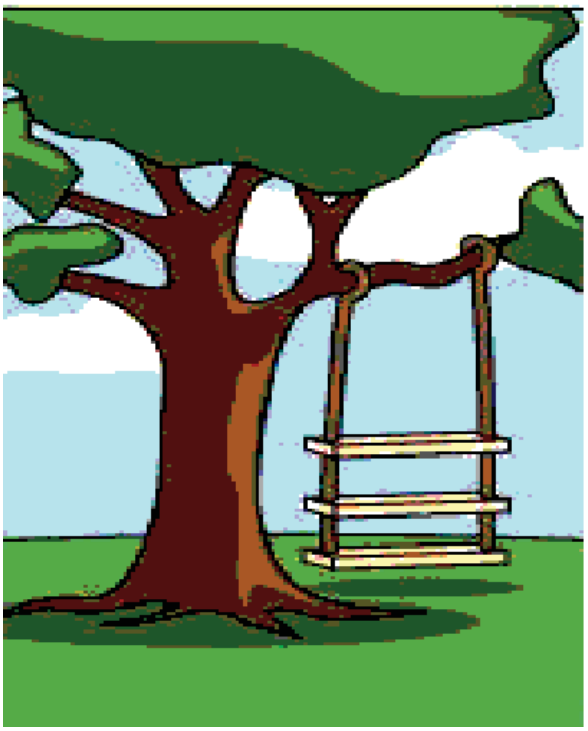
1. Using the Java Application Programming Interface

Chapter One takes examples from the Java Application Programming Interface (API) and the themed application in order to emphasise the critical importance of documentation. The examples are used to show how documentation is organised in the API and how it is inserted into developer's code.

1.1 Documentation in Developer-Written Java Classes

Previous chapters include a number of examples of classes (or partial classes) from the themed application, the main purpose of which is to illustrate programming concepts. It should not have escaped the notice of the reader that these examples often include material other than Java statements, usually in a non-bold font. The inclusion of such material raises a question: *what is the purpose of documentation is a class definition?*

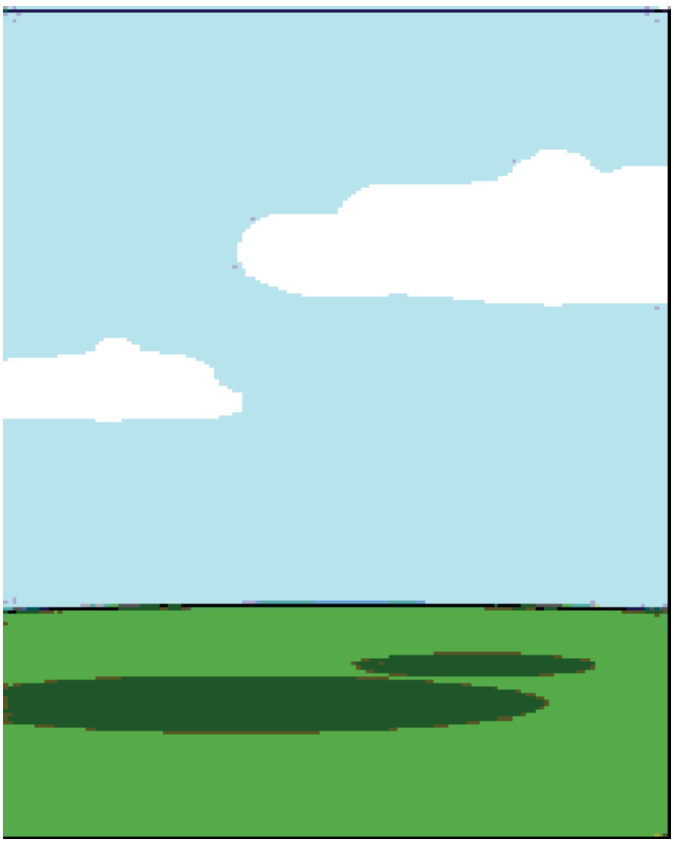
Perhaps one way to address this question is to consider figures 1.1 and 1.2 below.



Source: http://www.cvr-it.com/PM_Jokes.htm

Figure 1.1 A view of an application as explained by the users

Download free eBooks at bookboon.com



Source: http://www.cvr-it.com/PM_Jokes.htm

Figure 1.2 How the programme was documented

While it is obvious that the images are meant to be amusing, they make a serious point: *there is nothing worse than trying to maintain code written by someone else if there is little or no documentation.*

Documentation is an integral part of a Java class in that it is used, inter alia, to explain the purpose of members of the class (to the development team) and show consistency with the class diagram.

1.1.1 Documentation in the Themed Application

The class diagrams of two of the classes of the themed application are shown in the figure on the next page.

Download free eBooks at bookboon.com

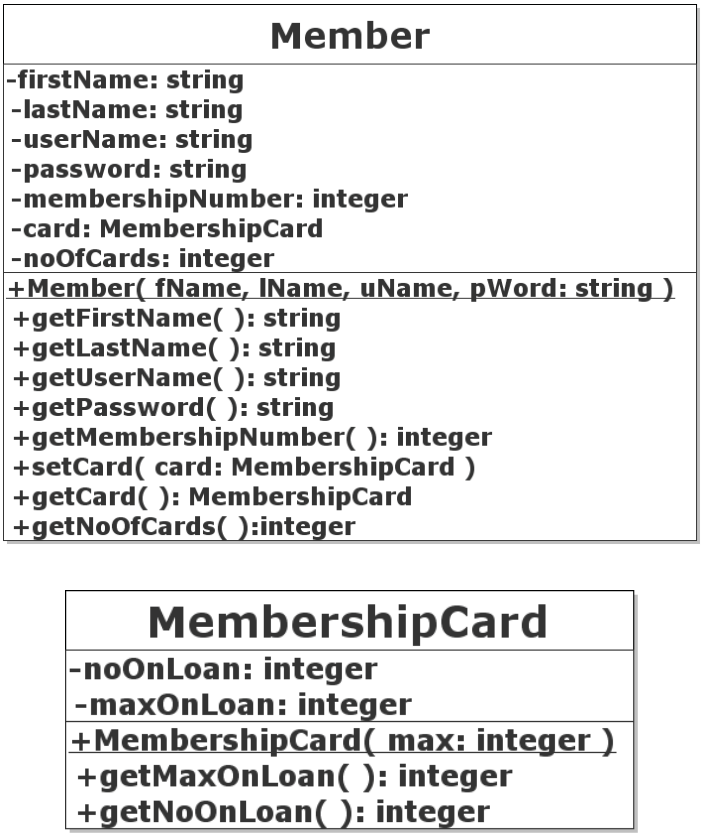


Figure 1.3 The Member and MembershipCard classes of the themed application

The classes are related by a 'has a' relationship, as shown in the next figure.

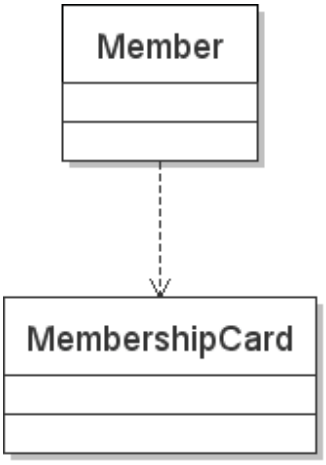


Figure 1.4 Each Member 'has a' MembershipCard

The source code for the **Member** class follows next. Some of the Java statements, white space and single-line comments are omitted for the sake of brevity so that the learner can concentrate on Java documentation, rather than on the logic of the code. Documentation blocks are displayed in a **bold** font.

```
/**
* The purpose of the Member class is … < to be completed by the developer >.
* @author  David M. Etheridge.
* @version 1.0, dated 29 October 2008.
*/
public class Member {
        private String firstName;
        private String lastName;
        private String userName;
        private String password;
        private int membershipNumber;
        private String noOfCards;
        private MembershipCard card;

        /**
        * This constructor is used to initalise the first four attributes.
        * @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.
        */
```

Download free eBooks at bookboon.com

Click on the ad to read more

```
public Member( String fName, String lName, String uName, String pWord ) {
        firstName = fName;
        lastName = lName;
        userName = uName;
        password = pWord;

} // End of constructor.

/**
* Accessor for the attribute firstName.
* @return  firstName  The value of the attribute firstName.
*/
public String getFirstName( ) {
        return firstName;
}  // End of definition of getFirstName.

/**
* Accessor for the attribute lstName.
* @return  lastName  The value of the attribute lastName.
*/
public String getLastName( ) {
        return lastName;
}  // End of definition of getLastName.

/**
* Accessor for the attribute userName.
* @return  userName  The value of the attribute userName.
*/
public String getUserName( ) {
        return username;
}  // End of definition of getUserName.

/**
* Accessor for the attribute password.
* @return  password  The value of the attribute password.
*/
public String getPassword( ) {
        return password;
}  // End of definition of getPassword.

/**
* Accrssor for the attribute membershipNumber.
* @return  membershipNumber  The value of the attribute
* membershipNumber.
*/
```

```
        public int getMembershipNumber( ) {
                return membershipNumber;
        }  // End of definition of getMembershipNumber.

        /**
        * Accessor for the attribute noOfCards.
        * @return  noOfCards  The value of the attribute noOfCards.
        */
        public String getNoOfCards( ) {
                return noOfCards;
        }  // End of definition of getNoOfCards.

        /**
        * Mutator for the attribute card.
        * @param  card  The member's membership card.
        */
        public void setCard( MembershipCard card ) {
                this.card = card;
        } // End of setCard.

        /**
        * Accessor for the attribute card.
        * @return  card  The member's membership card.
        */
        public MembershipCard getCard( ) {
                return card;
        } // End of setCard.

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

Comments in the source code that are placed between /** and */ are known as *documentation comments* and can be used to generate documentation about classes automatically. Comments that are tagged with '@' have a specific meaning in that they are used to refer to elements such as the programme's author, parameters and return values.

The (similarly simplified) class definition of the **MembershipCard** class follows on the next page.

Download free eBooks at bookboon.com

```
/**
* The purpose of the class MembershipCard is …
* @author David M. Etheridge.
* @version 1.0, dated 29 October 2008.
*/
public class MembershipCard {

        private int noOnLoan;
        private int maxOnLoan;

        /**
        * This constructor is used to initalis the maxOnLoan attribute.                *
        * @param  max  The maximum number of items permitted to be on loan
        * against this card.
        */
        public MembershipCard( int max ) {
                maxOnLoan = max;
        } // End of constructor.
```

Download free eBooks at bookboon.com

Click on the ad to read more

```
    /**
    * Accessor for the attribute noOnLoan.
    * @return  noOnLoan  The value of the attribute noOnLoan.
    */
    public int getNoOnLoan( ) {
            return noOnLoan;
    }  // End of definition of getNoOnLoan.

    /**
    * Accessor for the attribute maxOnLoan.
    * @return  maxOnLoan  The value of the attribute maxOnLoan.
    */
    public int getMaxOnLoan( ) {
            return maxOnLoan;
    }  // End of definition of getNoOnLoan.

} // End of class definition.
```

Most development environments for Java include a feature that runs the **javadoc** tool that is provided with the **javac** compiler and other java tools in Java's bin directory. When it is executed, the **javadoc** tool scans the tags and generates a set of linked HTML files. A snapshot of part of the documentation for the **MembershipCard** class is shown next.

Class MembershipCard

```
java.lang.Object
  └MembershipCard
```

`public class MembershipCard extends java.lang.Object`
The purpose of the class definition for the class Member is …

Version:

1.0, dated 29 October 2008.

Author:

David M. Etheridge.

Constructor Summary

`MembershipCard(int max)` This constructor is used to initalize the maxOnLoan attribute.	

Method Summary

`int`	`getMaxOnLoan()` Accessor for the attribute maxOnLoan.
`int`	`getNoOnLoan()` Accessor for the attribute noOnLoan.

Constructor Detail

MembershipCard

```
public MembershipCard(int max)
```

This constructor is used to initalize the maxOnLoan attribute.

Parameters: `max` - The maximum number of items permitted to be on loan against this card.

Method Detail

getMaxOnLoan

```
public int getMaxOnLoan()
```

Accessor for the attribute maxOnLoan.

Returns: maxOnLoan The value of the attribute maxOnLoan.

---

getNoOnLoan

```
public int getNoOnLoan()
```

Accessor for the attribute noOnLoan.

Returns: noOnLoan The value of the attribute noOnLoan.

The **javadoc** tool detects tags such as **@aurhor**, **@version**, **@param** and **@return** and generates the relevant HTML file, as seen by comparing the source code and documentation for the **MembershipCard** class shown above. Single-line comments are not detected by the **javadoc** tool; they are, however, an important component of documentation, as is evident by their extensive use in examples presented in previous chapters.

For further details about **javadoc** tags, the reader is referred to the section titled *Tag Conventions* in http://java.sun.com/j2se/javadoc/writingdoccomments/#sourcefiles

Using the **javadoc** tool is straightforward and should always to be sued to produce documentation for all classes written by Java developers as an integral part of the development process.

1.2 Documentation in the Java Application Programming Interface

The opening page of the version of the API (stored on the author's computer at the time of writing) is shown in the next screen shot.

Download free eBooks at bookboon.com

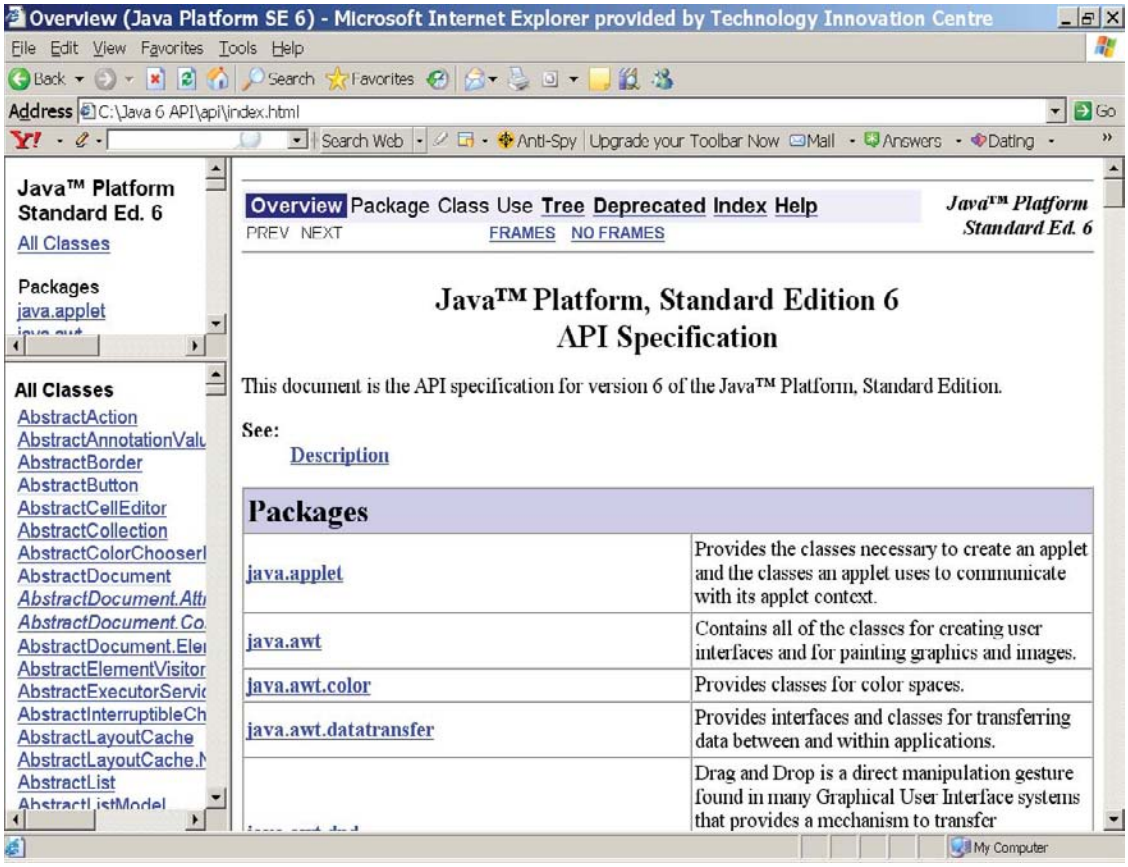


Figure 1.5 The Java API

Download free eBooks at bookboon.com

Click on the ad to read more

Part of the section for the **String** class is shown below.

java.lang

Class String

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

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String extends Object
implements Serializable, Comparable<String>, CharSequence
```

Constructor Summary

String() Initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes) Constructs a new String by decoding the specified array of bytes using the platform's default charset.
… other constructors follow but are not shown

Method Summary

char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
	... other methods follow but are not shown

Clicking on any of the constructors or methods reveals the details about that member. For example, clicking on the compareTo method of the String class displays the following page.

Download free eBooks at bookboon.com

`compareTo`

`public int` **`compareTo`**`(String anotherString)`

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns `0` exactly when the `equals(Object)` method would return `true`.

Specified by:
`compareTo` in interface `Comparable`<`String`>
Parameters:
`anotherString` - the `String` to be compared.
Returns:
the value `0` if the argument string is equal to this string; a value less than `0` if this string is lexicographically less than the string argument; and a value greater than `0` if this string is lexicographically greater than the string argument.

The principal purpose of the illustrations in this section is to show that the documentation provided by the API is organised in an identical fashion to that produced by the **`javadoc`** tool for developer-written classes, as exemplified in Section 1.1.1. This gives rise to consistently-structured documentation for the infinitude of Java classes used and written by the worldwide Java development community.

While the examples in Section 1.1.1 are relatively straightforward in terms of the logic of Java source code, they are included to illustrate the essential principles and purpose of providing single-line and block documentation as an integral part of writing class definitions.

The next chapter returns to the Java language itself and explores how flow of control is managed in blocks of Java source code.

Download free eBooks at bookboon.com

2. Flow Control

2.1 Introduction to Flow Control

The Java language provides a number of constructs that enable the developer to control the sequence of execution of Java statements. Chapter Two provides examples of how these constructs are used to control the flow of execution through a block of code that is typically contained in the body of a method.

2.2 Sequential Flow

Sequential flow of execution of statements is the execution of Java source code in a statement-by-statement sequence in the order in which they are written, with no conditions. Most of the examples of methods that are discussed in previous chapters exhibit sequential flow. In general terms, such a method is written as follows.

```
public void someMethod( ) {

    // first statement to execute
    // second statement to execute
    // third statement to execute
    // and so on, to the final statement
    // final statement to execute

} // end of method definition
```

A number of the **main** methods, presented in previous chapters, are structured in this sequential way in order to satisfy straightforward testing criteria.

2.3 Conditional Flow

While sequential flow is useful, it is likely to be highly restrictive in terms of its logic. Executing statements *conditionally* gives the developer a mechanism to control the flow of execution in order to repeat the execution of one or more statements or change the normal, sequential flow of control. Constructs for conditional flow control in Java are very similar to those provided by other programming languages. Table 2.1 on the next page identifies the flow control constructs provided by the Java language.

Download free eBooks at bookboon.com

Statement Type	Key words
Decision	if … then
Decision	if … else
Decision	switch … case
Loop	for
Loop	while
Loop	do … while
Branching	break: labelled and unlabelled form
Branching	continue: labelled and unlabelled form
Exception handling (see Chapter Four)	try … catch

Table 2.1 Flow control constructs

The sub-sections that follow show, by example, how these constructs are used.

2.4 Making Decisions

Using a decision-making construct allows the developer to execute a block of code *only* if a condition is true. The sub-sections that follow illustrate how decision-making constructs are used.

Download free eBooks at bookboon.com

Click on the ad to read more

2.4.1 The if ... then Construct

The **if ... then** construct is the most basic of the decision-making constructs provided by the Java language. If a condition is true, the block of code is executed: otherwise, control skips to the first statement after the **if** block. The following code snippet illustrates a simple use of the **if ... then** construct.

```
// assume that the value of age has been entered via the keyboard
if ( age >= 18 ) // the if condition is placed between ( and )
{ // start of if block
        System.out.println( “You can drink legally.” ); // the then clause
} // end of if block
// execute the next statement
System.out.println( “The rest of the programme is next.” );
```

When the code snippet is run (in a **main** method), the output when **age = 20** is:

```
You can drink legally.
The rest of the programme is next.
```

and when **age = 17**, the output is:

```
The rest of the programme is next.
```

In some programming languages, the word ‘then’ is included in the **then** clause. As the code snippet above shows, this is not the case in Java.

An example taken from the themed application shows an **if ... then** construct in action in one of the methods of the **Member** class. The method adds a member to the array of members only if there is room in the array of (arbitrary) size 6.

```
/**
* This method adds a member if there is room in the array of members called members.
* @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 ) {

        if( noOfMembers < 6 )
        {
                members[ noOfMembers ] = new Member( fName,
                        lName, uName, pWord );
```

Download free eBooks at bookboon.com

```
                System.out.println( "The member has been added." );
                // Increment the number of members.
                noOfMembers++;
            }
            System.out.println( "No room for another member." );

        } // End of addMember.
```

If there is no room in the array because **noOfMembers** is equal to or greater than 6, control skips to the print statement that outputs the message “No room for another member.”

2.4.2 The if … else Construct

The **if … else** construct (sometimes known as the **if … then … else** construct) provides an alternative path of execution if the **if** condition evaluates to false. Figure 2.1 illustrates, diagrammatically, the logic of the **if … else** construct.

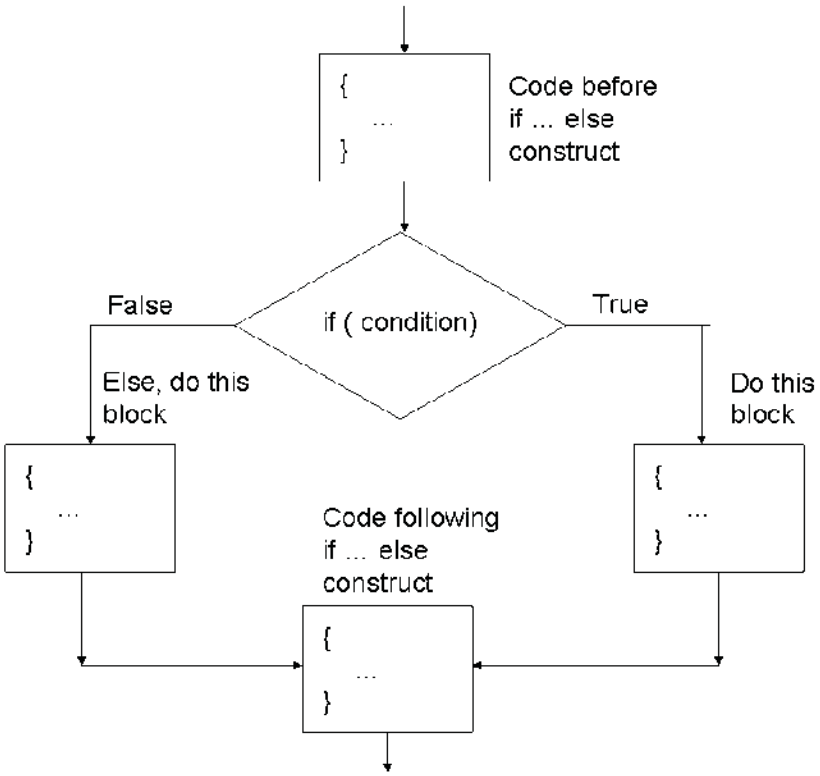


Figure 2.1 The logic of the **if … else** construct

Flow of control enters the **if** clause and the **if** condition is tested. The result of evaluating the **if** condition returns either **true** or **false** and one or other of the paths of execution are followed depending on this value. The **else** block is executed if the **if** condition is **false**.

The next code snippet illustrates a simple use of the **if … else** construct by modifying the first code snippet in Section 2.4.1.

Download free eBooks at bookboon.com

```
if ( age >= 18 ) // the if condition
{ // start of if block
        System.out.println( "You can drink legally." ); // the then clause
} // end of if block
else
{ // start of else block
        System.out.println( "You are too young to drink alcohol!" );
} // end of else block
// execute the next statement
System.out.println( "The rest of the programme is next." );
```

When the code snippet is run (in a `main` method), the output when `age = 20` is:

```
You can drink legally.
The rest of the programme is next.
```

and when `age = 17`, the output is:

```
You are too young to drink alcohol!
The rest of the programme is next.
```

Download free eBooks at bookboon.com

Click on the ad to read more

Another example taken from the themed application shows an **if ... else** construct in action in another of the methods of the **Member** class. The **setCard** method is used to associate a member of the Media Store with a virtual membership card. Each member may have up to two cards, so the method checks whether another card can be allocated to a member.

```
/**
* This method gives a card to a member by adding it to the member's array of (two) cards.
* The array has the identifier cards.
* @return result  A boolean value to state whether the addition of a card is possible.
* @param card  A parameter of the MembershipCard type.
*/
public boolean setCard( MembershipCard card ) {

        // declare a local variable
        boolean result = true;
        // noOfCards is the number of cards allocated to the member
        if ( noOfCards < cards.length ) {
                cards[ noOfCards ] = card;
                noOfCards++;
        }
        else {
                System.out.println( "No more cards allowed for this member." );
                result = false;
        }
        return result;

} // End of setCard.
```

The **if ... else** construct in the method is used to return either **true** or **false**, depending upon the result of evaluating the **if** condition that determined whether or not the member has fewer than two cards.

2.4.3 Compound if ... else Constructs

There is another form of the **else** part of the **if .. else** construct: **else ... if.** This form of compound or cascading construct executes a code block depending on the evaluation of an **if** condition immediately after the initial **if** condition. The compound **if ... else** construct is illustrated diagrammatically in Figure 2.2 below.

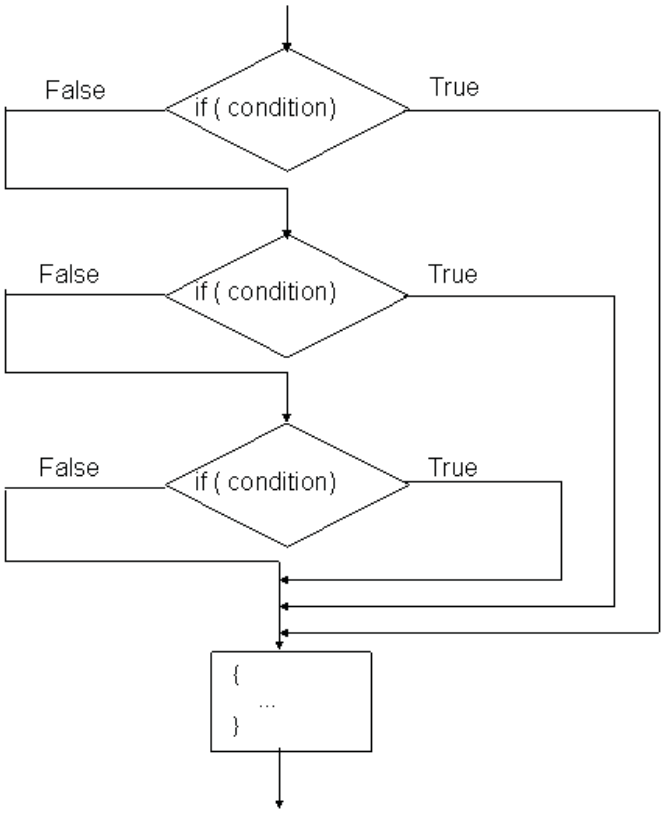


Figure 2.2 The logic of the compound **if … else** construct

The figure shows that any number of **else ... if** statements can follow the initial **if** statement.

The example on the next page illustrates how the **if .. else** construct is used to identify the classification for degrees awarded by universities in the United Kingdom, based on the average mark achieved in the final year.

```
// declare two local variables
int average  = 0;
String result = null;
if( average >= 70 )
{
        result = "First Class";
}
else if( average >= 60 )
{
        result = "Upper Second";
}
else if( average >= 50 )
{
        result = "Lower Second";
}
else if( average  >= 40 )
{
        result = "Pass";
}
else
{
        result = "You are going to have to tell your mother about this!";
}
System.out.println( "Your result is: " + result);
```

American online
LIGS University
is currently enrolling in the
Interactive Online BBA, MBA, MSc,
DBA and PhD programs:

- enroll by September 30th, 2014 and
- save up to 16% on the tuition!
- pay in 10 installments / 2 years
- Interactive Online education
- visit www.ligsuniversity.com to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education.
More info here.

Download free eBooks at bookboon.com

Click on the ad to read more

Running the code with an average of 30 % produces the following output:

Your result is: You are going to have to tell your mother about this!

and with an average of 65 %, the output is as follows:

Your result is: Upper Second

When the value of average is equal to 65, this satisfies more than one of the **else ... if** statements in the code above. However, the output confirms that the first time that a condition is met – when average >= 60 – control passes out of the initial **if** statement without evaluating the remaining conditions. When a condition is met in the code above, the output shows that control skips to the first statement after the initial **if** statement, i.e. to the statement

System.out.println("Your result is: " + result);

It is worthwhile alerting learners to the use of braces in compound **else ... if** constructs. Care must be taken when coding compound **else .. if** constructs due to the number of pairs of brackets involved: a common error is to omit one or more of these brackets. In cases where there is only one statement in an **if** block, it is good practice to include braces – as shown in the example above – in anticipation of **if** blocks that include more than one statement.

The final example in this sub-section shows a compound **else ... if** construct in action in the **Member** class of the themed application. The method scans the array of (virtual) cards held by a member and outputs some information that is stored against each card. (**for** loops are discussed in a later section of this chapter.)

```
/** This method scans the array of cards in a for loop.  */
public void getDetialsOfCards( ) {

    // Declare a local variable.
    MembershipCard card = null;
    // note the use of the instanceof operator
    for ( int i = 0; i < noOfCards; i++ )
    {
        if ( cards[ i ] instanceof DvdMembershipCard )
    {
            card = cards[ i ];
            System.out.println( "This is a DVD card with " + getNoOnLoan( )
                + " DVDs currently on loan." );
    } else if ( cards[ i ] instanceof GameMembershipCard )
    {
            card = cards[ i ];
            System.out.println( "This is a games card with " +
                getNoOnLoan( ) + " CDs currently on loan" );
    } else
    {
```

```
                    System.out.println( "Neither type of card." );
            }
            } // End of for loop.

    } // End of getDetailsOfCards.
```

2.4.4 Nested if Statements

As an alternative to compound **if** statements, as described in sub-section 2.4.3, **if** statements can be nested if the method demands this kind of logic. The simple example of nesting **if** statements shown next is a variant of the first example in sub-section 2.4.3.

```
int average = 65;
String result = null;
String course = "Java";
if ( course == "Java" )
{ // start of outer if
        if( average >= 70 ) // start of inner if
        {
                result = "First Class";
        }
        else if( average >= 60 )
        {
                result = "Upper Second";
        }
        else if( average >= 50 )
        {
                result = "Lower Second";
        }
        else if( average >= 40 )
        {
                result = "Pass";
        }
        else
        {
                result = "You are going to have to tell your mother about this!";
        }
        System.out.println( "Your result is: " + result);
} // end of outer if
System.out.println( "No more results are available." );
```

When the code is run, the output is

```
Your result is: Upper Second
No more results are available.
```

When the next version is run, the output is

No more results are available.

```
int average = 65;
String result = null;
String course = "C++";
if ( course == "Java" )
{ // start of outer if
        if( average >= 70 ) // start of inner if
        {
                result = "First Class";
        }
        else if( average >= 60 )
        {
                result = "Upper Second";
        }
        else if( average >= 50 )
        {
                result = "Lower Second";
        }
        else if( average >= 40 )
        {
                result = "Pass";
        }
        else
        {
                result = "You are going to have to tell your mother about this!";
        }
        System.out.println( "Your result is: " + result);
} // end of outer if
System.out.println( "No more results are available." );
```

Care should be taken when using any of the variants of the **if** statement to ensure and test that the required logic is implemented in the construct. It is often helpful to draw a diagram of the logic required, in order to help decide which variant to use to meet specific requirements.

2.4.5 The Conditional Operator

Java provides a ternary, conditional operator **? :** that is a compact version of an **if ... else** statement. The operator takes three operands: the first is a boolean condition; the second is the result if the condition is **true**; the third is the result if the condition is **false**.

Download free eBooks at bookboon.com

Let us recall a previous example from this chapter.

```
if ( age >= 18 ) // the if condition
{ // start of if block
        System.out.println( “You can drink legally.” ); // the then clause
} // end of if block
else
{ // start of else block
        System.out.println( “You are too young to drink alcohol!” );
} // end of else block
// execute the next statement
System.out.println( “The rest of the programme is next.” );
```

The logic of the **if .. else** statement can be re-written as follows, using Java’s ternary operator.

```
int age = 21;
System.out.println( age >= 18 ? "Old enough to drink" : "Too young to drink" );
System.out.println( "The rest of the programme is next." );
```

and produces the following output when run:

```
Old enough to drink
The rest of the programme is next.
```

The next and final sub-section in the category of decision-making constructs describes the **switch ... case** construct.

2.4.6 The switch ... case Construct

The **switch ... case** construct is an alternative to compound **if** statements if the condition is an evaluation of an integer expression. As such, it is often easier to code than compound **if** statements in that is it less prone to errors such as the omission of brackets. The **switch ... case** construct is illustrated diagrammatically in Figure 2.3 on the next page.

Download free eBooks at bookboon.com

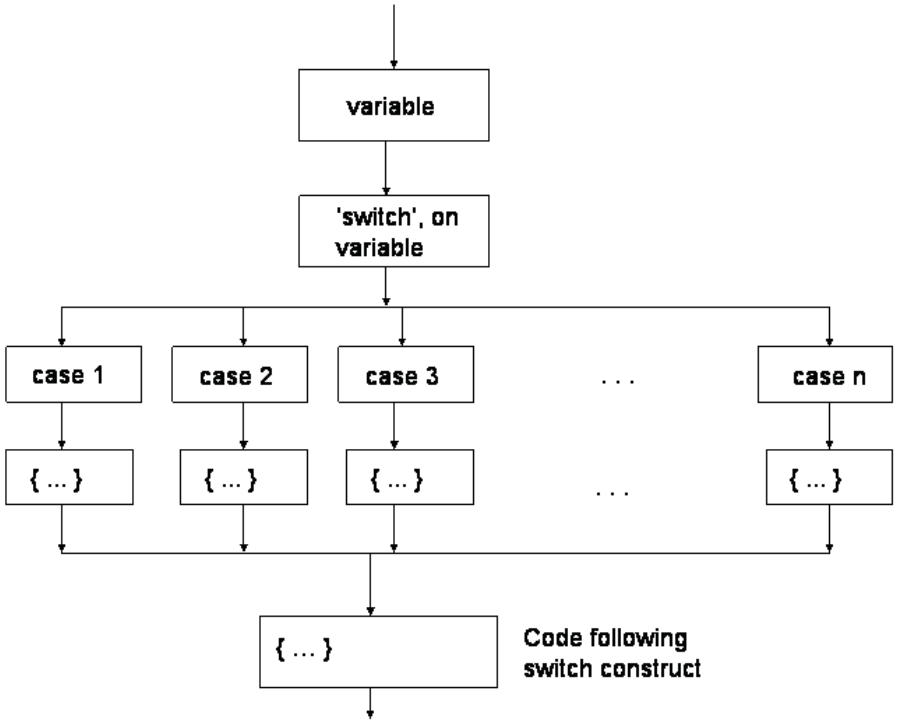


Figure 2.3 The logic of the **switch ... case** construct

Download free eBooks at bookboon.com

Click on the ad to read more

The logic of the **switch ... case** construct is such that the integer condition is tested against each case in order from left to right. When this logic is translated in Java source code, the generalised syntax is as follows.

```
int someIntegerValue;
switch ( someIntegerValue ) { // start of switch block
        case 1: // do something; break;
        case 2: // do something else; break;
        case 3: // do something else ; break;
        case 3: // do something else; break;
        case 5: // do something else; break;
} // end of switch block
// first statement after the switch block
```

The **break** statement after each **case** statement is necessary to exit the enclosing **switch** block when the **switch** condition has been satisfied. When a **break** statement is executed, control passes out of the enclosing **switch** block to the first statement after the end of the **switch** block. On the face of it, it would seem logical to omit the final **break** statement. However, it is advisable to include it in case additional **case** statements are added to an existing **switch** block.

In the days before the ubiquitous use of icon-driven applications, old-fashioned text-based user interfaces for green screen types of applications were often menu-driven. **Case** statements were typically used to construct this kind of interface. While it is generally true that menu-driven applications have largely disappeared, **case** statements are useful for testing the conditional flow through application logic.

The following example illustrates testing a **switch ... case** construct with a value entered via the keyboard. (It isn't necessary to show the code used to capture a number via the keyboard for the purposes of the example.)

```
// a number entered via the keyboard is stored in a variable with the identifier month
int days;
switch( month )
{
        case 9:
        case 4:
        case 6:
        case 11: days = 30; break;
        case 2:  days = 28; break;
        default: days = 31; break;
}
System.out.println( "The number of days is: + days );
```

When this code is run in a **main** method, the output is as follows:

```
Enter the number of the month: 1
The number of days is: 31
```

Download free eBooks at bookboon.com

```
Enter the number of the month: 2
The number of days is: 28

Enter the number of the month: 3
The number of days is: 31

Enter the number of the month: 4
The number of days is: 30

Enter the number of the month: 5
The number of days is: 31

Enter the number of the month: 6
The number of days is: 30

Enter the number of the month: 7
The number of days is: 31

Enter the number of the month: 8
The number of days is: 31

Enter the number of the month: 9
The number of days is: 30

Enter the number of the month: 10
The number of days is: 31

Enter the number of the month: 11
The number of days is: 30

Enter the number of the month: 12
The number of days is: 31
```

The output shows that the default statement is used to detect all values that aren't detected by any of the **case** statements. The output also shows that when the value of **month** is **9**, **4** or **6**, the first three **case** statements are said to 'fall through' so that the value of **days** is **30** when the value of **month** is **9**, **4**, **6** or **11**.

2.5 Controlling the Repetition of Blocks of Code

Controlling the repetition of a block of code can be achieved in one of two ways: by using a counter to repeat a block of code a known number of times; by using the evaluation of a boolean expression to decide when to stop repeating the block. The general requirement to repeat a block of code is illustrated in Figure 2.4 on the next page.

Download free eBooks at bookboon.com

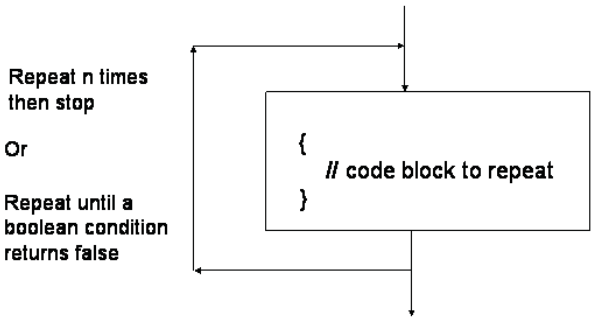


Figure 2.4 The requirement to repeat a block of code

2.5.1 Counter-Controlled Repetition

The logic of counter-controlled repetition is visualised in Figure 2.5.

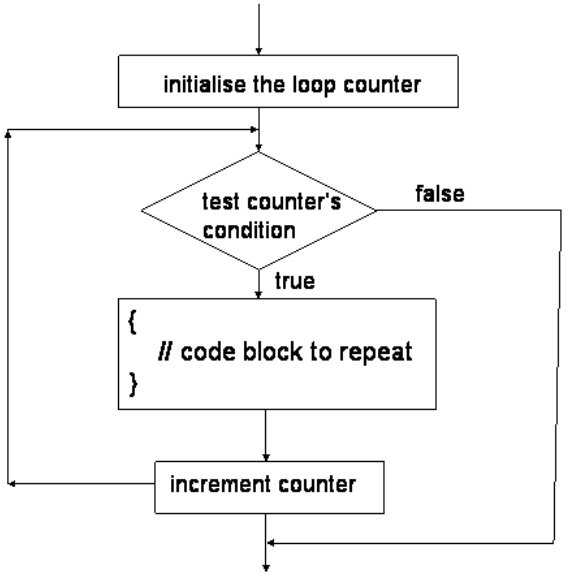


Figure 2.5 Counter-controlled repetition of a block of code

Working with the counter is implemented by what is known as a **for loop**. The general syntax of a **for loop** is as follows:

```
for ( declare and initialise the counter;
      final value condition of the counter;
      update counter )
{
        // block to repeat
}
```

The general syntax implies that the value of the final condition of the counter is a *known* value. For example, consider the following code snippet:

```
for ( int i = 0; i <= someMaxValue; i ++ )
{
        // code block to repeat
}
```

If the value of **someMaxValue** is **10**, the code block will execute 11 times. If the code snipped is modified to read as follows with the same value of **someMaxValue**

```
for ( int i = 0; i < someMaxValue; i ++ )
{
        // code block to repeat
}
```

the code block will execute 10 times. The purpose of including the two code snippets above is to show that care must be taken when initialising and setting the final value of the counter's condition when controlling the number of times that a **for loop** is executed.

The example, shown on the next page, shows a **for loop** in action in one of the methods of the **Member** class in the themed application,.

Download free eBooks at bookboon.com

Click on the ad to read more

```
/**
* This method outputs the details of the cards held by the member by scanning across the
* member's array of cards with the identifier cards.
*/
public void getDetialsOfCards( ) {

        // Declare a local variable.
        MembershipCard card = null;

        // Note the use of the instanceof operator.
        for ( int i = 0; i < noOfCards; i++ ) // the value of noOfCards is known
        {
                if ( cards[ i ] instanceof DvdMembershipCard )
                {
                        card = cards[ i ];
                        System.out.println( "This is a DVD card with " +
                                card.getNoOnLoan( ) + " DVDs currently on loan." );
                } else if ( cards[ i ] instanceof GameMembershipCard )
                        {
                                card = cards[ i ];
                                System.out.println( "This is a games card with " +
                                        card.getNoOnLoan( ) + " CDs currently on
                                        loan" );
                        } else System.out.println( "Neither type of card." );
        } // End of for loop.

} // End of getDetailsOfCards.
```

The purpose of the **for loop** in the method is to scan the array of a member's (virtual) cards in order to output some information stored on each card. In the example, the number of cards held by a member is known: therefore, a **for loop** is the clear choice of construct to use to repeat the required block of code.

2.5.2 Boolean-Controlled Repetition

Java provides two boolean-controlled constructs to control the repetition of a block of code: the **while loop** and the **do ... while loop**.

The while loop

The logic of the **while** loop is visualised in Figure 2.6 below.

Download free eBooks at bookboon.com

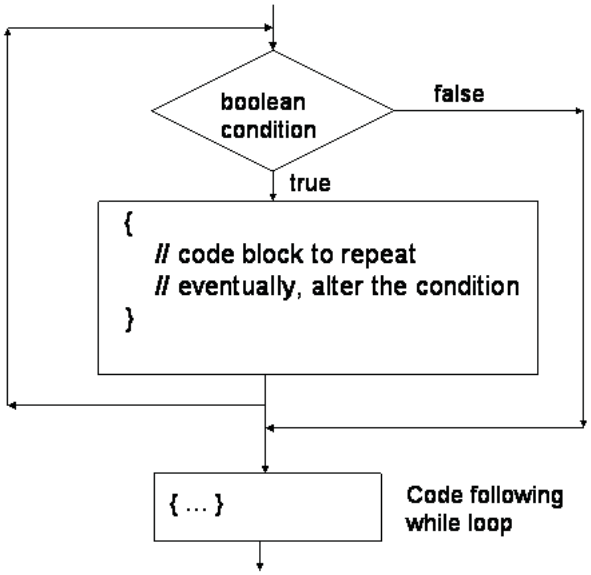


Figure 2.6 The logic of a **while** loop

Figure 2.6 implies that if the boolean condition returns an initial value of 'false', the loop will not execute at all. If, on the other hand, the boolean condition returns an initial value of 'true', the loop will repeat its execution until such time as the condition eventually returns 'false'. The general syntax of a **while** loop is as follows.

```
// initialise the variable used in the boolean condition
while( condition is true )
{
        {
                // statements to repeat
        }
        // update the condition: finally, exit the loop
}
```

The first example from Section 2.5.1 can be modified to use a **while** loop as shown on the next page.

```
int someMaxValue = 10;
int counter = 0;
while( counter < someMaxValue )
{
        {
                System.out.println( "Hello world!" );
        }
counter ++ ;
}
```

The **String** "Hello World!" is output ten times.

The do …while loop

The logic of the **do … while** loop is visualised in Figure 2.7 below.

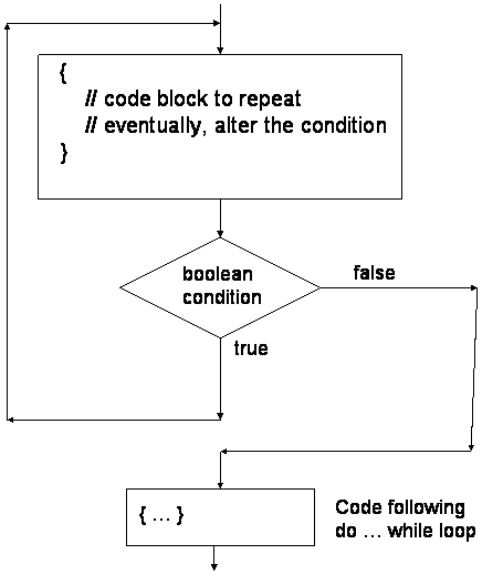


Figure 2.7 The logic of a **do … while** loop

Figure 2.7 implies that if the boolean condition returns a value of 'false', the loop will execute once. If, on the other hand, the boolean condition returns a value of 'true', the loop will repeat its execution until such time as the condition eventually returns 'false'. The general syntax of a **do … while** loop is shown on the next page.

Download free eBooks at bookboon.com

```
// initialise the variable used in the boolean condition
do
{
        {
                // statements to repeat
        }
        // update the condition: finally, exit the loop
} while( condition is true );
```

The same example from Section 2.5.1 can be modified to use a **do ... while** loop as shown next.

```
int someMaxValue = 10;
int counter = 0;
do
{
        {
                System.out.println( "Hello world!" );
        }
counter ++ ;
} while( counter < someMaxValue );
```

Download free eBooks at bookboon.com

Click on the ad to read more

In this case, the **String** "Hello World!" is output ten times.

The next example is taken from one of the classes of the graphical user interface (GUI) used to run the themed application; the example shows a **while** loop in action. The purpose of the loop is to scan the array of members of the Media Store when a member logs in via the GUI. When the member is found in the array, the loop is exited to avoid searching the remainder of the array.

```
// This method finds out if a member has registered with the Media Store and has been added
// to the array of members.  The method responds to the 'Login' button when a member
// attempts to login via the GUI.  The main purpose of the method is to find the member in the
// array of members.
// Read in the array of members. mediaStore is the reference to the MediaStore object created
// elsewhere in the application.
mediaStore.readMembers( );
// Capture the member's user name and password from the GUI.  Store these values in local
// variables username and password.
// Concatenate the user's 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( );
// Scan the array of existing members and compare the search string with each member's
// combined user name and password.
boolean flag = true;
while( flag == true )
{
        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.
                // Output a message to the GUI.
                        flag = false;
                break; // out of the for loop
                } // end if
        }  // end of for loop
break; // out of the while loop
} // end of while loop
// if there is no match, output a suitable message
```

```
if ( flag == true )
{
        // output "No such member; please try again." );
}
```

2.6 Deciding Which Construct to Use

Deciding which construct to use in any particular situation is a matter of judgement that will become easier to make when the learner gains practical experience. For example, the method implementation explained at the end of the previous sub-section is a direct consequence of the logic required to find a member in the array of members and then break out of the loop to avoid unnecessary iterations of the main loop. As can be seen from the code, the full implementation of the method is a combination of nested **if**, **while** and **for** constructs.

Before completing our examination of the contents of Table 2.1, it is worthwhile mentioning another 'loop within a loop' nested construct.

2.6.1 Nested for loops

One of my university colleagues sets a fiendish exercise to students enrolled on his Java course. The main task of the exercise is to print a calendar. While on the face of it this may seem a straightforward exercise, it actually involves a number of loops within loops, generally referred as *nested for loops*. A highly generalised skeleton solution to this exercise might read as follows.

```
for ( int months = 1; months < 13; months ++ ) // the month loop
{
        for ( int weeks = 1; weeks < 5; weeks ++ ) // the week loop
        {
                for ( int days = 1; days < 32; days ++ ) // the day loop
                {
                        // output the day and date in a calendar format
                }
        }
} // end of outer for loop
```

The outline example above suggests that nested **for** loops can be used to construct tables or two-dimensional arrays of data. The inner loop is used to output each entry in a row and the outer loop can be used to move to the next column. The next code snippet outputs the value of each cell of a table or array.

```
for ( int i = 1; i <= numberOfColumns; i++ )
{
        for ( int j = 1; j <= cellValue; j++ )
        {
                // output the known value of the cell at co-ordinates j , i, i.e. the jth row of the
                // ith column
```

Download free eBooks at bookboon.com

```
            } // end of inner for loop
        } // end of outer for loop
```

The next section almost completes our examination of Table 2.1 by considering *branching statements*.

2.7 Branching Statements

Branching statements are used to terminate a loop or a decision construct or to skip an iteration of a loop.

2.7.1 The Unlabelled break Statement

A number of the examples discussed in this chapter include **break** statements to expedite the immediate exit from a flow control construct. The **break** statement is used to terminate **switch**, **for**, **while** and **do ... while** constructs. For example, when the following method is invoked

```
public void someMethod( ) {

        for ( int i = 0; i < 11; i++ )
        {
                System.out.print( i + ", " );
                if ( i == 5 )
                break; // out of the enclosing for loop using an unlabelled break
        }
        System.out.println( "for loop terminated." );

} // End of someMethod
```

the output is:

0, 1, 2, 3, 4, 5, for loop terminated.

The output shows that the **break** statement causes the enclosing **for** loop to terminate and control passes to the first statement after the **for** loop.

2.7.2 The Unlabelled continue Statement

We have not encountered the **continue** statement thus far in this guide. The **continue** statement is used to skip an iteration of **for**, **while** and **do ... while** loops.

For example, when the method shown on the next page is invoked, it produces the output displayed after the body of the method.

Download free eBooks at bookboon.com

```
public void someMethod( ) {

        for ( int i = 0; i < 11; i++ )
        {
                if ( i == 5 )
                continue; // skip this iteration of the loop using an unlabelled continue
                        // statement
                System.out.print( i + ", " );
        }
        System.out.println( "The iteration when i is 5 is skipped." );

} // End of someMethod.
```

The output is:

0, 1, 2, 3, 4, 6, 7, 8, 9, 10, The iteration when i is 5 is skipped.

The output shows that the **continue** statement causes the enclosing **for** loop to skip the iteration when **i** is **5** and control passes to the next iteration of the enclosing loop.

The examples in sub-sections 2.7.1 and 2.7.2 illustrate the use of the *unlabelled* form of the **break** and **continue** statements. When loops are nested, the *labelled* form of the **break** and **continue** statements identify which of the outer loops are involved in a branch. The reader is referred to the relevant section of Sun's on-line Java tutorial for examples that explain the labelled form of the branching statements.

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

2.8 Handling Exception Objects

The final row of Table 2.1 mentions a special kind of decision, namely the one that uses the **try ... catch** construct to detect error conditions in Java programmes. Chapter Four explains how errors are detected by objects of the **Exception** class. For the purposes of this chapter, and to complete the consideration of Table 2.1, one way of looking at the **try ... catch** block is to consider the generalised code snippet that is shown on the next page.

Download free eBooks at bookboon.com

```
// one or more of the methods invoked in the next code block are known to produce errors, i.e.
// they are said to 'throw' errors that must be detected or 'caught' in a separate code block
try
{
        // method invocations that throw errors are coded here
}
catch ( Exception e )
{
        // do something about the error
}
```

The structure of the **try ... catch** construct implies that a decision is made depending upon whether an error is detected or not. If an error is thrown by one of the method invocations in the **try** block, the remaining statements of the **try** block are skipped and the statements in the **catch** block are executed. If, on the other hand, method invocations in the **try** block do not throw any errors, the statements of the **catch** block are skipped and control passes to the first statement after the end of the **catch** block. Chapter Four goes into details about how **try ... catch** blocks are used to write robust Java code.

In the next chapter, we will find out how we can extend classes by means of a very important concept known as *inheritance*.

Download free eBooks at bookboon.com

Click on the ad to read more

3. Extending Classes by Means of Inheritance

Chapter Three explores one of the cornerstones of OOP, namely that of *inheritance*. Java classes can be easily modified and extended for the purpose of re-use by means of *inheritance* rather than by re-writing the source code of the class to be modified.

3.1 What Does Inheritance Mean?

In order to explain what inheritance means in the Java language, let us consider a very simple example. Suppose that we have a class called **MyClass**, with the following trivial class definition that declares a single member:

```
public class MyClass {

        private int someValue;

}
```

and suppose that we wish to *re-use* this class and, in doing so, we wish to give the replica a *different* name. To do this, we would write the following class definition:

```
public class MyOtherClass {

        private int someValue;

}
```

In other words, we have re-written the body of **MyClass** in the class definition of **MyOtherClass**. In our trivial example, the effort of re-writing the body of **MyClass** is minimal in that there in only one member to declare. In the general case, on the other hand, the effort of re-writing a large class definition ought to be avoidable.

The concept of *inheritance* is common to OOP languages and avoids re-writing code from one class to another. In the example above, the replica class definition is written as follows, to take advantage of the concept of inheritance as it applies in Java.

```
public class MyOtherClass extends MyClass {
        // the variable someValue is inherited and does not have to be declared again
}
```

The keyword **extends** indicates that the class **MyOtherClass** *inherits* the single member of the class **MyClass** so that it becomes a members of **MyOtherClass**.

Download free eBooks at bookboon.com

The use of the keyword **extends** in the simple example above raises a question: *what is the purpose of replicating MyClass in the guise of MyOtherClass when we could simply use it as many times as we wish?* The answer to this question lies in the fact that inheritance means that inherited members can either be modified or left unchanged. This important concept means that we can modify an inherited method if we wish to and we are free to leave it (or other) inherited methods unchanged. We can also provide *additional* methods in an extended class.

The next example enhances the one above to illustrate the true value of extending a class.

```
public class MyClass {

        private int someValue;

        public void someMethod( String string ) {
                System.out.println( “This is the parent class.” );
        }

}
```

and

```
public class MyOtherClass extends MyClass{

        // MyOtherClass inherits both members of MyClass.

}
```

Given that **MyOtherClass** inherits both members of **MyClass**, consider the test class shown next.

```
public class TestClass {
         public static void main( String[ ] args ) {
                MyOtherClass moc = new MyOtherClass( );
                System.out.println( "The value of someValue is: " + moc.someValue );
        }
}
```

When an attempt is made to compile the test class, the compiler outputs the following message:

someValue has private access in MyClass

The purpose of this simple example is to show that although the private variable **someValue** is inherited by **MyOtherClass**, it is a private variable and, as such, is not directly accessible by selecting the member via a reference to an object of the class **MyOtherClass**. The compiler highlights the statement

```
System.out.println( "The value of someValue is: " + moc.someValue );
```

Download free eBooks at bookboon.com

when it outputs its message because it is complaining that

moc . someValue

attempts to access a private variable, albeit an inherited one. Thus we can see that the compiler is consistent when we attempt to access a private variable directly whether or not the variable is inherited.

In order to access the variable **someValue**, we could write an accessor method in **MyClass** and invoke it via a reference to an object of **MyOtherClass** in a test class as shown next.

```
public class MyClass {
        private int someValue = 42;

        public void someMethod( String string ) {
                System.out.println( “This is the parent class.” );
        }

        public int getSomeValue( ) {
                return someValue;
        }
}
```

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

```
public class TestClass {
        public static void main( String[ ] args ) {
                MyOtherClass moc = new MyOtherClass( );
                System.out.println( "The value of someValue is: " +
                        moc.getSomeValue( ) );
        }
}
```

Executing **main** produces the following output.

```
The value of someValue is: 42
```

and shows that the public accessor method **getSomeValue** is inherited by **MyOtherClass**.

Alternatively, we could *modify* the inherited method **getSomeValue** in order to carry out some simple processing on **someValue** as shown next.

```
public class MyOtherClass extends MyClass {

        public int getSomeValue( ) {

                someValue ++ ;
                return someValue;

        }

}
```

The modified method does not compile because it attempts to access an inherited private variable. However, the following modification of the method *does* compile:

```
public class MyOtherClass extends MyClass {

        public int getSomeValue( ) {

                int value = 0;
                // invoke getSomeValue in MyClass
                value = super.getSomeValue( );
                value = value + 1;
                return value;

        }

}
```

The same test class produces the following output; note the value is 43, not 42:

```
The value of someValue is: 43
```

The statement

```
value = super . getSomeValue( );
```

includes a call to **getSomeValue** 'on' an object with the reference 'super'. The use of the keyword **super** in this context refers to an object of the class **MyClass** from which **MyOtherClass** inherits its members and shows that we can invoke methods of an extended class in the extending class by referring to the former via the object reference *super*.

The relationship between **MyOtherClass** and **MyClass** is such that the latter is said to be the *superclass* of the former or, looking at the relationship the other way round, the former is a *subclass* of the latter. The IDE used to compile the three classes referred to in this section displays this relationship in the screen shot shown in the next figure.

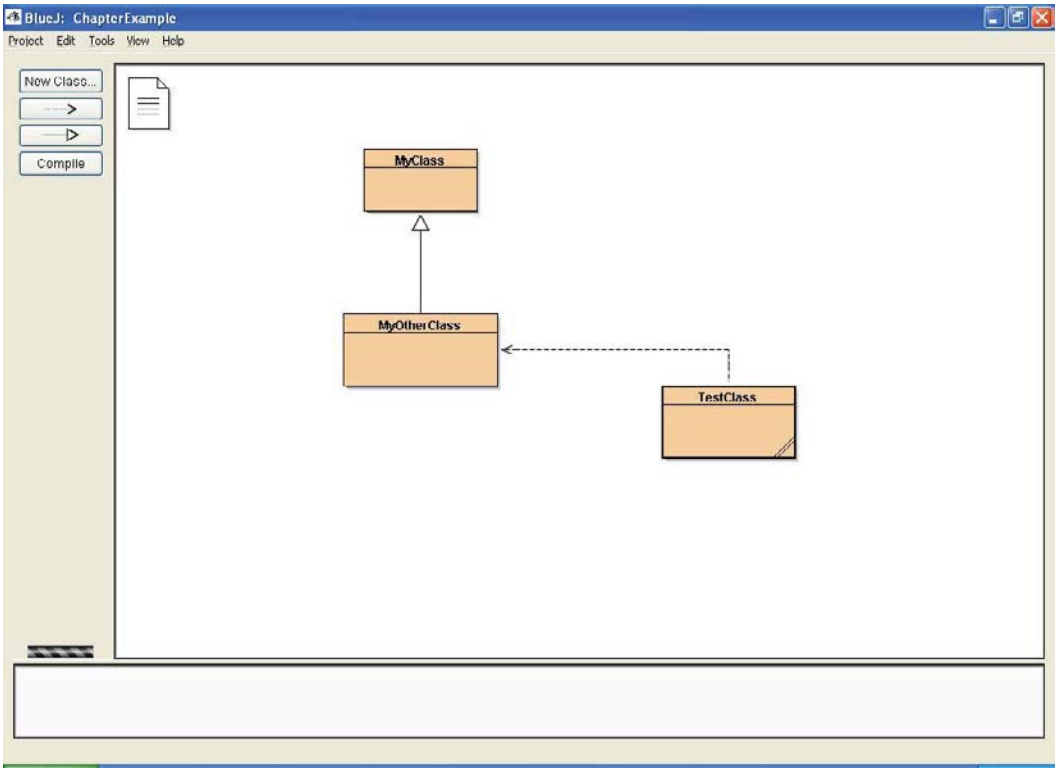


Figure 3.1 The relationship amongst classes used in Section 3.1

The solid line with the closed arrow indicates that **MyOtherClass** is a subclass of its superclass **MyClass**; in other words, **MyOtherClass** has an *'is a'* relationship with its superclass. The dotted line with the open arrow shows that **TestClass** has a *'has a'* relationship with **MyOtherClass** because **TestClass** declares a local variable of the **MyOtherClass** type in its **main** method. (The dotted line between **MyOtherClass** and **TestClass** should be a straight line according to the conventions of UML diagrams. It is a quirk of the IDE that it does not draw straight lines for 'has a' relationships.)

Download free eBooks at bookboon.com

The simple example discussed in this section shows the superclass-to-subclass relationship between two classes. In fact, *all* classes written in Java inherit *implicitly* from a class whose type is **`Object`** and, as a consequence, inherit the members of the class **`Object`**. The following extract from the API shows some of the members of the **`Object`** class.

java.lang

Class Object

`java.lang.Object`

`public class` **`Object`**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now

Go to www.helpmyassignment.co.uk for more info

Helpmyassignment

Download free eBooks at bookboon.com

Click on the ad to read more

Method Summary	
protected Object	**clone**() Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	**getClass**() Returns the runtime class of this Object.
int	**hashCode**() Returns a hash code value for the object.
String	**toString**() Returns a string representation of the object.
void	**wait**() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
	wait is overloaded; not shown here.

Thus, the class declaration for MyClass

public class MyClass

actually implies

public class MyClass extends Object

The *extends* clause is omitted because *every* Java object inherits from the class **Object**.

The next extract from the API shows that all classes provided by the Java development environment inherit from the class **Object**.

java.lang
Class Short
java.lang.Object
└ java.lang.Number
└ **java.lang.Short**

The extract indicates that the class **Short** is a subclass of **Number** which is, in turn, a subclass of **Object**.

Before we move on to explore aspects of inheritance in Java more closely, let us summarise the principal concepts introduced or implied by the discussion in this section.

- Inheritance is the ability to create new classes from existing ones;
- Java exhibits *single* inheritance; i.e. a subclass can have only one superclass;
- fields and methods are inherited in a subclass; new ones can be introduced;
- constructors are not inherited;
- a class called Object is at the top of the inheritance tree, as shown by the Java API;
- all Java classes implicitly inherit from Object;
- a subclass can modify methods inherited from its parent class; this is known as *overriding*; thus, an instance method with the same signature and return type as a method in the superclass is said to 'override' it;
- the keyword *super* is used to refer to the members of a superclass, i.e. variables, methods and constructors; thus, an overriding method can invoke the overridden method using the keyword *super*;
- a method invocation does not have to be on a method in the superclass; it can be to a method further up the class hierarchy.

Download free eBooks at bookboon.com

Click on the ad to read more

3.2 Overriding and Hiding Methods in a Subclass

Section 3.1 gives a simple example of overriding a method in a subclass. In essence, method overriding is a means by which inherited behaviour can be modified to suit the specific logic of a subclass, where this is derived from the general logic of its superclass. Thus, we can think of a superclass comprising members that are common to its subclasses. Common members are inherited and may or may not be modified in each subclass as required.

Consider, for example, the simple class hierarchy shown in the Figure 3.2 below.

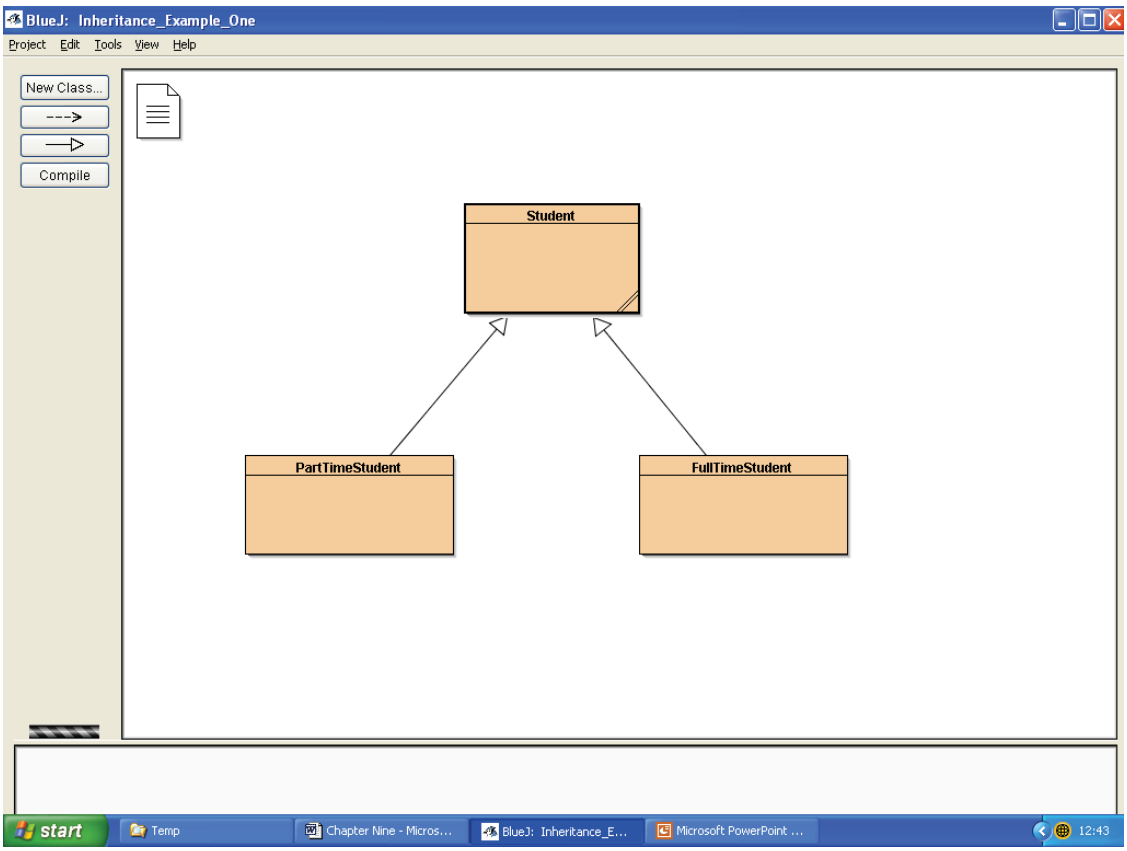


Figure 3.2 A superclass and two of its subclasses

The class definitions of the three classes follow on the next two pages: some of the documentation has been omitted for the sake of brevity.

```
public class Student {

    // Declare instance variables.
    private int idNumber;
    private String degreeCourseCode;
    private int yearOfEnrollment = 2008;

    public Student( int idNumber, String degreeCourseCode ) {
        // Intialise two of the instance variables.
```

```
            this.idNumber = idNumber;
            this.degreeCourseCode = degreeCourseCode;
        }

        public int getIdNumber( ) {
            return idNumber;
        }

        public String getDegreeCourseCode(  ) {
            return "course code " + degreeCourseCode;
        }

        public int getYearOfEnrollment( ) {
            return yearOfEnrollment;
        }

} // End of class definition.

public class FullTimeStudent extends Student {

        // Declare instance variables.
        private String fullName;

        public FullTimeStudent( int idNumber,
            String degreeCourseCode ) {

            // Call the constructor for the superclass.
            super( idNumber, degreeCourseCode );
        }


        // This method overrides the getDegreeCourseCode method in Student; it also calls
        // the overridden method.
        public String getDegreeCourseCode(  ) {
            return "This full-time student is enrolled on: "  +
                    super.getDegreeCourseCode( );
        }

} // End of class definition.

public class PartTimeStudent extends Student {

        // Declare instance variables.
        private String fullName;

        public PartTimeStudent(int idNumber,
```

```
                String degreeCourseCode) {
                // Call the constructor for the superclass.
                super( idNumber, degreeCourseCode );
        }

        // This method overrides the getDegreeCourseCode method in Student; it also calls
        // the overridden method.
        public String getDegreeCourseCode( ) {
                return "This part-time student is enrolled on: "
                        + super.getDegreeCourseCode( );
        }

} // End of class definition.
```

An examination of the source code illustrates how an instance method with the same signature and return type as a method in the superclass overrides it. Thus, the method **getDegreeCourseCode** in **PartTimeStudent** overrides **getDegreeCourseCode** in **Student**. The example also shows how the keyword *super* is used to invoke the overridden method in the body of the overriding method.

(One of the rules that govern method overriding states that an overriding method cannot throw different types of **Exception** objects than the overridden method. We will find out how exceptions are handled in the next chapter.)

Download free eBooks at bookboon.com

Click on the ad to read more

A *class* method (i.e. a *static* method) with the same signature as a class method in the superclass is said to "hide" it. For class methods, the runtime system invokes the method defined in the *compile-time* type of the reference on which the method is called; for instance methods, the run-time system invokes the method defined in the *run-time* type of the reference on which the method is called. This is illustrated by providing the following method in the **Student** class definition.

```
public static String getDetails( ) {
        return "I am a student.";
}
```

When the **main** method shown next is run

```
public class TestStudents {

        public static void main( String[ ] args ) {

                // Create a Student object.  The variable s is of the Student type, but refers to
                // a FullTimeStudent object (at run-time).
                Student s = new FullTimeStudent( 1234, "Java" );
                // Call the static method of Student.
                System.out.println( s.getDetails( ) );
                // Call an instance method of FullTimeStudent.
                System.out.println( s.getDegreeCourseCode( ) );
        }
}
```

the output is:

```
I am a student.
This full-time student is enrolled on: course code Java
```

and illustrates the difference between a method invocation on a compile-time type and a run-time type. There is one further rule to state concerning class methods at this point: *an instance method cannot override a static method and a static method cannot hide an instance method.*

3.3 Invoking a Parent Class Constructor from a Subclass Constructor

A number of examples in previous sections show how the keyword *super* is used to invoke an overridden method. The keyword *super* is also used to invoke a parent class constructor from a constructor of a subclass. In fact, the call to super(< parameter list >) *must* be the first statement of a subclass constructor, as illustrated above in the constructor of **PartTimeStudent** and **FullTimeSudent.**

Download free eBooks at bookboon.com

There are a number of rules that pertain to the call to super(< parameter list >):

- it is necessary to initialise all fields of a superclass; therefore its constructor must be called;
- a specific constructor is called as determined by the arguments that are passed to the call to *super*; this is illustrated by the first statement in the constructor for FullTimeStudent;
- if no call to *super* is used in a subclass constructor, the compiler adds an implicit call to super( ) that calls the parent, no argument constructor (which could be its default constructor);
- if the parent class define constructors but does not provide a no argument constructor, an error message is issued by the compiler if a call to super( ) is made from a subclass.

Given these rules, care should be taken when calling the correct superclass constructor from the first statement of a subclass constructor.

3.4 final and abstract Classes

Figure 3.3 shows an enhanced version of the simple class hierarchy shown in Figure 3.2.

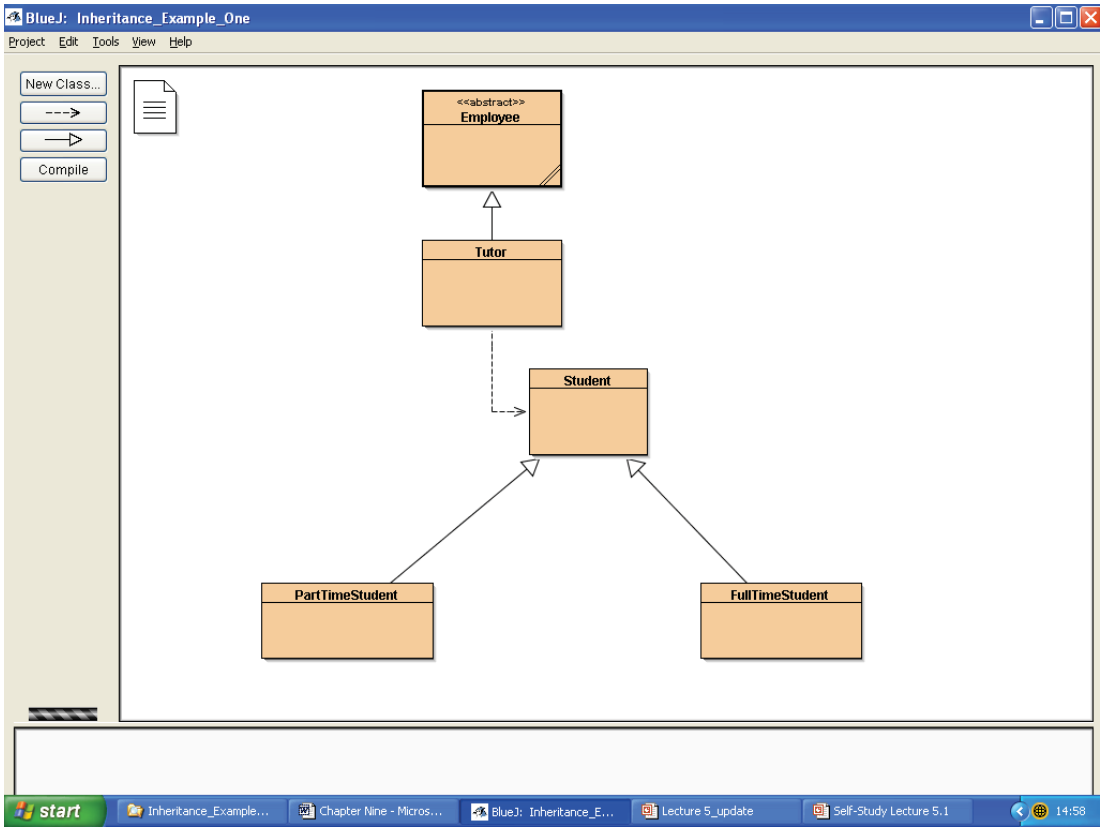


Figure 3.3 An enhanced version of Figure 3.2

The **Employee** class is labelled (by the IDE) as **abstract**. For the purposes of this simple example, the **Employee** class is defined to be **abstract** because we do not want to instantiate objects of the **Employee** class. Instead, we wish to instantiate objects of the **Tutor** class, in other words we wish to populate the application with *concrete* employees such as **Tutor**, rather than use the abstract and generalised notion of an employee. The class definitions for **Employee** is shown next.

```
public abstract class Employee {

    // Declare instance variables.
    String department;

    /**
    * This constructor initialises the variable with the identifier department.
     * @param department  The employee's department.
     */
    public Employee( String department ) {
            this.department = department;
    }

    // All subclasses of Employee must implement this abstract method.
    // Note how it is declared.
    public abstract String getDepartment( );

} // End of class definition.
```

Download free eBooks at bookboon.com

Click on the ad to read more

The class definition of **Tutor** is next.

```
public class Tutor extends Employee {

    // Declare instance variables.
    private String tutorsName;
    // An array to store references to students.
    private Student [ ] studentsInJavaGroup;

    /**
    * Constructor for objects of class Tutor.
    * @param department  The tutor's department.
    */
    public Tutor( String department ) {
        // Call the constructor of the superclass.
        super( department );
    }

    /**
    * Abstact method inherited from Employee; must be overridden.
    */
    public String getDepartment( ) {
        return department;
    }

} // End of class definition.
```

It should be noted that an **abstract** class may declare **abstract** methods. For example, the **abstract** class **Employee** declares an **abstract** method **getDepartment** that must be overridden in sublclasses of **Employee**.

The aim of the example shown in Figure 3.3 is to illustrate some of the following rules concerning **abstract** classes:

- the complier prevents an abstract class from being directly instantiated, though it usually has constructors that are called from the constructor of its subclasses;
- an abstract class may have abstract methods:
  - such methods contain no implementation;
  - non-abstract subclasses must override these methods and implement them;
  - if all methods are abstract, the class should be an *interface*; (we will find out what a Java interface is in Chapter Five);
- any class with one or more abstract methods is called an abstract class;
- abstract classes can have data attributes, concrete methods and constructors.

Download free eBooks at bookboon.com

At this point, it is useful to note that there is another category of class known as a **final** class. A **final** class cannot be subclassed and a **final** method cannot be overridden.

3.5 What Does Type Compatibility Mean?

Now that we have explored some of the essential concepts associated with inheritance, we can address the deferred discussion about conversion of type variables from Chapter Three (An Introduction to Java Programming 3: The Fundamentals of Objects and Classes).

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

Checking for type compatibility is carried out when an expression is assigned to a type variable as follows:

SomeClass sc = < expression that returns an object reference >;

The compiler will regard the types as compatible when one of three conditions applies to the expression:

1. the expression returns an object reference of the SomeClass type;
2. the expression returns an object reference to a subclass of SomeClass;
3. the expression returns an object reference to an object that implements the SomeClass interface. (Java interfaces are explored in Chapter Five.)

We can use the class hierarchy shown in Figure 3.3 to illustrate the first two conditions.

Consider the following statement:

Student s1 = new Student(111, "Java");

This statement compiles because it complies with the first condition.

Consider the next statement:

Student s2 = new PartTimeStudent(222, "Java");

This statement compiles because it complies with the second condition.

It is worthwhile dwelling on the general nature of the second condition:

SuperClass sc = new SubClass( );

Download free eBooks at bookboon.com

Up to this point in the guide, we have created objects of a particular class by calling one of the constructors of that class. However as the statement shows, this condition of compatibility is permitted in Java. In other words, while the class of an existing object doesn't change during its lifetime, it can be referenced by a variable of either its own type or of its superclass type.

Therefore, we would expect the next statement to compile:

Student s3 = new FullTimeStudent(333, "Java");

In the statement above we are, in effect, converting between a subclass type and its superclass type when we make the association of a **FullTimeStudent** to a **Student**.

Types higher up a hierarchy are said to be *wider* than the *narrower* types lower down the hierarchy. In the statement

Student s3 = new FullTimeStudent(444, "Java");

the widening conversion, or *upcast*, is carried out automatically by the compiler. The implicit safe cast to a **Student** object is valid at run-time when it can be shown that **s3** refers to a **FullTimeStudent** object.

On the other hand, consider the next statement:

PartTimeStudent pts = new Student(555, "Java");

Download free eBooks at bookboon.com

Click on the ad to read more

The compiler issues the following message:

incompatible types: found Student expected PartTimeStudent.

A *narrowing conversion* or *downcast* in this statement will satisfy the compiler, as follows:

PartTimeStudent pts = (PartTimeStudent) new Student(555, “Java”);

If this kind of cast survives a compile-time check, a second check occurs at run-time to determine whether the class of the object being cast is compatible with the object reference. In other words, a downcast may not be a safe cast in the sense that it may not be valid at run-time.

When the statement

PartTimeStudent pts = (PartTimeStudent) new Student(555, “Java”);

is included in a **main** method and **main** is run, a run-time **Exception** occurs:

java.lang.ClassCastException: Student cannot be cast to PartTimeStudent

and shows that this kind of cast *is* invalid at run-time.

In the context of the class hierarchy shown in Figure 3.3, the run-time rules imply that **Student** objects cannot be cast to **PartTimeStudent** or **FullTimeStudent** objects, but that **PartTimeStudent** and **FullTimeStudent** objects can be cast to **Student** objects. In other words, all **PartTimeStudent** and **FulTimeStudent** objects are **Student** objects in that the former inherit from the latter, but all **Student** objects are not necessarily FullTimeStudent or **PartTimeStudent objects**: a **Student** object may be a **Student** object, as in the next statement:

Student s = new Student(999, “Java”);

While the type of an object reference may be obvious at compile-time, the actual class of the object referenced in memory may be less obvious or may not be known until run-time. For example, consider the following statement taken from the **MediaStore** class of the themed application.

Member[ ] members = someStream.readObject( );

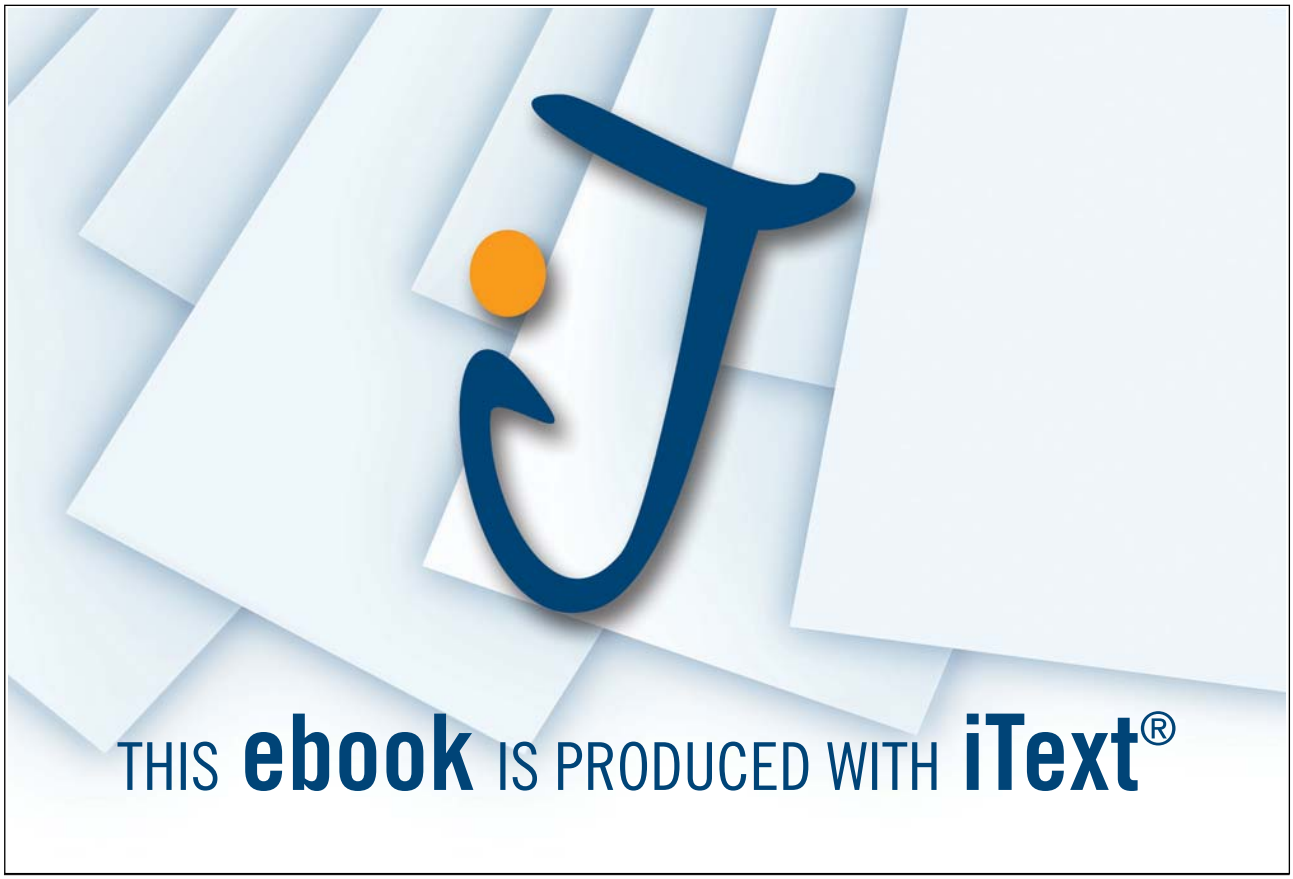
The purpose of the input stream **someStream** is to read the array of existing members of the Media Store from a file into the application by calling the stream’s **readObject** method. (We will examine some of the stream classes in Chapter One in *An Introduction to Java Programming 3: Graphical User Interfaces*). The statement does not compile because the API states that the **readObject** method returns an object of the **Object** type. Therefore, a cast is required, as follows:

Member[ ] members = (Member[ ])someStream.readObject( );

Download free eBooks at bookboon.com

On the face of it, (down)casting an **Object** object to an object of the Member[ ] type may not be valid at run-time. In this case, however, the actual object stored in memory *is* of the **Member[ ]** type; therefore, run-time compatibility is preserved.

Finally in this section, it is worth reminding the learner that the **instanceof** operator can be used to find out the type of an object held in memory, so that a cast can be used to restore full functionality to the object. For example, one of the test classes of the object hierarchy shown in Figure 3.3 includes the method shown on the next page.



Download free eBooks at bookboon.com

Click on the ad to read more

```
public static void getDetails( Student student ) {
        // find out the type of the object passed when the method is called
        if ( student instanceof PartTimeStudent )
        {
                // note the cast
                PartTimeStudent pt = ( PartTimeStudent )student;
                // call an inherited method
                System.out.println( "This part-time student enrolled in "
                        + pt.getYearOfEnrollment( ) );
        }
        else if ( student instanceof FullTimeStudent )
        {
                FullTimeStudent pt = ( FullTimeStudent )student;
                System.out.println( "This full-time student enrolled in "
                        + pt.getYearOfEnrollment( ) );
        }
        else if ( student instanceof Student )
        {
                System.out.print( "This student enrolled in "
                        + student.getYearOfEnrollment( ) );
        }
}
```

The reader should note how the **instanceof** operator and casting is used, in the method shown above, to find out the type of **Student** object passed to it as an argument so that it can be processed accordingly.

3.6 Virtual Method Invocation

The rules of compatibility discussed in Section 3.5, raise a question when invoking methods: *how do we know which object is being used when invoking a method?*

To illustrate the answer to this question, consider the following code snippet from a test class of the simple application shown in Figure 3.3.

```
Student student = new PartTimeStudent( 1234, "Java" );
System.out.println( student . getDegreeCourseCode( ) );
```

The output is:

```
This part-time student is enrolled on: course code Java
```

The output shows that when you invoke a method via an object reference, it is the *run-time type* of the object referred to which governs which implementation is used. Thus, in the statement above, the object reference **student** refers to a **PartTimeStudent** object and the call to

Download free eBooks at bookboon.com

```
student . getDegreeCourseCode( );
```

invokes the **getDegreeCourseCode** method implemented in the **PartTimeStudent** class definition and not that in the **Student** class definition.

For class methods, on the other hand, the run-time system invokes the method defined in the *compile-time type* of the reference on which the method is called. Thus, a call to the static method **getDetails** as follows

```
student . getDetails( );
```

invokes the **getDetails** method of the **Student** class and not that of **PartTimeStudent**, as shown in Section 3.2 above.

3.7 Controlling Access to the Members of a Class

Access modifiers are used to determine whether other classes have access to a member of a class. Up to this point in the guide, we have met the access modifiers *public* and *private* as they are applied to modify declarations of fields and methods of a class. The access modifier *public* means that all other classes have access to such members of a class and the access modifier *private* means that other members of a class have access to private members of that class.

Extended classes give us an opportunity to explain a further access modifier: that of *protected*. The access levels for the access modifiers *public*, *private* and *protected* are summarised in Table 3.1 shown on the next page.

Download free eBooks at bookboon.com

Click on the ad to read more

Modifier	Same Class	Same Package	Subclass	Universe
`private`	Yes			
default: no specifier	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

Table 3.1 Access levels

The first row confirms what we know about the access modifier *private*: i.e. the class itself has access to other private members, as we would expect.

The second row indicates that if no modifier is specified, classes in the same *package* have access to such members. We will encounter packages in Chapter Six. Until then, suffice it to say for the present purposes that a package is a convenient way to group together a number of related classes to provide namespace management.

The fourth row shows that *all* classes have access to public members, regardless of their package and parentage.

The third row indicates the level of access provided when a class member is declared to be *protected*. The first column indicates that other members of the class itself have access to the protected member of that class; the second column indicates that classes in the same package, regardless of their parentage, have access to the protected member of the class; the third column indicates that subclasses of the class have access to the protected member, regardless of what package they are in. However, the subclass-protected table entry has an interesting twist that we will defer until Chapter Six.

The example code that follows on the next page illustrates the ‘rules’ encapsulated in Table 3.1, but ignores the second column for the time being.

Download free eBooks at bookboon.com

Example One

```
public class MySuperClass {

        private int privateInt;
        protected int protectedInt;

        public void aMethod( ) {

                System.out.println( privateInt );
                System.out.println( protectedInt );
        }

}
```

The class compiles and merely shows that one of the members of the class – **aMethod** – has access to the private and protected variables of the class.

Example Two

```
public class MySubClass extends MySuperClass {

        public void aMethod( ) {

                // System.out.println( privateInt );
                // the statement above is illegal: privateInt has private access
                // in MySuperClass
                System.out.println( protectedInt );
        }

}
```

The class compiles and shows that the subclass does not have access to the private variable of the superclass but that it *does* have access to the protected variable of the superclass.

Whilst on the face of it, the access level known as *protected* might be regarded as implying a high degree of 'protection' from other classes, the table and example code above shows that this is not the case. The class **MySubClass** shows that all subclasses of **MySuperClass** have access to protected members of **MySuperClass**. Thus we can see that the *protected* access level is not as protected as *private*. Nevertheless, it may be the case that the developer wishes to make a number of members of a superclass protected in order to provide easy access to them from subclasses.

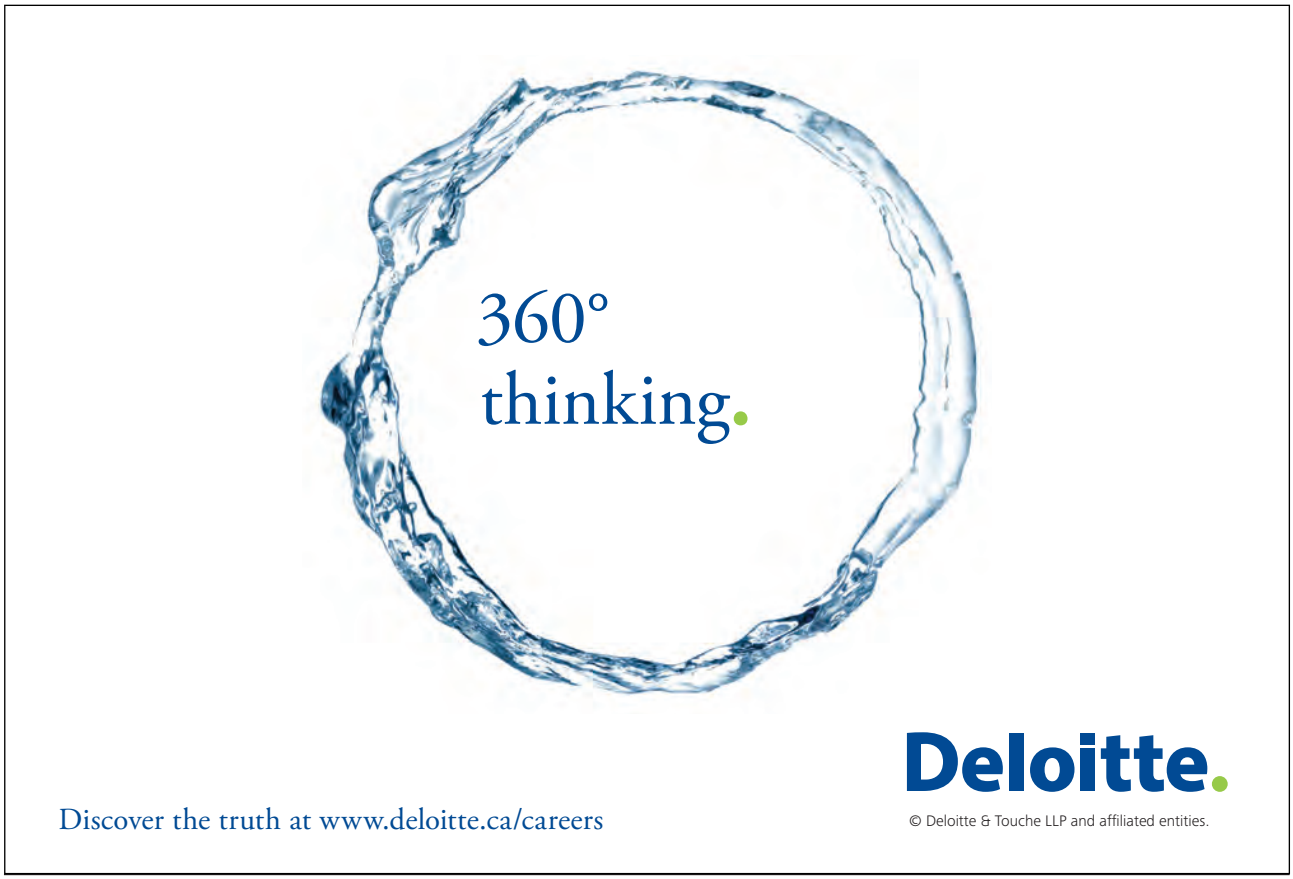
Download free eBooks at bookboon.com

3.8 Summary of Inheritance

Although it is not made explicit at the beginning of this chapter, the examples used aim to illustrate *two* forms of inheritance. Rather than merely mention them both at the outset, it is to be hoped that both forms emerge from the explanations and code examples used in the chapter.

The previous sections aim to show how a class can be extended or subclassed and that a subclass can be used in code designed to work with the superclass. For example, **FullTimeStudent** objects can be used by code designed to work with **Student** objects. If a method expects a parameter of the type **Student**, you can pass it a **FullTimeStudent** object and it will work, although you are likely to have to find out the type of the argument – by using the **instanceof** operator – and restore the full functionality of the object by a suitable cast, as shown in Section 3.5. This feature of object references is known as *polymorphism*. An object that is declared to be of the **Student** type can have many forms in that it can be used as a **Student** object, a **FullTimeStudent** object, or a **PartTimeStudent** object.

The behaviour of a **Student** object is inherited by a **FullTimeStudent** object: the latter is said to *extend* the former. Extended behaviour can be entirely new – by means of adding new methods to the subclass – or it can modify inherited behaviour by overriding a method with the same signature and return type as the overridden method. Thus an extended class can override the behaviour of its superclass by providing new implementations of one or more of the inherited methods.



Download free eBooks at bookboon.com

Click on the ad to read more

Overall, extending classes gives rise to two forms of inheritance:

1. *inheritance of type*, where the subclass acquires the type of the superclass so that it can be used polymorphically in code designed to work with the superclass type;
2. *inheritance of implementation*, where the subclass acquires the implementation of the superclass in terms of its members.

Inheritance is a fundamental concept in OOP languages; as a consequence, further examples could have been provided in this chapter to illustrate further the essential features of extending classes. At this stage, however, it is probably wise to conclude this chapter and leave it to the leaner to work with extended classes in practice and encounter concepts in practical situations.

The next chapter explains how errors are handled in Java programmes.

Download free eBooks at bookboon.com

4. Errors in Java Programmes

With the best will in the world, errors will occur when developing applications irrespective of the target language: Java programmes are no exception. However, object-oriented programming (OOP) languages such as Java offer a distinct advantage over non-OOP languages in that the former are able to respond to certain error conditions by creating an *object* to represent the error. This approach leaves the developer with the responsibility of processing error objects in order to respond to errors occurring when an application is running.

This chapter explains how errors are handled in Java programmes; it does not include syntax errors. Syntax errors are caused by incorrect use of the Java language on the part of the developer; the compiler checks for this category of error and notifies the developer accordingly.

4.1 Categories of Error

A search of the previous nine chapters reveals the following error messages in the discussion of examples:

ArrayIndexOutOfBoundsException

and

ClassCastException

These two messages imply that something happened at run-time when the example programmes were executed. In fact, both of these error messages imply that the developer has made a logic error. Logic errors that occur at run-time are usually reflected in the output from the application. Logic errors are eliminated by further testing and debugging of the programme at compile time.

At compile time, in the first case, the compiler could not be expected to anticipate that an out of bounds array index is being processed at runtime. The occurrence of an **ArrayIndexOutOfBoundsException** should be sufficient information for the developer to check the logic of the code that produces such an error.

At compile time, in the second, case, the developer has not obeyed the rules for casting object references. Although, the code has compiled because the compile-time rules have been obeyed, the run-time rules have been broken; again, the nature of the error message should be sufficient information for the developer to check that the statements that include a cast obey both the compile-time and run-time rules of object reference casting as explained in the previous chapter.

In short, the two errors indicated by the messages

ArrayIndexOutOfBoundsException

and

ClassCastException

Download free eBooks at bookboon.com

should have been fixed by the developer at compile time; they cannot be recovered from at run-time. Given that they were not anticipated by the developer, they are relatively easily eliminated by working through and correcting the logic of the code that produced the error messages. Working through the logic of source code is known as *debugging*. Integrated Development Environments (IDEs) usually provide a debugging tool that can be used to step through code statement by statement.

There is, on the other hand, a number of other run-time errors that can occur when a programme executes that are outside the control of the programme's logic. These include, for example, error conditions that are reasonably likely to occur in that they have the potential to impair access to data and access to other local and networked resources. The kind of error that is reasonably likely to occur at run-time should be recoverable so that the programme does not terminate abnormally.

Anticipating that such errors have the potential to occur does not include *expected* conditions such as, for example, detecting the end of a file that is being read by a method. In this case, the method that reads the file should include code that detects the end of the file so that an 'end of file' error does not occur at run-time.

The inevitable existence of run-time errors raises an important question: *how does the developer anticipate run-time error conditions?* If the developer takes an exhaustive position and anticipates that all error conditions have the potential to occur with all methods, the resulting code is highly likely to be cumbersome to write and almost unreadable. On the other hand, if the developer takes an optimistic position and doesn't anticipate many error conditions, the resulting code may not be robust enough when the application is released to its users. The answer to the question (posed at the beginning of this paragraph) is not a straightforward one in that it is not easy to decide which errors to anticipate and which not to anticipate. In practice, the answer lies in arriving at a reasonable practical compromise between the two positions.

4.2 What are Unexpected Error Conditions?

The two error messages listed in Section 4.1 reveal a clue to how unexpected error conditions are dealt with in a Java programme. A cursory examination of the messages

ArrayIndexOutOfBoundsException

and

ClassCastException

reveals that the two compound words have something in common: the word 'Exception'. It is to be hoped that the reader immediately recognises that an **Exception** is a Java class. Therefore an **ArrayOutOfBoundException** is a type of **Exception** class. This is indeed the case as the following extract from the API confirms.

Download free eBooks at bookboon.com

java.lang

Class ArrayIndexOutOfBoundsException

java.lang.Object
└ java.lang.Throwable
└ java.lang.Exception
└ java.lang.RuntimeException
└ java.lang.IndexOutOfBoundsException
└ **java.lang.ArrayIndexOutOfBoundsException**

Exception objects (usually shortened to *exceptions*) that are subclasses of the **RunTimeException** class, as in the case of an **ArrayOutOfBoundException**, usually arise as a result of logic errors and are the responsibility of the developer to eliminate at compile time. This kind of exception is known as an *unchecked exception*.

The discussion of the two error messages

ArrayIndexOutOfBoundsException

and

ClassCastException

implies that they are both examples of unchecked exceptions.

SIMPLY CLEVER
ŠKODA

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.
Send us your CV on
www.employerforlife.com

Click on the ad to read more

Exceptions that are reasonably recoverable at run-time and that are not the result of logic errors are not unchecked exceptions; this type of exception is incorporated into the implementation of methods that give rise to them, as shown in the next section.

4.3 Checked Exceptions

Exceptions provide a straightforward mechanism to check for errors without cluttering code with additional statements such as **if .. else** constructs and other statements to test the value of fields in order to detect when an error condition might arise. The exceptions that a method might produce are explicitly included as part of the method's declaration. This means that exceptions are as important a part of a method's programming interface as are its return type and parameters. The inclusion of exceptions in the declaration of a method means that they are made known to the code that invokes the method and, as a result, the compiler knows about them. The type of exception that is checked by the compiler is known as a *checked exception.*

When an error occurs when a method is invoked, the exception object is passed to the run-time system; this process is known as *throwing an exception.* The run-time system tries to find some code in the calling method that is designed to respond and *handle* the error. If this handling code cannot be found in the calling method, the run-time system works its way through the set of methods that has been called to call the method that throws the exception until it finds some handling code. This set of methods is known as the *call stack*. If handling code has not been provided by the developer, the run-time system eventually arrives at the end of the method invocation stack to the thread that runs the application's **main** method. If **main** does not handle the exception, **main**'s thread of execution will terminate abnormally. In other words, the application will 'crash' when **main** terminates in an abnormal way. (We will examine threads in Chapter Four in *An Introduction to Java Programming 3: Graphical User Interfaces*.)

On the other hand, if handling code *has* been provided by the developer somewhere in the call stack, the exception is said to be *caught* by the block of code that is the handler.

It will be instructive, at this point in the discussion of exceptions, for the learner to study an example that illustrates how checked exceptions are thrown and caught.

4.3.1 How is a Checked Exception Handled in a Java Programme?

The example that follows shows how an exception is *thrown and caught* in order to illustrate the key concepts associated with exception handling in a Java programme.

The code for a simple exception class follows on the next page.

Download free eBooks at bookboon.com

```
public class MyException extends Exception {

    // constructor
    public MyException( ) {

        super( "You are attempting to divide by zero." );

    }

} // end of class definition
```

The constructor for **MyException** passes a **String** to the superclass **Exception**; this **String** is used to construct an error message.

Strictly speaking, an attempt to divide a number by zero throws an instance of an **ArithmeticException**, which is type of **RunTimeException**: i.e. it is an unchecked exception. However making **MyException** a checked exception by inheriting directly from **Exception**, means that the example can be used to illustrate how a checked exception is thrown and caught.

The class **MyObject** includes a method that declares that it throws **MyException** objects: note the use of the keyword 'throws' in the declaration of **quotient**. The class definition is next.

```
public class MyObject {

    public double quotient( int num, int den ) throws MyException {

        if( den == 0 )
        {
            System.out.println( "quotient has exited and thrown an " +
                "instance of MyException." );
            throw new MyException( );
        }
        else
        {
            System.out.println( "quotient has completed." );
            return ( double ) num / den;
        }

    } // end of quotient

    public void myMethod( int num, int den ) {

        try
        {
            double answer = quotient( num, den );
```

```
                System.out.println( "The value of num / den = " + answer );
        }
        catch( MyException me )
        {
                me.printStackTrace( );
        }
        System.out.println( "myMethod has completed." );

    } // end of myMethod

} // end of class definition
```

The method **quotient** includes an **if .. else** construct that determines the condition when an instance of **MyException** is thrown. If the denominator (**den**) is not zero, the method returns the **double** value of the numerator (**num**) divided by the denominator (**den**).

The method **myMethod** invokes the method **quotient** and checks for the exception using what is known as a **try ... catch** construct. The **try** block includes the call to **quotient**. If **den** is not equal to zero, the statements of the **try** block execute and the **catch** block is skipped and the message "myMethod has completed." is output.

If **den** *is* equal to zero when **quotient** is called by **myMethod**, the method **quotient** will stop its execution and the remaining statements in the **try** block are skipped and the **catch** block is executed. The statement in the **catch** block calls an inherited method of **MyException** in order to output useful information about the error condition. In short, the exception is caught in **myMethod** when it is thrown by the call to **quotient**. The print statements in both methods in **MyObject** serve to show where processing has reached when an exception is thrown.

In order to illustrate what happens when **myMethod** is called, a test class is required. The code for the test class follows on the next page.

Download free eBooks at bookboon.com

```
public class TestClassOne {

    public static void main ( String[ ] args ) {

        // use EasyInput to input two int values; -99 escapes the loop
        int den = 0;
        while( den != -99 )
        { // start of loop
            EasyInput keyboard = new EasyInput( );
            System.out.println( "Entering -99 for the second number " +
                "will terminate main." );
            System.out.print( "Enter the first number: ");
            int num = keyboard.nextInt( );
            System.out.print( "Enter the second number: ");
            den = keyboard.nextInt( );

            // test the value of den
            if( den == -99 )
            {
                break;
            }
            else // carry on
```

Click on the ad to read more

```
                {
                        // instantiate a MyObject object
                        MyObject mo = new MyObject( );
                        mo.myMethod( num, den );
                        System.out.println( "main is still running." );
                } // end else
        } // end of loop
        System.out.println( "main has terminated." );

    } // end of main

} // end of class definition
```

TestClassOne uses an instance of a class called **EasyInput** (the code for which is not shown); **EasyInput** provides a number of methods to capture data entered via a computer's keyboard. The **main** method in the test class above includes a simple loop and exit strategy.

Firstly, let us find out what happens when we don't attempt to divide by zero. The output is as follows and is what is expected, given the source code shown above.

```
Entering -99 for the second number will terminate main.
Enter the first number: 1
Enter the second number: 2
quotient has completed.
The value of num / den = 0.5
myMethod has completed.
main is still running.
Entering -99 for the second number will terminate main.
Enter the first number: 1
Enter the second number: -99
main has terminated.
```

Next, let us find out what happens when we attempt to divide by zero. The output is as follows.

```
Entering -99 for the second number will terminate main.
Enter the first number: 1
Enter the second number: 4
quotient has completed.
The value of num / den = 0.25
myMethod has completed.
main is still running.
Entering -99 for the second number will terminate main.
Enter the first number: 2
Enter the second number: 0
quotient has exited and thrown an instance of MyException.
myMethod has completed.
```

Download free eBooks at bookboon.com

```
main is still running.
Entering -99 for the second number will terminate main.
Enter the first number:

MyException: You are attempting to divide by zero.
at MyObject.quotient(MyObject.java:9)
at MyObject.myMethod(MyObject.java:23)
at TestClassOne.main(TestClassOne.java:27)
```

(There are other methods in the call stack that call **main** that are specific to the IDE [BlueJ] that was used to run the example. However, these are omitted for the sake of clarity.)

The output is what is expected: **quotient** throws an exception and exits; the exception is caught by the **catch** block of **myMethod** and outputs the stack trace. However, **main** is still running and shows that the exception has been recovered at run-time.

Next, let us find out what happens when **myMethod** *doesn't* catch **MyException** objects but declares that it throws them.

The code for **MyObject** is now as follows.

```
public class MyObject {

    public double quotient( int num, int den ) throws MyException {

        if( den == 0 )
        {
            System.out.println( "quotient has exited and thrown an " +
                "instance of MyException." );
            throw new MyException( );
        }
        else
        {
            System.out.println( "quotient has completed." );
            return ( double ) num / den;
        }

    }
```

```
    public void myMethod( int num, int den ) throws MyException {

        double answer = quotient( num, den );
        System.out.println( "The value of num / den = " + answer );
        System.out.println( "myMethod has completed." );

    }

} // end of class definition
```

It is now the responsibility of **main**, as shown by the call stack, to catch **MyException** objects. The relevant section of the amended test class follows on the next page.

Download free eBooks at bookboon.com

Click on the ad to read more

```
// test the value of den
if( den == -99 )
{
        break;
}
else // carry on
{
        // instantiate a MyObject object
        MyObject mo = new MyObject( );
        try
        {
                mo.myMethod( num, den );
                System.out.println( "main is still running." );
        }
        catch( MyException me )
        {
                me.printStackTrace( );
                System.out.println( "main is still running." );
        }
```

The output is as follows.

```
Entering -99 for the second number will terminate main.
Enter the first number: 12
Enter the second number: 0
quotient has exited and thrown an instance of MyException.
main is still running.
Entering -99 for the second number will terminate main.
Enter the first number:

MyException: You are attempting to divide by zero.
at MyObject.quotient(MyObject.java:9)
at MyObject.myMethod(MyObject.java:21)
at TestClassOne.main(TestClassOne.java:29)
```

Again, the output is what is expected.

Finally, in this section, let us see what happens it **main** does not catch **MyException** objects but declares that it throws them. The amended declaration for **main** is as follows:

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

The body of **main** does not attempt to catch **MyException** objects.

Download free eBooks at bookboon.com

The output is as follows:

```
Entering -99 for the second number will terminate main.
Enter the first number: 9
Enter the second number: 0
quotient has exited and thrown an instance of MyException.

MyException: You are attempting to divide by zero.
at MyObject.quotient(MyObject.java:9)
at MyObject.myMethod(MyObject.java:21)
at TestClassOne.main(TestClassOne.java:27)
```

The IDE used to run the application indicates that **main** has terminated abnormally – in other words the programme has crashed – because **main** has not caught the exception and has merely declared that it is thrown. This illustrates that exceptions must be either caught or declared to be thrown by methods in the call stack and that **main** is the final opportunity to catch them.

> As a general rule, it is good practice to catch an exception when the method that throws it is called.

The example code and output explained in this section shows that the run-time system searches the call stack in the reverse order in which methods are called until it finds a **catch** block that is designed to respond to the exception thrown by a method. When a **catch** block is found, the exception is handed to it by the run-time system. If the exception is caught before it reaches **main** or if **main** catches it, the programme will not terminate abnormally; if **main** does not handle it but declares that it is thrown, **main** will crash.

Now that we have thoroughly explored an example and shown how a simple exception is thrown and caught in different places in the call stack, we are in a position to make some more comments about handling exceptions in the sections that follow.

Download free eBooks at bookboon.com

4.4 try … catch … finally Blocks

The code and output associated with the examples discussed in the previous section provide practical evidence that enables us to bring together a number of points concerning handling exceptions.

- Developer-written code that might throw an exception is enclosed in a **try** block.
- Similarly, method invocations that are defined to throw exceptions as indicated by the API for a method should be enclosed in a **try** block.
- A **try** block is followed by one or more **catch** blocks.
- Each **catch** block specifies the type of exception it catches (i.e. handles) and contains a handler for that exception type.
- After the last **catch** block, an optional **finally** block contains code that *always* executes.
- When an exception occurs, **catch** blocks are searched in their order for the appropriate handler.
- It is usual to sequence **catch** blocks from the specific to the general, i.e. objects of the **Exception** class are caught in the last **catch** block, which serves as a catch-all if any specific exceptions have not been caught.
- If an exception is not handled in a **try…catch** block, it is thrown to the next method in the call stack.
- If the exception is passed to the **main** method and is not handled there, the program terminates abnormally.

When non-memory resources such as files and I/O Streams – see Chapter One in *An Introduction to Java Programming 3: Graphical User Interfaces* - are used in a programme, they must eventually be released independently of Garbage Collection (of memory
resources such as identifiers and object references). The use of the **finally** block that follows a **try … catch** block is a good opportunity to release such resources. The general syntax of a **try … catch … finally** construct is as follows.

```
try
{
   // statements that invoke methods that throw Exceptions
   // statements that acquire resources
}
catch( AKindOfException ex1 )
{
   // exception handling statements for ex1
}
catch( AnotherKindOfException ex2 )
{
   // exception handling statements for ex2
}
```

```
catch( Exception e )
{
   // exception handling statements for e
}
finally
{
    // resource-release statements
}
```

The code for **myMethod** in the class definition for **MyObject** is modified to illustrate the use of a **finally** block; it doesn't release any resources, but the output of the programme shows that it is always executed.

```
public void myMethod( int num, int den ) {

        try
        {
                double answer = quotient( num, den );
                System.out.println( "The value of num / den = " + answer );
        }
        catch( MyException me )
        {
                me.printStackTrace( );
        }
        catch( Exception e )
        {
                e.printStackTrace( );
        }
        finally
        {
                System.out.println( "finally block: myMethod has completed." );
        }

} // end of myMethod
```

The output is shown on the next page.

Download free eBooks at bookboon.com

```
Entering -99 for the second number will terminate main.
Enter the first number: 1
Enter the second number: 2
quotient has completed.
The value of num / den = 0.5
finally block: myMethod has completed.
main is still running.
Entering -99 for the second number will terminate main.
Enter the first number: 1
Enter the second number: 0
quotient has exited and thrown an instance of MyException.
finally block: myMethod has completed.
main is still running.
Entering -99 for the second number will terminate main.
Enter the first number:

MyException: You are attempting to divide by zero.
at MyObject.quotient(MyObject.java:9)
at MyObject.myMethod(MyObject.java:23)
at TestClassOne.main(TestClassOne.java:27)
```

4.5 Throwing Exceptions

The example in Section 4.3.1 shows that an exception is thrown by a method by using the following generalised syntax:

```
public void aMethod( ) throws AnException {

        if ( <some condition> )
        {
                throw new AnException( );
        }
        else { }
        }

} // end of method implementation
```

and is caught by the calling method as shown on the next page.

Download free eBooks at bookboon.com

```
// a method that calls aMethod
try {
        objectRef.aMethod( );
}
catch ( AnException ae ) {
        // do something about the Exception
}
```

The **catch** block can *re-throw* the exception to its calling method, as follows:

```
// a method that calls aMethod
try {
        objectRef.aMethod( );
}
catch ( AnException ae ) {
        throw ae;
}
```

Download free eBooks at bookboon.com

Click on the ad to read more

Returning to the example, the method **myMethod** could be re-written as shown next.

```
public void myMethod( int num, int den ) throws MyException {

        try
        {
                double answer = quotient( num, den );
                System.out.println( "The value of num / den = " + answer );
        }
        catch( MyException me )
        {
                // re-throw object me to the next method in the call stack
                throw me;
        }
        finally
        {
                System.out.println( "finally block: myMethod has completed." );
        }

} // end of myMethod
```

In this case, it will be the responsibility of **main** to catch exceptions of the **MyException** type.

A **catch** block can throw a different type of **Exception**, as follows.

```
// code that calls aMethod
try {
        objectRef.aMethod( );
}
catch ( AnException ae ) {
        throw new AnotherException( );
}
```

Referring to example again, the method **myMethod** could be re-written as shown next.

```
public void myMethod( int num, int den ) throws SomeOtherException {

        try
        {
                double answer = quotient( num, den );
                System.out.println( "The value of num / den = " + answer );
        }
        catch( MyException me )
        {
                throw new SomeOtherException( );
        }
```

```
        finally
        {
                System.out.println( "finally block: myMethod has completed." );
        }

} // end of myMethod
```

In this case, it will be the responsibility of **main** to catch exceptions of the **SomeOtherException** type.

4.6 Exceptions in the Themed Application

The themed application includes a developer-defined exception that indicates when a member of the Media Store has exceeded their allowance of DVDs on loan against their virtual DVD membership card.

The source code for the class definition is shown on the following page.

Download free eBooks at bookboon.com

Click on the ad to read more

```
/**
* Class ItemLimitException detects transactions that exceed the limit set for a member's DVD
* card.
* @author D. M. Etheridge.
* @version 1.0, dated 6 December 2008.
*/
public class ItemLimitException extends Exception {

        // Declare instance variables.
        private int overLimit;

        /**
        * Constructor for objects of class ItemLimitException.
        */
        public ItemLimitException( String message, int overLimit ) {

                super( message );
                this.overLimit = overLimit;

        } // End of constructor.

        /**
        * This method overrides getMessage( ).
        *
        * @return a String message that includes the value of the overLimit attribute.
        */
        public String getMessage( ) {

                return super.getMessage( ) + overLimit;

        } // End of getMessage.

} // End of class ItemLimitException.
```

The **takeItemOnLoan** method of the **DvdMembershipCard** class throws this exception, as shown by the code that follows on the next page.

Download free eBooks at bookboon.com

```
/**
* This method takes a DVD on loan.  It overrides the method in the parent class.
*
* @param catNo  The catalogue number of the DVD taken on loan.
*/
public void takeItemOnLoan( String catNo ) throws ItemLimitException {

        // Find out if the DVD exists before attempting a transaction.
        // Note: the instance variable dvd and the method findDvd are members of
        // the same class as this method.
        dvd = findDvd( catNo );
        if ( dvd == null )
        {
                System.out.println( "No such dvd; please try again." );
        }
        else // Carry out the transaction.
        {
                if ( noOnLoan + 1 <= maxOnLoan )
                {
                        System.out.println( "You are taking 1 DVD on loan." );
                        System.out.println( "This transaction is acceptable and " +
                                "increases the number\n of DVDs that you have " +
                                "on loan by 1." );
                        noOnLoan = noOnLoan + 1;
                        dvdsOnLoan[ noOnLoan - 1 ] = dvd;
                        System.out.println( "You are taking " + dvd.getCatNo( ) + " " +
                                dvd.getTitle( ) + " on loan." );
                }
                else
                {
                        System.out.println( "This transaction is not acceptable " +
                                "because it exceeds the maximum number\n" +
                                "of DVDs that you are permitted to have on loan." );
                        throw new ItemLimitException( "This transaction is not “ +
                                “acceptable because it exceeds the maximum number\n" +
                                "of DVDs that you are permitted to have on loan by ",
                                ( maxOnLoan - noOnLoan + 1 ) );
                } // end inner else
        } // end outer else

} // End of takeItemsOnLoan.
```

It is not important that the reader understands all of the code. The important things to notice are that the method declaration specifies that the method throws **ItemLimitException** objects and these are thrown in the inner else block.

Download free eBooks at bookboon.com

Instances of **ItemLimitException** are caught by one of the buttons of the application's graphical user interface (GUI). We will explore GUIs in a later chapter.

The simplified code for the method that calls **takeItemsOnLoan** is as follows:

```
// if the button pressed is the 'Borrow DVD' button.
{
        // Get the film from the list of DVDs available for loan.
        // Get the catalogue number of the DVD.
        // Use the catalogue number to borrow the film using the member's DVD card.
        // takeItemOnLoan throws an exception.
        try
        {
                membersCard.takeItemOnLoan( catNo );
        }
        catch( ItemLimitException ile )
        {
                messagesArea.setText( ile.getMessage( ) );
        }

} // end if
```

Error messages are output to a text area component of the user's GUI.

Download free eBooks at bookboon.com

Click on the ad to read more

There in only one exception specified in the API that is thrown by methods in the themed application. An **IOException** is thrown by the **readObject** method of the **ObjectInputStream** class and the **writeObject** method of the **ObjectOutputStream** class, as shown by the following extracts from the API.

readObject

```
public final Object readObject()
                    throws IOException
```

and

writeObject

```
public final void writeObject(Object obj)
                   throws IOException
```

Therefore when either of these methods is called, the calling method must either catch **IOException** objects or declare them to be thrown. The two methods in the **MediaStore** class of the themed application that call **readObject** and **writeObject** are shown next.

```
/** This method reads the file of members.   */
public void readMembers( ) {

        try { // Start of try block.
             // The String is the path to the file.
                FileInputStream fis = new FileInputStream("C:\\Temp\members.dat");
                ObjectInputStream ois = new ObjectInputStream( fis );
                // Note the cast in the next statement.
                members = ( Member [ ] )ois.readObject( );
                ois.close( );
                fis.close( );
        } // End of try block.
        catch ( IOException e ) { // Start of catch block.
                System.out.println( "Error: " + e.getMessage( ) );
        } // End of catch block.

} // End of readMembers.

/**  This method writes the array of members to its file.  */
public void writeMembers( ) {
        try { // Start of try block.
                FileOutputStream fos = new
                        FileOutputStream("C:\\Temp\\members.dat");
                ObjectOutputStream oos = new ObjectOutputStream( fos );
                oos.writeObject( getMembers( ) );
                oos.flush( );
                oos.close( );
                fos.close( );
        } // End of try block.
```

```
        catch ( IOException e ) { // Start of catch block.
                System.out.println( "Error: " + e.getMessage( ) );
        } // End of catch block.

    } // End of writeMembers.
```

The code from the themed application shown in this section illustrates the flexibility of the **Exception** class in that objects of this class are used to catch developer-defined exceptions *and* those declared in the Java API.

4.7 Summary of Exceptions

Objects of the **Exception** class are handled in a Java application when they are declared to be thrown by methods of classes documented in the Java API. These checked exceptions respond to error conditions outside the control of the developer. In the case of the themed application, **IOException** objects are caught by the method that transfers data out of the application to a file and the method that reads these data back in to the application.

Developer-defined exceptions respond to specific invalid conditions that are a function of the business rules associated with an application. In the case of the themed application, a developer-defined exception occurs when a member of the Media Store attempts to use their card in a way that is not permitted by the business rules of the Media Store.

The examples discussed in this chapter aim to show that there are advantages to using **Exception** objects to represent error conditions compared to the traditional approach to error handling adopted in non-OOP languages.

1. The use of objects to represent error conditions means that code to handle - i.e. catch - exceptions is separate from application logic.
2. If there are circumstances where an exception does not need to be caught at the point it is thrown, it can be propagated up the call stack to reach whichever calling method is chosen to handle it.
3. Given that all exceptions are objects, the class hierarchy of the Exception class can be used to put similar exceptions into groups.

The outcome of point 3 is that the class of an exception object indicates the type of exception thrown by a method. Instances of the **IOException** class and its descendants, for example, are a group of related exceptions that represent the kinds of error associated with input/output (I/O) to/from a Java application. We will find out how to use some of the I/O classes in Chapter One in *An Introduction to Java Programming 3: Graphical User Interfaces*.

The next chapter explores one of the most important concepts associated with Java, namely that of the *interface*.

Download free eBooks at bookboon.com

5. Java Interfaces

Up to this point in the guide, the reader will not have encountered any examples of a *Java interface*. This chapter explains how members, in addition to those accessible to a class by means of inheritance, can be introduced into Java classes by means of a special class known as a Java interface.

5.1 What is a Java Interface?

Examples discussed in previous chapters readily show that *method implementations* embody the application logic of a Java application and *method invocations* run the application. It almost goes without saying, therefore, that the methods implemented in the classes defined in a Java application are fundamental to its operation.

In the light of the content of previous chapters, *what do we know about a method written by a developer or one that is specified by the Java API?* To answer this question, let us re-visit two of the methods in the themed application.

Firstly, the documentation and declaration of the **takeItemOnLoan** method of the **DvdMembershipCard** class is as follows.

```
/**
* This method takes a specific DVD on loan.  It overrides the method in the parent class.
* @param catNo  The catalogue number of the DVD to take on loan.
*/
public void takeItemOnLoan( String catNo ) throws ItemLimitException { }
```

Secondly, let us recall from Chapter Four that the **readMembers** method of the **MediaStore** class calls the **readObject** method of one of the I/O classes in a **try** block. Part of the documentation of the **readObject** method, taken from the Java API, includes the following information:

```
readObject
  public final Object readObject()
                    throws IOException
```

The documentation of both methods includes a number of elements that are known to the user of the method. The known elements of the two methods are listed in the box on the next page.

Download free eBooks at bookboon.com

- the method's identifier;
- the method's return type;
- the method's parameters;
- the exceptions thrown by the method.

The contents of the box above are, in fact, the method's return type, signature and throws clause. For the sake of brevity, we will refer to the set of (four) components as the method's *description*.

There is no reason why we cannot extrapolate this notion and state that *any* Java method can be similarly described.

The description of a method is aimed at the *user* of the method, i.e. the code – another method - that invokes it. This means that the description of a method must be known to the developer so that other methods can invoke the method without having to know *how* the method is implemented. All that the calling method needs to know are the four items in the box above.

For example, the calling method of the **takeItemOnLoan** method in the themed application is a method that calls **takeItemsOnLoan** when one of the buttons of the application's GUI is pressed. Recalling the simplified code for the calling method from Chapter Four, we can see that the code shown below only needs to know that **takeItemsOnLoan** throws an exception and that it takes an argument of the **String** type, as follows:

American online
LIGS University
is currently enrolling in the
Interactive Online BBA, MBA, MSc,
DBA and PhD programs:

- enroll by September 30th, 2014 and
- save up to 16% on the tuition!
- pay in 10 installments / 2 years
- Interactive Online education
- visit www.ligsuniversity.com to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education.
More info here.

Download free eBooks at bookboon.com

Click on the ad to read more

```
// if the button pressed is the 'Borrow DVD' button.
{
        // Get the film from the list of DVDs available for loan.
        // Get the catalogue number of the DVD; this is a String.
        // Use the catalogue number to borrow the film using the member's DVD card.
        // takeItemOnLoan throws an exception.
        try
        {
                membersCard.takeItemOnLoan( catNo );
        }
        catch( ItemLimitException ile )
        {
                messagesArea.setText( ile.getMessage( ) );
        }

} // end if
```

The code shows that the calling method does not need to know how **takeItemOnLoan** is implemented in terms of its statements; the calling method calls **takeItemOnLoan** in the knowledge of its description.

Similarly, the code that calls the **readObject** method of the I/O stream needs to know that **readObject** returns an **Object**, takes no arguments and throws an exception. The calling method in the themed application is shown below.

```
/**
* This method reads the file of members.
*/
public void readMembers( ) {

        try
        { // Start of try block.
                // The String is the path to the file.
                FileInputStream fis = new FileInputStream("C:\\Temp\members.dat");
                ObjectInputStream ois = new ObjectInputStream( fis );
                // Note the cast in the next statement.
                members = ( Member [ ] ) ois.readObject( );
        } // End of try block.
        catch ( IOException e ) { // Start of catch block.
                System.out.println( "Error: " + e.getMessage( ) );
        } // End of catch block.
```

Download free eBooks at bookboon.com

```
        finally
        {
                ois.close( );
                fis.close( );
        }

} // End of readMembers.
```

The code shows that the calling method does not need to know how **readObject** is implemented; the calling method calls **readObject** in the knowledge of its description specified in the API.

The two examples above show that a calling method does not need to know *how* the method called is implemented; rather, it only needs to know *what* the method does in terms of its description. The outcome of decoupling the *how* and *what* of a method, illustrated by the examples above, means that the implementation of a method can change – to make it more efficient for example – *without* changing its description.

The outcome of the examples and their explanation above serves to emphasise the fundamental importance of a method's description in that it provides the developer with sufficient information to invoke it. Therefore, we can regard the notion of a method's description as a kind of *contract* offered by the method in that it indicates how it is invoked. Extrapolating from the examples above implies that *any* method that invokes another method can do so in the knowledge of the latter's description; the calling method does not need to know how the called method is implemented. In practice, if the body of a method is changed without changing its description, method invocations do not have to be re-written.

Download free eBooks at bookboon.com

Click on the ad to read more

The outcome of decoupling method implementation from method description has profound implications in Java. Methods can be published, as in the Java API, so that developers can programme to their description without regard to their implementation.

Publishing the descriptions of methods of Java classes implies that they can be standardised so that a *client application* – i.e. one that invokes published methods – can be written in the knowledge of the description of the methods provided by a *server application* – i.e. one that implements the published method.

(There is a category of applications – that are outside the scope of this guide – in which the client application calls server methods across the Internet. The provider of the server part of the application publishes – to Java developers – the description of methods implemented by the server application so that the developer can write Java clients independently of the implementation of the server. This category of Internet-based application is known as a *Web service*. Web services rely on published method descriptions known as *interfaces*. In the case of a Web service, the client and server applications can be written in any language; in fact, the client and server do not have to be written in the same language given that the client is programmed to the server application's interface.)

The outcome of this section is to emphasise the fundamental importance of a method's description and leads to the concept of a *Java interface*.

> A Java interface is a category of Java class that includes the description of one or more methods in a collective contract that other classes can use.

Classes that use a Java interface are said to *implement* that interface. Defining and implementing interfaces is explained in the next section.

5.2 Defining and Implementing a Java Interface

An interface is similar to a class in that it declares members. An interface can declare constants and declares methods by their description; there are no method bodies. Interfaces cannot be instantiated; they can only be implemented by classes or extended by other interfaces.

Defining an interface is similar to declaring a class definition. For example, a version of the themed application includes the interface shown below.

```
/** The Interface CardStatus introduces new behaviour to its implementing classes.
* @author  David M. Etheridge.
* @version 1.0, dated 29 November 2008.
*/
```

Download free eBooks at bookboon.com

```
public interface CardStatus {

    /**
    * setStatus sets the status of the card to either "Standard" or "Premier" when it is called.
    * @param  status  A String to set the variable status in the MembershipCard class.
    */
    void setStatus( String status );

    /**
    * getStatus returns the value of the attribute status.
    * @return  The value of the variable status in the MembershipCard class.
    */
    String getStatus( );

    /**
    * setDiscout sets the level of discount of the card.
    * @param  discount  An integer used to set the discount for the card.
    */
    void setDiscount( int discount );

    /**
    * getDiscount returns the value of the discount.
    * @return  The value of the discount.
    */
    int getDiscount( );

} // End of interface definition.
```

It should be noted that method declarations in an interface are terminated with a semi-colon and do not require the modifier public.

In this version of the themed application, the **MembershipCard** class implements the **CardStatus** interface and provides a body for each method declared in the interface. If the developer omits any of these methods, the compiler issues a warning to this effect. A simplified version of the **MembershipCard** class follows, the purpose of which is to show the reader how the interface is implemented by the class.

```
/**
* The MembershipCard class implements the methods required for transactions carried out
* by a member's virtual membership card.
* @author D. M. Etheridge.
* @version 1.0, dated 29 November 2008.
*/
public class MembershipCard implements Serializable, CardStatus {
```

```
// Declare instance variables.
protected int noOnLoan.
protected int maxOnLoan;
protected int discount;
protected String status;

/**
* Constructor for objects of class MembershipCard.
* @param max  The maximum number of items allowed on loan.
*/
public MembershipCard( int max ) {

    // The argument passed to this constructor is used to initialise the
    // maxOnLoan field.
    maxOnLoan = max;

} // End of constructor.

// Methods to return the maximum number of items permitted to be on loan, to return
// the number of items currently on loan, take items on loan and return items on loan
// would follow but are omitted for the purposes of the present discussion.

// Methods associated with the interface CardStatus are on the next page.
```

Download free eBooks at bookboon.com

Click on the ad to read more

```
/**
* This method sets the variable called status.
* @param status  The value of the card's status: "Standard" or * "Premier", is passed to
* the method when it is called.
*/
public void setStatus( String status ){

        this.status = status;

} // End of implementation of setStatus.

/**
* This method returns the value of the variable called status.
* @return status  The value of status.
*/
public String getStatus( ){

        return status;

} // End of implementation of getStatus.

/**
* This method has an empty body in this version of the application.
*/
public void setDiscount( int discount ) { }

/**
* This method also has an empty body in this version of the application.
* @return discount  The value of discount.
*/
public int getDiscount( ) { }

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

As the code shows, the setter and getter methods for the variable **discount** are empty in this version of the themed application. (An empty method is often referred to as a *stub*.) However **setDiscount** and **getDiscont** must be implemented, either with a body or as a stub, because the contract of the **CardStatus** interface demands as much. If either **setDiscount** or **getDiscount** are omitted by the developer, the compiler would issue a warning to this effect.

The class declaration also shows that an interface with the identifier **Serilaizable** is also implemented by **MembershipCard**. It is evident, therefore, that a class may implement more than one interface.

Download free eBooks at bookboon.com

The definition of a Java interface can be summarized by the following general syntax:

public interface < NameOfInterface > extends < SuperInterfaces >

An interface, unlike a class, can extend any number of interfaces in a comma-separated list.

5.3 The Role of Interfaces as a Means to Introduce Behaviour to a Class

A similar version of the themed application modifies the class definition for **MembershipCard** as follows:

public class MembershipCard extends Card implements Serializable, CardStatus {

and shows that a class can extend from only one class – as we discovered in Chapter Three – but can implement more than one interface. This means that the **MembershipCard** class not only contains its own methods and those inherited from the **Card** class, it also contains methods declared in the interface **CardStatus**. (The interface **Serializable** is one of a number of *tagged* interfaces. Tagged interfaces do not declare methods; they are used to indicate to the run-time system that an activity is required; in this case the activity required involves reading and writing objects to an input/output stream. [We will encounter input/output streams in Chapter One in *An Introduction to Java Programming 3: Graphical User Interfaces*.] This activity is known as *object serialization* – or *serialization* for short - and is used in the themed application to store an array of **Member** objects to a file. Serialization is achieved by declaring that a class implements the **Serializable** interface.)

Both versions of the class declarations of **MembershipCard** referred to above implement the **CardStatus** interface and, in effect, introduce behaviour to the class in addition to inherited behaviour. Given that Java does not permit multiple inheritance, interfaces are used as an alternative in order to add methods to a class. Adding methods to a class by means of interfaces gives them a very important role in Java.

The role of Java interfaces arises due to the fact that interfaces are not part of the general class hierarchy. An extract from the API for the **Serializable** interface of the **java.io package** shown on the next page illustrates this.

Download free eBooks at bookboon.com

java.io

Interface Serializable

All Known Subinterfaces:

AdapterActivator, Attribute, Attribute, Attributes, BindingIterator, ClientRequestInfo, ClientRequestInterceptor, Codec, CodecFactory, Control … etc.

Selecting the subinterface **Attribute** displays to the following page of the API.

Download free eBooks at bookboon.com

Click on the ad to read more

Interface Attribute

All Superinterfaces:

Cloneable, Serializable

All Known Implementing Classes:

BasicAttribute

The point of these illustrations is not to find out what the various interface do, it is to note that interfaces have their own internal hierarchy that is not part of the overall class hierarchy. This means that a class can implement more than one interface and an interface can be implemented by more than one unrelated class.

Interfaces combine with classes in that the API specifies which classes implement which interface, as is shown above for the interface **Attribute**.

5.4 Interfaces as Types

This chapter has shown that a Java class can inherit from only one class but that it can implement more than one interface. This means that an object can have multiple types: its own type, superclass types – as we saw in Chapter Three - and the types of all interfaces that the class implements.

Consider, for example, the version of the themed application shown in the screen shot:

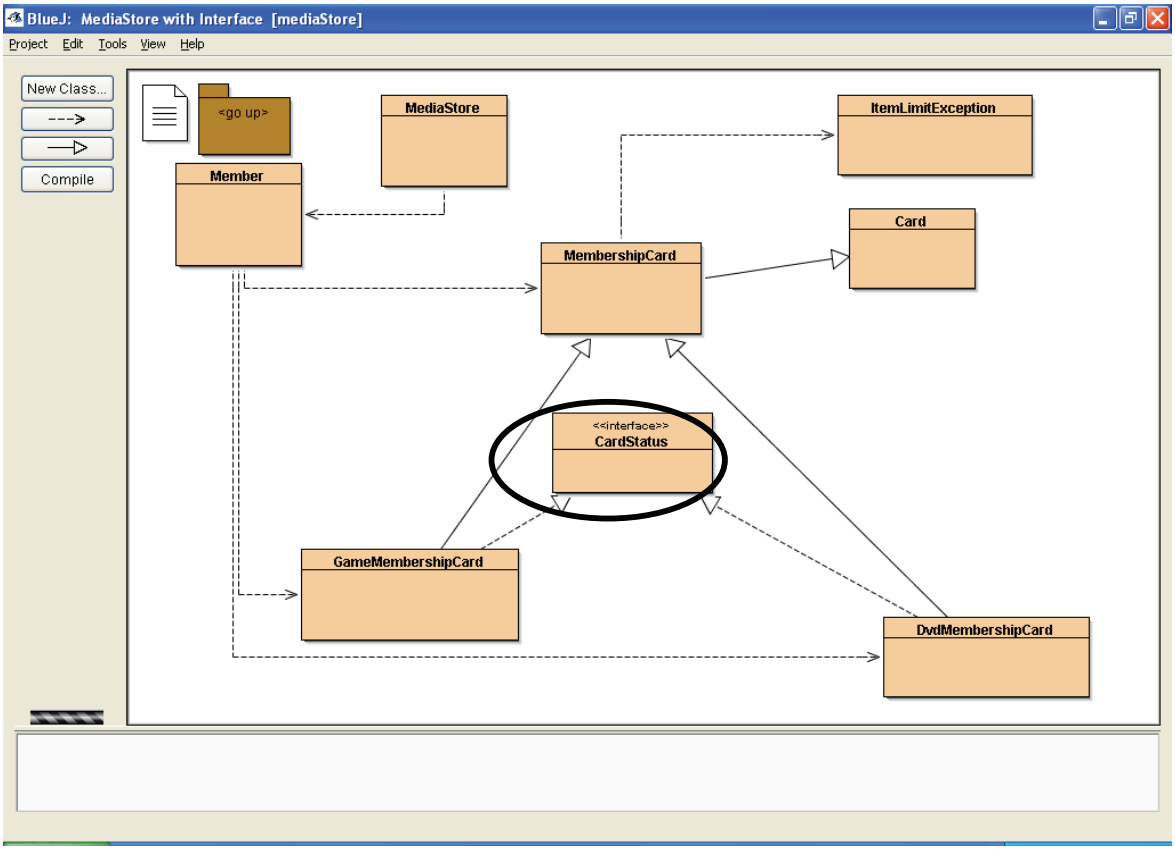


Figure 5.1 The class diagram for (a version of) the themed application

For the purposes of the present discussion, it is not important to know what each class in the application does, rather, the reader should note that the figure shows that **MembershipCard** inherits from **Card** and has two subclasses which both implement the **CardStatus** interface.

In the context of the application shown in Figure 5.1, the following statements compile:

```
// Note: the constructors for MembershipCard, DvdMembershipCard and
// GameMembershipCard take an integer argument.
// Instantiate an object whose type is the same as the run-time type.
Card cardOne = new Card( );
// Instantiate an object whose type is the superclass of the run-time type.
Card cardTwo = new MembershipCard( 10 );
// Instantiate an object whose type is the interface implemented by one of its run-time types.
CardStatus cardThree = new DvdMembershipCard( 10 );
// Instantiate an object whose type is the interface implemented the other run-time type.
CardStatus cardFour = new GameMembershipCard( 10 );
```

The third and fourth statements show that when a class type variable is declared to be an interface type, the variable can be used as an object reference to an object that implements that interface. In other words, the instance can be accessed by a reference of the interface type.

Now that it has been have shown - by example - that interfaces can be declared as class types, this raises a question: *how can we use an interface as a type?*

5.4.1 The Use of Interfaces as Class Types

Referring to the example in Section 5.4, let us assume that a substantial amount of code has to be written for a class that process instances of **DvdMembershipCard** objects. The code would, firstly, instantiate, an instance of **DvdMembershipCard** as follows:

```
// Instantiate an object whose type is the interface implemented by DvdMembershipCard.
CardStatus membersCard = new DvdMembershipCard( 10 );
```

The code that follows this declaration might include a number of statements that include the variable **membersCard**.

Let us also assume that code also has to be written for a similar class that process instances of **GameMembershipCard** objects. This code would also instantiate an instance of a **GameMembershipCard** object as shown next:

```
// Instantiate an object whose type is the interface implemented by GameMembershipCard.
CardStatus membersCard = new GameMembershipCard( 10 );
```

Let us assume that the code that follows this declaration is the same as the code that processes a member's DVD card. (This assumption is based on the notion that card processing is the same irrespective of the type of card.)

Download free eBooks at bookboon.com

This means that the code that processes a member's DVD card can be copied to form the code that processes a member's games card. The only statement that would need to be re-written is the one that calls the constructor of the card; all other statements that include the variable **membersCard** would not have to be re-written.

A similar use of interfaces as types arises when a number of similar classes implement the same interface. For example let us assume that there are a number of data structures that can be used to store a specific primitive data type or object type, all of which implement the same interface.

Instantiating an object of one of these data structures is generalised as follows:

```
// SomeDataStructure implements AnInterface.
AnInterface dataStructure = new SomeDataStructure( );
```

followed by code that includes the variable **dataStructure**.

If we wish to change the data structure, in the light of testing the application, we need to re-write only one line of code, i.e. the call to the constructor of the new data structure, as follows:

```
// SomeOtherDataStructure implements AnInterface.
AnInterface dataStructure = new SomeOtherDataStructure( );
```

Download free eBooks at bookboon.com

Click on the ad to read more

Subsequent statements that include the variable **dataStructure** do no have to be re-written.

The examples in this section show how the compatibility rules for using interfaces as types can be exploited to minimise re-writing code and gives rise to significant re-useability of code.

5.5 Summary of Java Interfaces

The principal concepts associated with Java interfaces are summarised next.

- Java allows a class to inherit from only one superclass but allows the class to implement one or more interfaces.
- Interfaces are not part of the class hierarchy; unrelated classes can implement the same interface.
- Interfaces can be declared as class types at compile time.

The ways in which Java interfaces can be used is summarised as follows.

- Declaring methods that one or more classes can implement.
- Determining an object's programming interface without the need to reveal the body of the methods implemented in the class.
- Exposing similarities amongst unrelated classes without forcing a "is a" class relationship.
- Simulating multiple inheritance by declaring that a class implements more than one interface.

A close inspection of the title of the window displayed in Figure 5.1 reveals that it includes the term **[mediaStore]** (in square brackets). The component of the application, whose classes are shown in Figure 5.1, referred to as **[mediaStore]** is an example of what is known as a *Java package*. The next chapter explains the purpose of packages in a Java application.

Download free eBooks at bookboon.com

6. Grouping Classes Together in a Java Application

Chapter Twelve explains how classes and interfaces can be grouped together, using a concept not unlike that of a generic namespace arrangement. Collecting a number of related classes together in a single structure known as a *package* makes them easier to manage and avoids potential naming conflicts.

6.1 An Introduction to Java Packages

As an illustration of namespace management, assume that a banking application has two classes named **Account**: one of the classes is the definition of a class for a current account and the other class is that for a savings account. The name of the two classes can be the same (**Account**) as long as they are placed in separate packages. This is because each package creates a new namespace such that the types in one package do not conflict with the same types in another package.

A simple analogy might help the reader to appreciate the need to manage class names and their namespaces. The use of packages in a Java application is rather like the obvious distinction between a business address in Main Street in Freetown and a similar address in Main Street in Freeville as

Top Ten Records, Main Street, Freetown and Top Ten Records, Main Street, Freeville

The notation for the address of each Top Ten Records store is for the obvious benefit to everyone concerned.

The outcome of placing the classes named **Account** in separate packages means that their unique, *fully-qualified* names are written as follows:

```
currentAccount . Account      // refers to the Account class in the package named
                              // currentAccount
```

and

```
savingsAccount . Account      // refers to the Account class in the package named
                              // savingsAccount
```

The notation used for the fully-qualified name of a class means that it can be referred to unambiguously and accessed by means of the usual ‘ **.** ‘ (dot) selector.

6.2 Creating Packages

A version of the themed application shown in Figure 6.1 below indicates that classes of the application are bundled together in one of two packages depending upon their function in the application.

Download free eBooks at bookboon.com

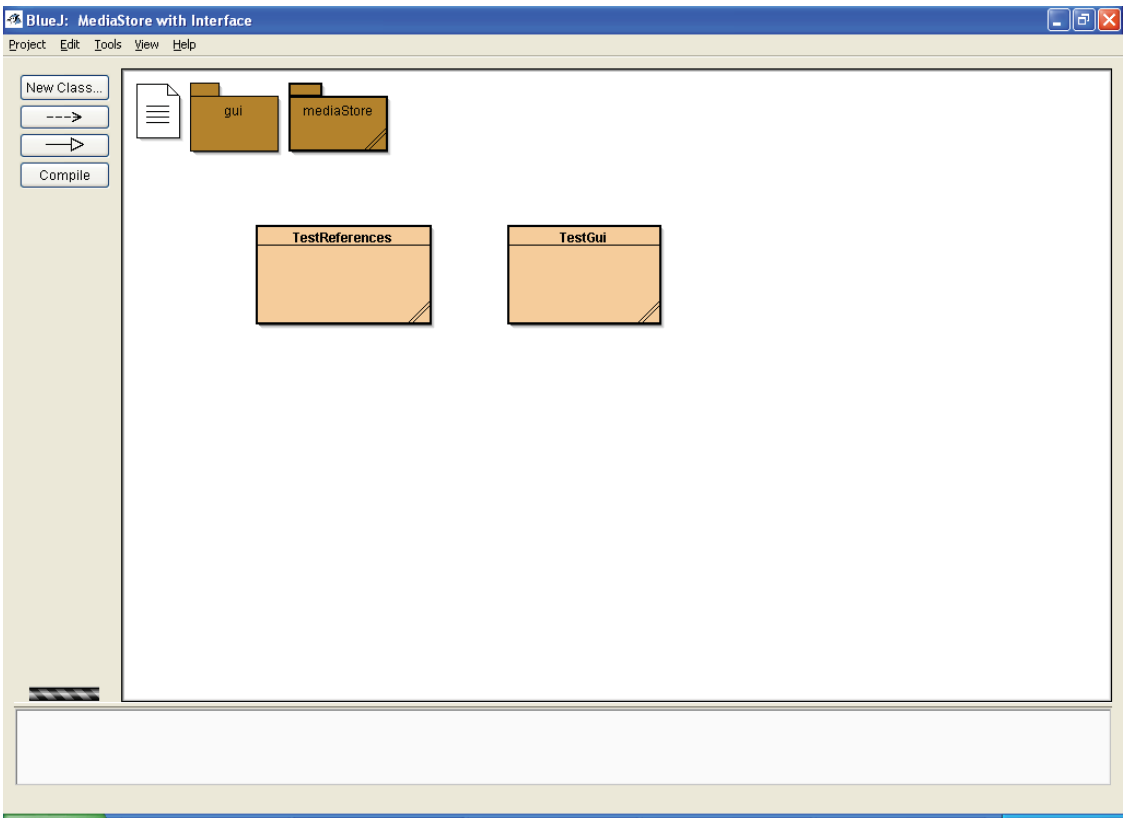


Figure 6.1 The package structure of the themed application

The two test classes – i.e. the ones that include a **main** method – named **TestGui** and **TestReferences** are not in a named package and are said to be in the *default package* of the application.

The package named **gui** contains a single class named **MediaStoreGui**. The statements

```
package gui;            // the class MediaStoreGui is in this package
import mediaStore.*;    // the class has access to the classes contained in the other package of
                        // the application
```

are written *before* the class declaration and state that the class named **MediaStoreGui** is in the package named gui and that it requires access to all of the classes in the package **mediaStore** by virtue of the wildcard ‘ * ‘ written to the right-hand side of the (dot) selector in the **import** statement.

This version of the themed application has only one class that displays the graphical user interface (GUI) for the application. Nevertheless, it makes sense to place this class in its own package: other versions of the application might contain additional classes that are concerned with the display of the GUI that and would be placed in the package named **gui**.

The **import** statement is required because components of the GUI require access to classes in the **mediaStore** package to invoke their methods and run the application.

Download free eBooks at bookboon.com

The package named **mediaStore** contains eight classes, each of which begins with the **package** statement

package mediaStore;

The eight classes are bundled together in their own package because they represent the business logic of the application. None of these classes has any role in the display of the GUI; therefore, there is no requirement to import the contents of the **gui** package into the package named **mediaStore**.

Whilst the two test classes *could* have been placed in their own package, this version of the themed application is used to illustrate the contents of the default package of the application. Both test classes require access to classes in both packages, which means that the following **import** statements appear at the beginning of each source code file:

import gui.*;
import mediaStore.*;

There is no package statement at the head of the source code file of the two test classes; this means that these two classes are automatically placed in the default package of the application.

The version of the themed application discussed above illustrates the principal reasons why classes and interfaces are bundled in a package; i.e. the types in a package are functionally related in the context of an application. A number of other reasons derive from this; they are summarised in the box shown on the next page.

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

Reasons why packages are employed in a Java application:

- developers can easily determine from the structure of an application how application logic is partitioned amongst its packages;
- developers can easily find related classes in an application;
- packages in the Java API imply that their contents are a collection of related classes (see later in this chapter);
- the names of types in one package won't conflict with the same names in another package;
- types in a package can have unrestricted access to one another but restricted access to types outside the package (see later in this chapter).

The structure of an application, in terms of its packages, is reflected in its directory structure as used by the operating system that hosts the application. For example, the package structure of the version of the themed application shown in Figure 6.1 above is reflected in its directory and folder structure in Windows™ as shown in Figure 6.2 below.

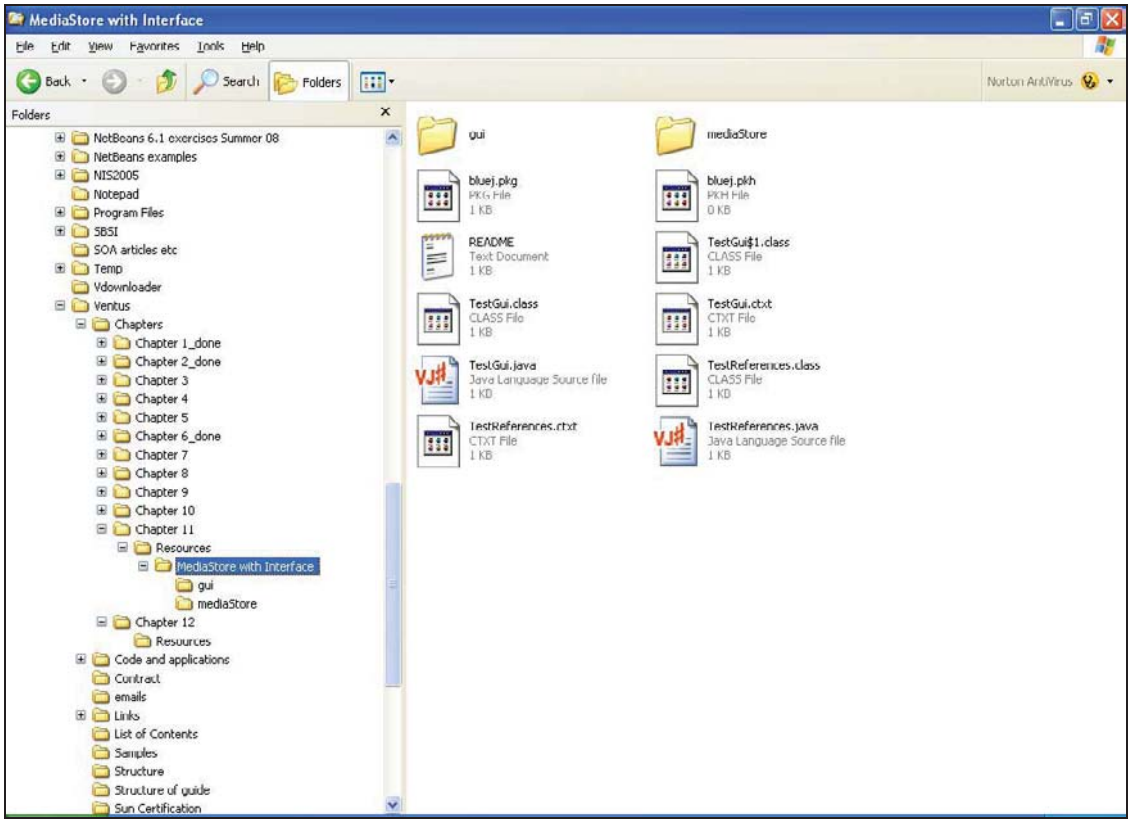


Figure 6.2 The directory of the application shown in Figure 6.1

Figure 6.2 shows that the two test classes are in the default (unnamed) package and remaining classes have been placed in one of two packages. The IDE used to develop the application shown in figures 12.1 and 12.2 automatically creates a Windows™ folder for each package created by the developer. In practice, all IDEs will create a Windows™ folder for each package created by the IDE.

Download free eBooks at bookboon.com

6.3 Naming Convention

Figures 12.1 and 12.2 show that the author (of this guide) has named the packages of the themed application with a single word that starts with a lower case letter. This simple convention works well unless independent developers use the same package and class names for applications that are in the public domain or that are accessible by members of the Java development community.

To overcome the potential problem of more than one application containing a class named **mediaStore.Member** – to take, as an example, one of the classes of the themed application – there is a convention in the Java developers' community whereby an organisation uses its Internet domain name in reverse to start package names. For example, the package named **mediaStore** shown in figures 12.1 and 12.2 should – by convention – be named along the following lines:

uk.ac.bcu.tic.etheridge.mediaStore

so that the class named **Member** is given the fully-qualified name of

uk.ac.bcu.tic.etheridge.mediaStore.Member

Up to this point in the chapter, we have largely considered packages provided by the developer of an application. The Java language itself is partitioned into a number of packages as is suggested by the statements at the head of the class definition for the **MediaStore** class of the **mediaStore** package of the themed application, as follows:

```
package mediaStore;
import java.io.*;
```

The first statement denotes that the **MediaStore** class is a member of the **mediaStore** package. The second statement implies that the class requires access to *all* of the classes of the **java.io** package.

The next section looks briefly at the use of package in the Java language.

6.4 Packages in the Java Language

The mane of the majority of the packages in the Java language begins with **java** or **javax.** For example, the relevant page of the Java API for the **java.io** package is shown in Figure 6.3 below.

Download free eBooks at bookboon.com

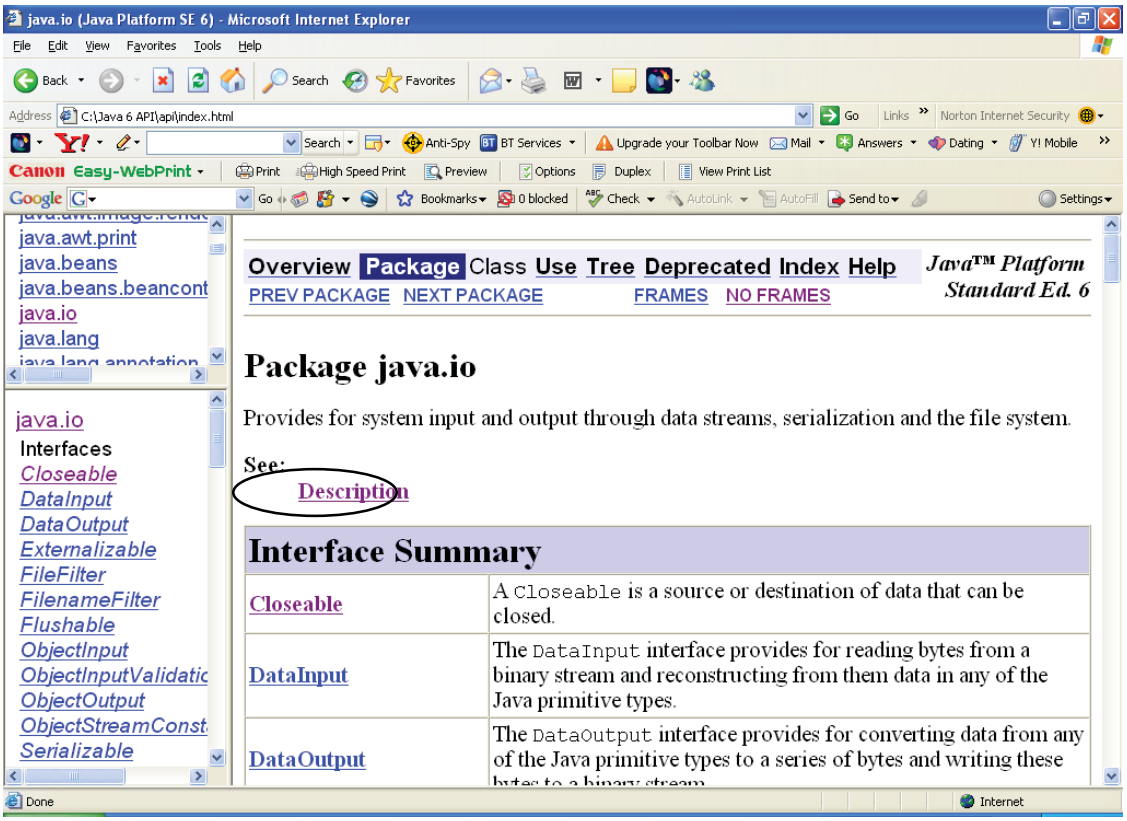


Figure 6.3 The opening page of the java.io package in the API

The lower left-hand pane shows the first part of the list of interfaces, classes and exceptions associated with the package. There are about 60 classes in the **java.io** package that are available to the Java developer.

Although the statement

import java.io.*;

referred to above implies that the author wishes to import *all 60* classes of the **java.io** package into the **MediaStore** class of the themed application, he used only four of these classes in the code. In this case, four import statements would have sufficed, as shown on the next page.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

The set of import statements imports the four classes of the **java.io** package required for the **Member** class. (Chapter Thirteen examines some of the classes of the **java.io** package in the context of what are known as input/output streams.)

The use of the 'import all' statement

```
import java.io.*;
```

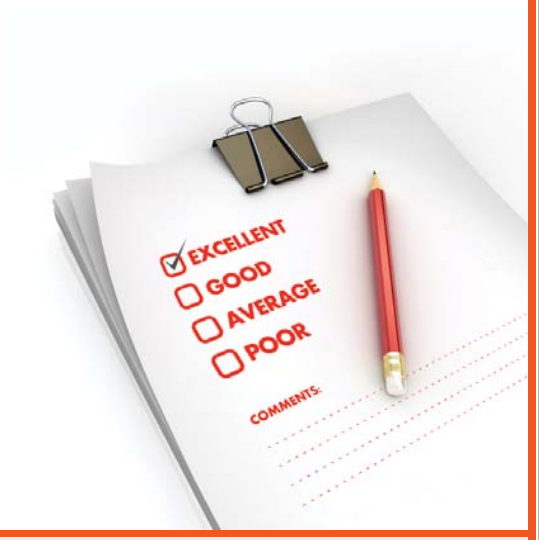
is useful when developing an application because it means that the developer can use *any* of the classes of – in this case - the **java.io** package without having to insert specific import statements.

The large, central pane shown in Figure 6.3 begins with a brief description of the contents of the **java.io** package and includes a hyperlink to a more detailed description. This kind of overall structure of linked HTML pages in the Java API is typical of most packages, of which there are about 225 in the Java Version 6 Standard Edition API. The sheer number of packages – in excess of 200 – confirms the large scale and very wide functionality of the standard edition of the Java language. Clearly, in a guide such as this one, the author can only scratch the surface of what is available to the developer in the API.

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now



Go to www.helpmyassignment.co.uk for more info

Helpmyassignment

Click on the ad to read more

At this point, it is worth mentioning that the Java run-time system automatically imports the (classes of) the **java.lang** package into all applications. The **java.lang** package contains about 40 classes, including widely-used ones such as the wrapper classes - **Boolean**, **Character**, **Double**, **Integer** and so on - that are object representations of the primitive data types, as well as classes such as **System**, **String** and **Thread**. The rationale for this automatic inclusion is that commonly-used classes should be readily available without having to insert import statements for them.

6.5 Using and Accessing Package Members

We have seen that the **import** statement provides access to one or more members of a named package in a class definition. In order to use a public member from *outside* its package, one of the following must be done:

1. The member must be referred to by its fully-qualified name.
2. The member must be imported.
3. The member's entire package must be imported.

Referring to a member by its fully-qualified name is required each time the reference is used in Java statements and, therefore, is a satisfactory approach when there are relatively few such uses. For frequent references, it makes more sense to import the member at the head of the class definition. Once a member has been imported by an **import** statement, it can be referred to in the code by its class name rather than by its fully-qualified name.

However, care should be taken when importing members of packages when, for instance, there are identically-named classes in them. For example, consider the following code snippet:

```
package mynewpackage;
import currentAccount.Account;
import savingsAccount.Account;
```

A reference to the type **Account** in the class definition that follows the **package** statement and **import** statements would produce a compiler error because the compiler does not know which of the two **Account** classes is being referred to. In such a case, the compiler will be able to distinguish between the two **Account** types when their fully-qualified name is used in the code even though both **Account** classes have been specifically imported.

When a member's *entire* package is imported, *any* member of that package can be referred to by its class name in the subsequent code.

6.5.1 Controlling Access to Package Members

Let us recall the access levels for the access modifiers listed in Table 3.1; the table is reproduced in Table 6.1 on the next page

Download free eBooks at bookboon.com

Modifier	Same Class	Same Package	Subclass	Universe
private	Yes			
default: no specifier	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

Table 6.1 Access levels to class members

The first row of Table 6.1 reaffirms what we know about the access modifier 'private': i.e. the class itself has access to its **private** members as we would expect.

The second row can be regarded as a class's *package contract* whereby if no modifier is specified classes in the same package have access to such members and trust one another to access each other's members. A class's package contract can be used to hide things such as implementation details that are not required to be known to classes outside the package. This type of contract means that such details can be changed without changing a class's *public contract.*

A class's public contract declares it as a type that is available to developers using its containing package. Thus the fourth row shows that *all* classes have access to **public** members, regardless of their package and parentage. We will return to the third row in due course.

As we work our way down the 'Modifier' column from '**private**', '**default**' (i.e. package level), '**protected**' and '**public**', each of the four access levels embraces a wider scope of the kind of class that can access that member.

Let us return to the third row of the table that was first displayed in Chapter Nine: the *protected contract.* The third row indicates what level of access is provided when a class member is declared to be 'protected'. The first column indicates that other members of the class itself have access to the **protected** member of that class; the second column indicates that classes in the same package, regardless of their parentage, have access to the **protected** member of the class; the third column indicates that subclasses of the class have access to the **protected** member, regardless of what package they are in. However, the subclass-protected table entry requires further comment.

The subclass-protected entry in the table can be stated more precisely: *a protected member can be accessed from a class via object references that are of the same type as the class or one of its subtypes.* Despite the apparent clarity of this statement, it gives rise to an interesting twist as the following example aims to illustrate.

Download free eBooks at bookboon.com

Suppose that we have a simple class in a package named **packageone** that includes a **protected** member, as shown by the following code.

```
package packageone;
public class Cat {
        // The class Mammal is also in packageone but its code is not included here.
        protected Mammal mammal;
        public void setSpeciesName( ) { }
} // end of class definition
```

A subclass of **Cat** is in a separate package.

```
package packagetwo;
public class DomesticCat extends packageone.Cat {
        // Inherits the protected variable mammal of the Mammal type.
        // Other members of DomesticCat would follow next.
} // end of class definition
```

Suppose that **DomesticCat** overrides the method **setSpeciesName** with a simple implementation.

```
public void setSpeciesName( ) {
        // The current object is a subclass of Cat, so access to the protected member inherited
        // from Cat is allowed.
        this.mammal = null;
}
```

Download free eBooks at bookboon.com

Click on the ad to read more

Given that the current object (**this**) is a subclass of **Cat**, access to the protected member **mammal** is allowed even though **Cat** and **DomesticCat** are in different packages.

Suppose that one of the methods of **DomesticCat** takes a **DomesticCat** as an argument and accesses its protected member, as follows.

```
public void aMethod( DomesticCat florence ) {
        packageone.Mammal  m = florence.mammal;
}
```

This method compiles because the method accesses the protected member of an object passed as an argument. Access is allowed because the class attempting to access the protected member (**mammal**) is a **DomesticCat** and the type of the reference **florence** is also a **DomesticCat**.

Finally, suppose also that the class **DomesticCat** defines an overridden method, as follows.

```
public void aMethod( packageone.Cat florence ) {
        packageone.Mammal  m = florence.mammal;
}
```

The statement

```
packageone.Mammal  m = florence.mammal;
```

doesn't compile; the compiler issues the message:

mammal has protected assess in packageone.Cat

The reason why the compiler issues the message is that the class attempting to access the protected member is **DomesticCat** and the type of the reference **florence** is **Cat**. Access to the protected member is not allowed because the **Cat** type is not the same as or a subclass of **DomesticCat**. Although every **DomesticCat** object a subclass of a **Cat** object, not every **Cat** object is a **DomesticCat**.

Although some of the points made in this sub-section may appear to be rather erudite at first, it *is* important that the leaner emerges from this section with a reasonable understanding of access levels and how they are controlled by means of access contracts. Even though some of the implications of controlling access to class members may appear to be technically difficult, they will be more easily understood when the reader encounters compiler messages that issue warnings about access to protected members.

6.6 Compiling and Running Package Members

It is likely that the learner will use a learner-level IDE such as BlueJ to write their first Java programmes to learn the basics of the language. When the learner gains experience, he or she can progress to the use of a professional-level IDE such as NetBeans™. In any event, an IDE typically provides buttons or menu options to use to compile and run a Java application; the IDE will 'find' and interoperate with the Java compiler installed on the learner's computer.

Download free eBooks at bookboon.com

If the learner does not use a learner-level IDE in the first instance, code can be written using a simple text editor and source code files can be compiled from the DOS prompt in the case of a Windows™ operating system.

Referring to the example shown in Section 6.5.1, the package named **packageone** includes a test class named **UserClassOne** with a **main** method. This class can be compiled from the DOS prompt with the following command:

C:\ > javac packageone \ UserClassOne.java

and **main** can be executed as follows:

C:\ > java packageone . UserClassOne

The fully-qualified class name and directory path are, as we would expect, in parallel. This means that the developer can go to the directory that *contains* the folder named **packageone** and compile and run any of the classes in that package. Similarly, the develop could compile all of the classes in the package named **packagetwo** as follows

C:\ > javac packagetwo \ *.java

As long as the programmes **javac** and **java** are on the computer's PATH environment variable, source code files can be compiled from the DOS prompt. The compiler issues the same messages in the DOS prompt window as it does if an IDE is used to compile and run the application.

The next chapter examines some of the classes of the **java.io** package that provide for input to and output from applications.