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

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.

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.

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.

```
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.

```
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
```

```
                {
                        // 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.
```

```
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.

```
// 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.

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.

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.

```
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.

```
// 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;
}
```

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.

```
/**
* 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.

```
/**
* 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.

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.

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*.