Java: Graphical User Interfaces

An Introduction to Java Programming

David Etheridge



bookboon
The eBook company

David Etheridge

Java: Graphical User Interfaces

– An Introduction to Java Programming

Download free eBooks at bookboon.com

Java: Graphical User Interfaces – An Introduction to Java Programming
© 2009 David Etheridge & Ventus Publishing ApS
ISBN 978-87-7681-496-0

Download free eBooks at bookboon.com

Contents

1.	**The Input/Output Package**	**6**
1.1	An Introduction to Streams	7
1.2	Categories of Streams and their Classes	7
1.3	Using Streams	11
1.4	Object Streams	19
1.5	Files and File I/O	21
1.6	Data Streams	25
1.7	Summary of Streams	27
2.	**Collecting Data II**	**28**
2.1	The Java Collections Framework	28
2.2	The Core Collection Interfaces	28
2.3	Implementation Types	31
2.4	Operations, Methods, Iterators and Algorithms	34
2.5	Generics and the Collections Framework	36
2.6	Collections in the Themed Application	42
2.7	Summary of the Java Collections Framework	46



Download free eBooks at bookboon.com

Click on the ad to read more

3.	**User Interfaces**	**47**
3.1	What is a User Interface?	47
3.2	Client/Server Applications	49
3.3	The Construction of User Interfaces	50
3.4	A Visual Approach to GUI Design	64
3.5	Activating User Interface Components	68
3.6	The GUI for the Themed Application	83
3.7	Summary of Event Handling	87
4.	**Concurrency with Threads**	**90**
4.1	An Introduction to Threads	90
4.2	Creating Threads	91
4.3	Using Threads in Java Applications	93
4.4	Summary of Threads	100



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

1. The Input/Output Package

Chapter One considers some of the classes of the **java.io** package. Java defines input and output (I/O) in terms of classes known as *streams*. Streams provide system input and output in a way that isolates the developer from the details about how an operating system provides access to system resources for the purposes of I/O. Streams are not required for input and output when a graphical user interface (GUI) is used to capture and display information in an application. Graphical user interface design is examined in Chapter Three.

There are approximately 60 classes in the **java.io** package. Consequently, this guide does not aim to cover every stream class. Instead, some of the main categories of streams are explained in general terms and examples are provided of the use of specific types of streams in a practical situation.

The reader is referred to the **java.io** package of the API for the documentation associated with the many stream classes provided by the Java language.



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

1.1 An Introduction to Steams

A stream is an abstraction of the underlying mechanism that is used by an operating system to transfer information into and out of a Java programme. The level of abstraction means that the developer uses classes of the **java.io** package for I/O. As a consequence, I/O can be regarded as a high-level programming activity that transparently maps onto the low-level mechanisms associated with system I/O.

A stream is a sequence of bits of information that is passed along a virtual path between a source and a destination. An input stream provides a path from a source to a programme and, similarly, an output stream is a path from a programme to a destination. Figure 1.1 visualises the general representation of streams as paths between code and a source or a destination.

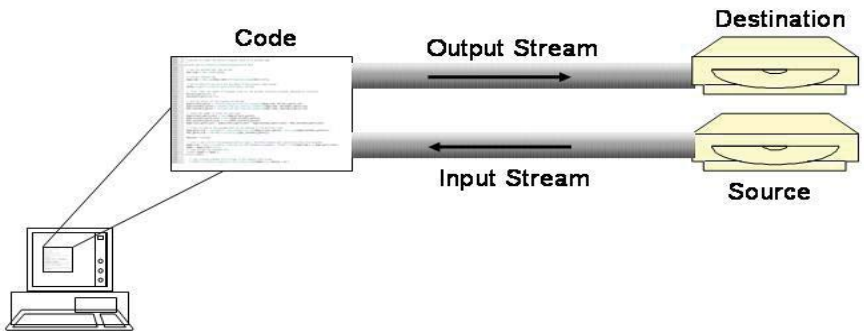


Figure 1.1 The source and destination associated with a stream

Sources and destinations of information include files, disks and networked resources; the information passed along streams can take any form, such as objects, text, images and sound.

1.2 Categories of Streams and their Classes

Streams in the **java.io** package usually occur as input/output pairs and fall into one of three categories - *byte streams*, *character streams* or *object streams*. This section looks at classes in the first of these two categories in relatively general terms to give a flavour of their functionality and to encourage the reader to refer to the API for the **java.io** package. A later section examines object streams.

1.1.1 Byte Streams

A byte streams carries a sequence of binary data and is one of two types, either an *input stream* or an *output stream*. To read a byte stream in an application, one of the subclasses of the **InputStream** class is used. An extract from the API displayed on the next page shows some of the input stream classes that are subclasses of the abstract class **InputStream**.

Download free eBooks at bookboon.com

java.io

Class InputStream

java.lang.Object
  └ **java.io.InputStream**

All Implemented Interfaces:

Closeable

Direct Known Subclasses:

AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, ObjectInputStream

Table 1.1 below summarises the main functions of these **InputStream** types, as indicated by the documentation for each class in the API.

Type	Function
AudioInputStream	Reads a specified audio format and length of audio file
ByteArrayInputStream	Contains in internal buffer that contains bytes read from the stream
FileInputStream	Inputs bytes from a file in a file system
FilterInputSteam: has a number of subclasses	Contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality
ObjectInputStream	Reads primitive data and objects previously written using an ObjectOutputStream

Table 1.1 Some of the input streams

Some of the corresponding output stream classes that are subclasses of the abstract class **OutputStream** are shown in the next extract from the API.

java.io

Class OutputStream

java.lang.Object
  └ **java.io.OutputStream**

All Implemented Interfaces:

Closeable, Flushable

Direct Known Subclasses:

ByteArrayOutputStream, FileOutputStream, FilterOutputStream, ObjectOutputStream

Download free eBooks at bookboon.com

Table 1.2 below summarises the main function of these **OutputStream** types, as indicated by the documentation for each class in the API.

Type	Function
ByteArrayOutputStream	Data is written into a byte array
FileOutputStream	Writes data to a file
FilterOutputStream: has a number of subclasses	Contains some other output stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality
ObjectOutputStream	Writes objects to a file for persistent storage

Table 1.2 Some of the output streams

The next sub-section presents a similar overview of some of the character streams.

1.2.2 Character Streams

The Java language uses the UTF–16 (Unified Transformation Format) 16 bit encoding to represent text characters. The streams that carry text-based information are called *readers* and *writers*. To read a character stream in an application, one of the subclasses of the **Reader** class is used. The following extract from the API shows some of the readers that are subclasses of the abstract class **Reader**.



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

java.io

Class Reader

java.lang.Object
  └ **java.io.Reader**

All Implemented Interfaces:

Closeable, Readable

Direct Known Subclasses:

BufferedReader, CharArrayReader, InputStreamReader, StringReader

Table 1.3 summarises the main function of these readers, as indicated by the API.

Type	Function
BufferedReader	Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines
CharArrayReader	This class implements a character buffer that can be used as a character input stream
InputStreamReader	An InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted
StringReader	A character stream whose source is a string

Table 1.3 Some of the readers

Some of the corresponding subclasses of the abstract class **Writer** are shown in the next extract from the API.

java.io

Class Writer

java.lang.Object
  └ **java.io.Writer**

All Implemented Interfaces:

Closeable, Flushable, Appendable

Direct Known Subclasses:

BufferedWriter, CharArrayWriter, OutputStreamWriter, PrintWriter, StringWriter

Table 1.4 on the next page summarises the main function of these writers, as indicated by the API.

Download free eBooks at bookboon.com

Type	Function
BufferedWriter	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings
CharArrayWriter	This class implements a character buffer that can be used as a character output stream
OutputStreamWriter	An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted
PrintWriter	Prints formatted representations of objects to a text output stream
StringWriter	A character stream that collects its output in a string buffer, which can then be used to construct a string

Table 1.4 Some of the writers

The next section gives some examples of using streams.

1.3 Using Streams

Although most applications use a GUI to input relatively small amounts of information to an application, streams are very useful for testing the methods of classes in an application before such an interface is constructed. On the output side of an application, object streams can be used to write objects out to a file so that the data associated with an application takes on a persistent state. Streams are also extremely useful when an application needs to output information to data storage devices or input data from them.

A general algorithm for using streams for I/O in an application can be expressed as follows:

1. Instantiate a stream object: this automatically opens the stream.
2. Read from or write to the stream in a **try** block.
3. Catch **IOException** objects (and any other exceptions that may occur).
4. Close the stream.

An application typically uses more than one stream *chained* together, depending on whether a buffer or a bridge or some other functionality is required: see tables 13.1 to 13.4 for a summary of the functionality of streams. Chaining streams can be visualised as connecting pipes together, as you would do when undertaking plumbing jobs at home, as shown in Figure 1.2. (The author strongly recommends that the reader *never* attempt this kind of thing in your own home; instead, bring in a professional plumber to do the work.)

Download free eBooks at bookboon.com

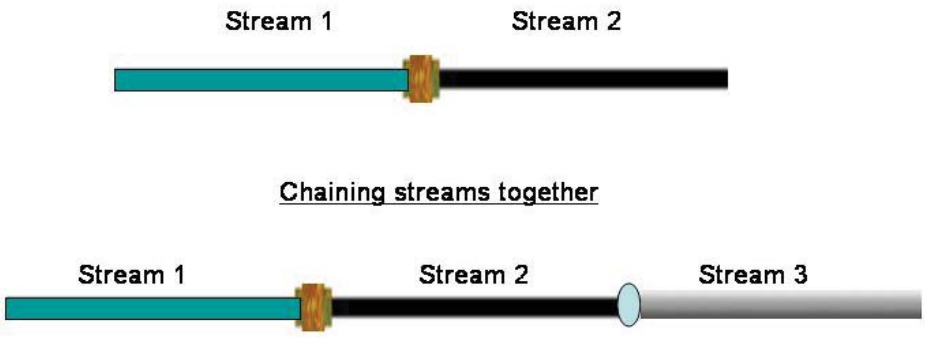


Figure 1.2 Chaining streams together

Chaining is achieved by passing a stream object into the constructor of another stream object, as will be shown by the examples that are presented in sub-section 1.3.2 below. Before embarking on the examples of chaining streams, sub-section 1.3.1 looks at how streams are used for screen output.

1.3.1 Screen Output

A number of examples in previous chapters use the statement

System.out.println(<parameter>);

to output the results of test classes to a computer's screen.



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

The **System** class includes a number of fields that are used for input and output. The static field with the identifier **out** provides an output stream of the **PrintStream** type. The stream is automatically open and is ready to accept output data.

The class **PrintStream** inherits from a subclass of **OutputStream**, as shown in the extract from the API.

java.io

Class PrintStream

java.lang.Object
  └ java.io.OutputStream
    └ java.io.FilterOutputStream
      └ **java.io.PrintStream**

A **PrintStream** object adds functionality to its parent output stream, so that it can print primitive data to the console. All characters printed by a **PrintSteam** object are converted into bytes using the platform's default character encoding. One interesting and convenient feature of **PrintStream** objects is that they never throw an **IOException**.

Standalone Java applications, such as those that use a dedicated **main** method for testing purposes, write a line of output as follows:

System.out.println(data);

The **print** and **println** methods of the **PrintStream** class are overloaded for primitive data types.

If an object reference is passed to any of the variants of **print** or **println**, as in

System.out.println(objectReference);

the **println** method calls **String.valueOf(objectReference)** to return the object's **String** value before it behaves as though it invokes **println** with a **String** passed to it. In effect, the statement behaves as if it invokes the **toString** method on **objectReference**.

In general, invoking **toString** on an object reference returns what is known as the **String** representation of the object. It is good practice to override this method for all developer-written classes so that the result is a representation of the object that is informative when invoked.

1.3.2 Keyboard Input

The static field with the identifier **in** of the **System** class is an **InputStream** object that can be used for keyboard input. Given that **InputStream** is an abstract class, an appropriate subclass is automatically instantiated when accessing the **in** field of **System**. This stream is already open and ready to supply input data. The **InputStream** class has a **read** method that reads a byte of data and returns an **int** in the range 0 to 255. The **int** is cast to convert it to a **char**.

The example that follows shows how a byte is entered via the computer's keyboard and output to the computer's screen.

Download free eBooks at bookboon.com

```
import java.io.*;

public class Keyboard {

    public static void main( String[ ] args ) {
        try
        {
            System.out.print( "Please press any key: " );
            char key = ( char )System.in.read( );
            System.out.print( "The key pressed was: " + key );
            System.out.println( "The class is: " + System.in.toString( ) );
        }
        catch( IOException ioe )
        {
            // do something about the exception
        }
    } // end of main

} // end of class definition
```

Executing main results in the following output:

```
C:\ > java Keyboard
Please press any key: a
The key pressed was: a
The class is: java.io.BufferedInputStream@3e25a5
```

The output shows that selecting **`System.in.toString( )`** returns the object reference of the object that is automatically instantiated when referring to **`System.in`** and shows that the object is of the type **`BufferedInputStream`**. The class **`BufferedInputStream`** is shown in the next extract from the API.

java.io

Class BufferedInputStream

```
java.lang.Object
  └java.io.InputStream
      └java.io.FilterInputStream
          └java.io.BufferedInputStream
```

We can conclude, therefore, that selecting **`System.in`** instantiates an object of the **`BufferedInputStream`** type and closes the stream when it has been finished with.

The code for the **`Keyboard`** class (on the previous page) implies that invoking **`read`** should be included in a loop to add each **`char`** to a **`String`** to, in effect, input a string of characters via the keyboard. However, the code would have to detect when the last character has been entered in order to determine the end of the loop.

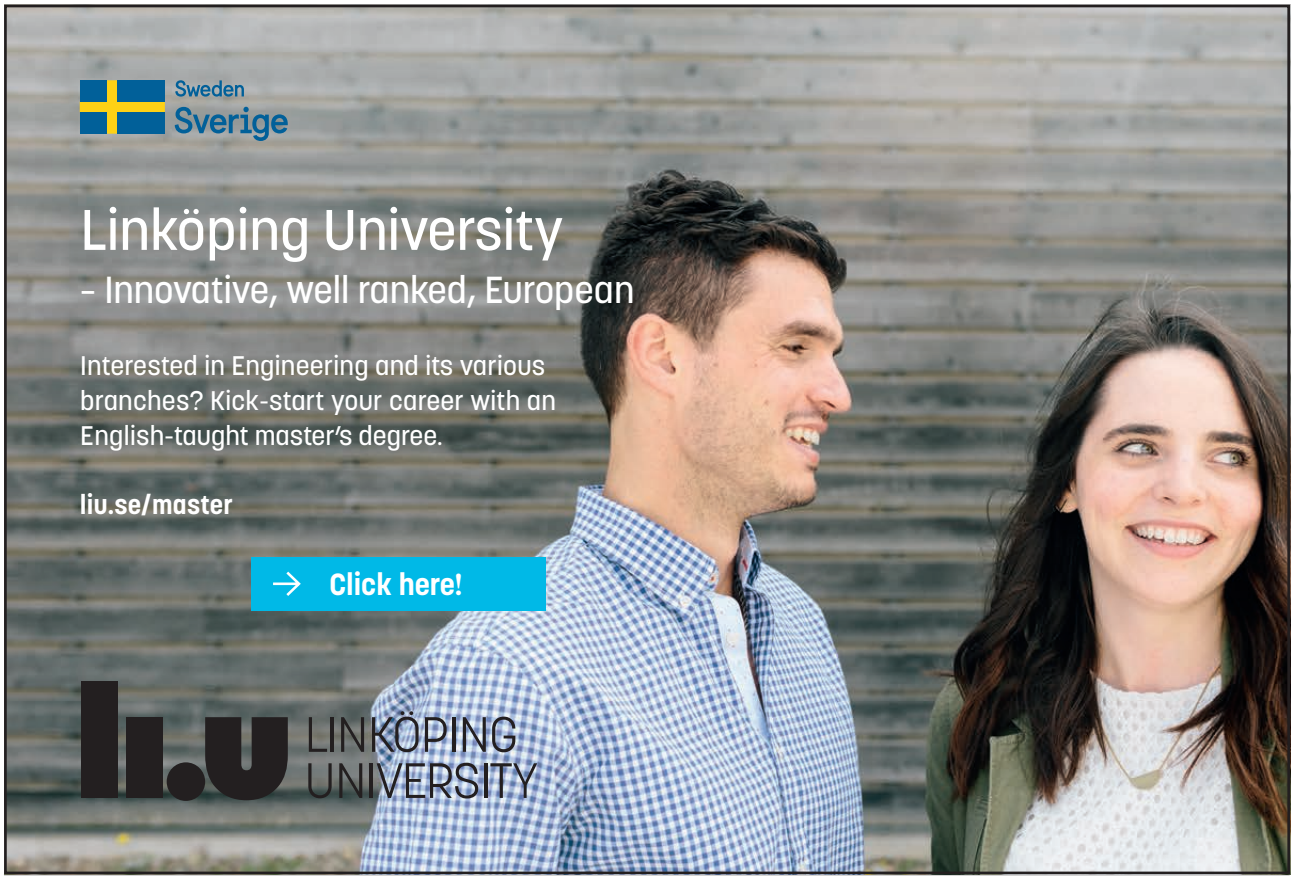
Download free eBooks at bookboon.com

In short, using **System.in**.read is not a particularly useful way to input data from the computer's keyboard. As an alternative, a number of streams can be chained together in order to read characters and strings and numbers from the keyboard.

String Input via the Keyboard

The following class chains three streams in order to read characters from the keyboard. Firstly, **System.in** is the byte stream that is used for keyboard input, as shown in the previous sub-section. Secondly, **System.in** is connected to a stream of the **InputStreamReader** type to act a bridge between the byte stream and a character stream. Thirdly, a buffer is required because keyboard input tends to be irregular. Therefore, the **InputStreamReader** is connected to a buffer of the **BufferedReader** type. The **readLine** method of **BufferedReader** reads, in the example below, a set of characters from the keyboard. When the enter key is pressed, the method returns a **String**.

Chaining the three streams together is achieved by passing one stream object to the constructor of the next stream in the chain, as shown in the class definition that follows and continues on the next page.



Click on the ad to read more

Download free eBooks at bookboon.com

```
import java.io.*;

public class KeyboardInput {

    public static void main( String[ ] args ) {
        // Instantiate a bridge stream and pass the object instantiated by System.in to
        // its constructor.
        InputStreamReader isr = new InputStreamReader( System.in );
        // Instantiate a buffered stream and pass the InputStreamReader to its
        // constructor.
        BufferedReader kbd = new BufferedReader( isr );
        try
        {
            System.out.print( "Enter some characters and press “ +
                    “return when finished: " );
            String s = kbd.readLine( );
            System.out.println( "The String was: " + s );
            // Close the initial stream; this will close all streams connected to it.
            kbd.close( );
        }
        catch( IOException e )
        {
            e.printStackTrace( );
        }
    } // end of main

} // end of class definition
```

The result of executing main is:

```
Enter some characters and press return when finished: Hello World
The String was: Hello World
```

Numerical Input via the Keyboard

System.in cannot be expected to distinguish numerical input from pressing any other of the keys on the keyboard. The object instantiated by **System.in** reads the next byte of data entered via the keyboard when its **read** method is invoked. When this object is chained, as in the code for KeyboardInput in the previous sub-section, the **readLine** method of a **BufferedReader** returns a **String**.

Numerical input via the keyboard is relatively straightforward in that a **String** is returned in a similar way to that used in the code for the KeyboardInput class. The **String** is then converted into the corresponding primitive type.

The class definition shown on the next page illustrates this technique for integer input.

Download free eBooks at bookboon.com

```
import java.io.*;

public class ReadInt {

    public static void main( String[ ] args ) {
        try
        {
            BufferedReader kbd = new BufferedReader( new
                InputStreamReader( System.in ) );
            System.out.print( "Enter an integer: " );
            String intStr = kbd.readLine( );
            // Convert the String into its corresponding integer by calling one of
            // the methods of the Integer wrapper class.
            int number = Integer.parseInt( intStr );
            System.out.println( "The number is " + number );
        }
        catch( IOException ioe )
        {
            // do something about it
        }

    } //End of main.

} // End of class definition.
```

The result of executing main is as follows:

```
Enter an integer: 123
The number is 123
```

The class definition shown on the next page shows a similar way to enter a double into a programme from the keyboard.

Download free eBooks at bookboon.com

```
import java.io.*;

public class ReadDouble {

        public static void main( String[ ] args ) {
                try
                {
                        BufferedReader kbd = new BufferedReader(
                                new InputStreamReader( System.in ) );
                        System.out.print( "Enter a double: " );
                        String dblStr = kbd.readLine( );
                        // The next two statements convert the String to a double.
                        Double dblObj = Double.valueOf( dblStr );
                        double number = dblObj.doubleValue( );
                        System.out.println( "The number is " + number );
                }
                catch ( IOException ioe )
                {
                        // do something about it
                }
        } // end of main

} // end of ReadDouble
```

STUDY FOR YOUR MASTER'S DEGREE
IN THE CRADLE OF SWEDISH ENGINEERING

Chalmers University of Technology conducts research and education in engineering and natural sciences, architecture, technology-related mathematical sciences and nautical sciences. Behind all that Chalmers accomplishes, the aim persists for contributing to a sustainable future – both nationally and globally.

Visit us on **Chalmers.se** or **Next Stop Chalmers** on facebook.



Click on the ad to read more

Download free eBooks at bookboon.com

As the examples show, it takes several lines of code to input primitive data values via the keyboard. Fortunately, there are a number of classes in the public domain that make this task easier than that shown in the examples above. One such class is **EasyInput** written by David J. Barnes. Non-static methods of the **EasyInput** class carry out the required steps to enter primitive data values from the keyboard. For example, the **nextInt** method of **EasyInput** returns an **int** value as follows.

```
// Instantiate an EasyInput object.
EasyInput kbd = new EasyInput( );
int intEnteredAtTheKeyboard = kbd.nextInt( );
```

That is all there is to it!

EasyInput can be downloaded from:

http://www.soft32.com/Download/free-trial/Easy_Input/4-227452-1.html

1.4 Object Streams

Section 1.2 mentions a third category of streams: *object streams*. Object streams are typically used in an application when live objects are required to be written to a byte stream and either saved to a local file or transferred across a network to a remote host. An **OutputObjectStream** object writes such an object to a stream and an InputObjectStream object reconstitutes the object from the stream.

The process of translating an object into a stream of bytes is known as *serialization*, where it is important to use the American spelling (for reasons that will be explained in the next paragraph); the reverse process of reconstituting an object from an input stream is known as *deserialization*. The term ‘serialization’ is usually used to describe the overall process of serialization and deserialization.

To enable an object to be serialized, it must implement the interface **java.io.Serializable** (spelt with a ‘z’). The **java.io.Serializable** interface is a type of interface known as a *tagged interface*. A tagged interface, such as the **java.io.Serializable** interface, does not declare any members. When **java.io.Serializable** is implemented, its purpose is to alert the JVM that an object is serializable. Serializing an object to a file saves the object’s state and deserializing the same object reconstitutes its saved state.

An interesting and vitally useful feature of object serialization is that when an object is serialized, an *object graph* is actually serialized. Writing an object graph to an object stream means that all objects referenced by the object being written to the stream are also written to the stream, as long as their class definitions include the clause to implement the **java.io.Serializable** interface.

The next sub-section discusses an example, taken from the themed application, to illustrate how serialization is used to maintain persistent data in the Media Store application.

Download free eBooks at bookboon.com

1.4.1 Using Object Streams in the Themed Application

When a member joins the Media Store, the new member is provided with a (virtual) membership card that is used to record the details about each DVD that the member takes on loan from the store. The object references of new members are added to an array that is used to store the object references of all members of the Media Store. Therefore, the array that stores the object reference to each member also stores an object graph for each **Member** object. In the context of the themed application, the object graph for a **Member** object includes references to the member's **DvdMembershipCard** object which, in turn, includes references to each **Dvd** object associated with the card. The classes **DvdMembershipCard**, **Dvd** and **Member** implement **java.io.Serializable**.

The object that represents the Media Store of the type **MediaStore** includes a method for writing out the array of members to a binary .dat file, as shown next.

```
/**
 * This method writes the array of members to a .dat file.
*/
public void writeMembers( ) {

        try
        {
                // Create a stream to write data to a file.  The String is the path to the .dat file.
                FileOutputStream fos = new
                        FileOutputStream("C:\\Temp\\members.dat");
                // Connect an object stream to the file stream.
                ObjectOutputStream oos = new ObjectOutputStream( fos );
                // Write the array of members to the object stream; getMembers returns the
                // array of members.
                oos.writeObject( getMembers( ) );
                // Release resources.
                oos.flush( );
                oos.close( );
                fos.close( );
        }
        catch ( IOException ioe )
        {
                System.out.println( "Error: " + ioe.getMessage( ) );
        }

} // End of writeMembers.
```

The method is called after each card transaction so that the application's data is persistent.

The corresponding method that reads the array of members into the application is shown on the next page.

Download free eBooks at bookboon.com

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

        try
        {
                // The String is the path to the .dat file.
                FileInputStream fis = new FileInputStream( "C://Temp//members.dat" );
                // Connect an object stream to the file stream.
                ObjectInputStream ois = new ObjectInputStream( fis );
                // Note the cast in the next statement; members is a variable of the Member [ ]
                // type.
                members = ( Member [ ] )ois.readObject( );
                // Release resources.
                ois.flush( );
                ois.close( );
                fis.close( );
        }
        catch ( IOException ioe )
        {
                System.out.println( "Error: " + ioe.getMessage( ) );
        }
} // End of readMembers.
```

The best that the compiler can promise is to return an object of the **Object** class when the **readObject** method of the **ObjectInputStream** class is invoked; it does not know – at compile time – what type of object is read from the stream. Hence there is a cast in the statement that invokes **readObject**.

Judicious use of the pair of methods **writeMembers** and readMembers serializes the object graph associated with the array of **Member** objects and provides a persistent data layer in the themed application. Thus, data about members is stored externally to the application to a .dat file on (the author's) C: drive.

1.5 Files and File I/O

File objects and file streams are useful when an application needs to write data to a file and read data from a file, where the file (or files) are used for persistent data associated with the application.

The activities associated with files and file I/O involves the following tasks: creating file objects; using utilities for file objects; reading and writing with file streams.

1.5.1 Creating File Objects

An instance of the **File** class is an abstract representation of the path name of a file and may or may not represent an existing file. For example, the following class definition compiles whether the file exists or not.

Download free eBooks at bookboon.com

```
import java.io.File;

public class Files {

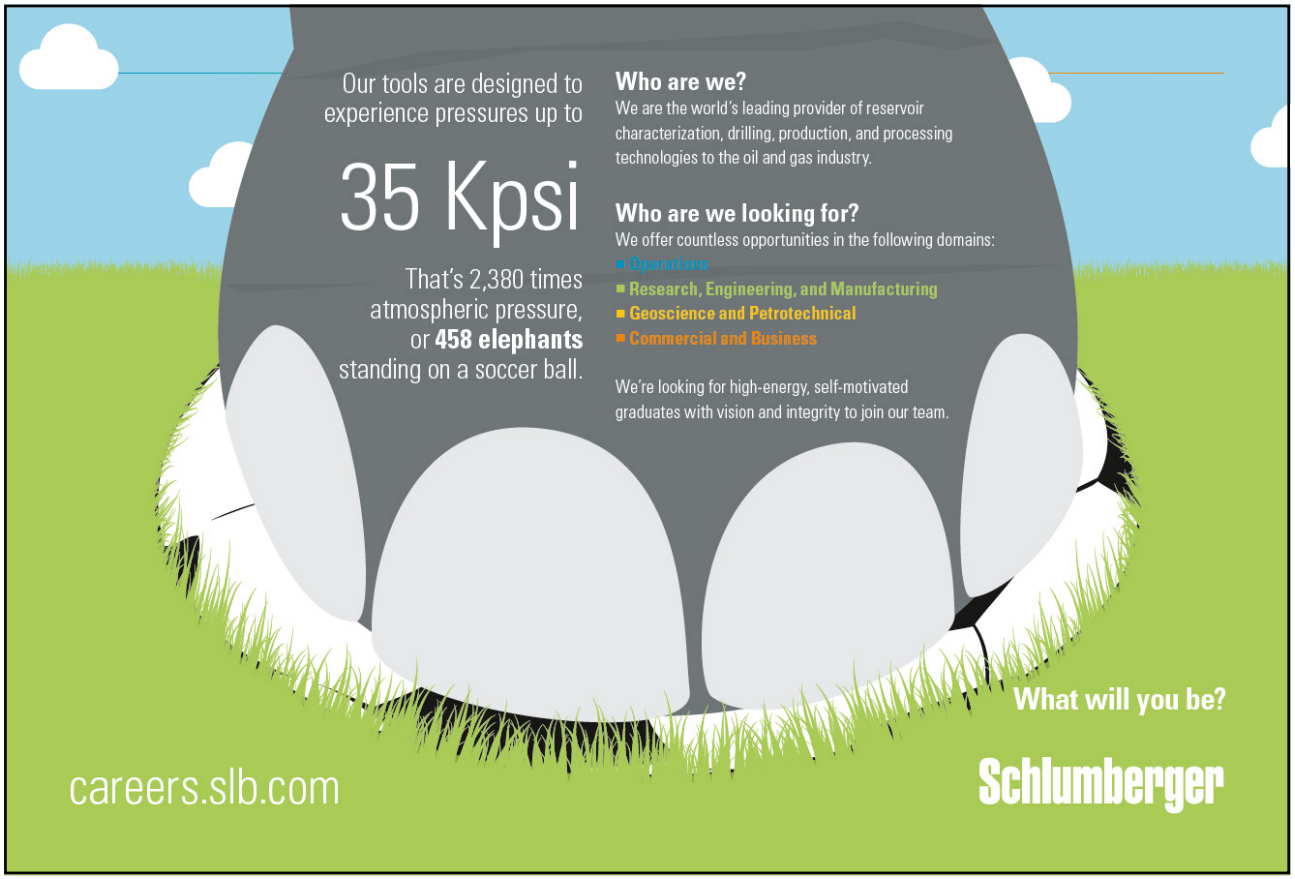
        public static void main( String[ ] args ) {
                File myFile = new File( "C:\\myfile.txt" );
        } // end of main

} // end of class definition
```

If the file does not exist at the path specified and if the file object with the reference **myFile** is selected in subsequent code, a **FileNotFoundException** is thrown.

1.5.2 File Tests and Utilities

If the file *does* exist, there are several tests and utilities that are available as methods of the **File** class. The class that follows illustrates some of these tests and utilities: **try ... catch** blocks are omitted for the sake of brevity.



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

```
import java.io.File;

public class Files {

    public static void main( String[ ] args ) {
        File myFile = new File( "C:\\myfile.txt" );
        System.out.println( myFile.getName( ) );
        System.out.println( myFile.getPath( ) );
        System.out.println( myFile.exists( ) );
    } // end of main

} // end of class definition
```

The result of executing **main** is.

```
myfile.txt
C:\myfile.txt
true
```

1.5.3 Reading and Writing with File Steam Classes

The **File** streams - **FileInputStream** and **FileOutputStream** – and **File** readers/writers – **FileReader** and **FileWriter** – are used to read from and write to a file.

A **FileReader** object can be used to read character streams, as illustrated by the following code. A **BufferedReader** object is connected to the **FileReader** object used so that its **readLine** method can be invoked.

```
import java.io.*;

public class FileRead {

    public static void main( String[ ] args ) {
        BufferedReader br = new BufferedReader (
            new FileReader( "c:\\myfile.txt" ) );
        String inStr = br.readLine( );
        System.out.println( "The contents of the file are: " + inStr );
    } // end of main

} // end of class definition
```

Executing **main** displays the contents of the existing file named **myfile.txt**, as follows:

```
The contents of the file are: Hello World
```

Similarly a **FileInputStream** object can be used to read a byte stream, as illustrated by the code that follows on the next page.

Download free eBooks at bookboon.com

```
import java.io.*;

public class FileRead {

    public static void main( String[ ] args ) throws IOException {
        FileInputStream fis = new FileInputStream( "c:\\myfile.txt" );
        InputStreamReader isr = new InputStreamReader( fis );
        BufferedReader kbd = new BufferedReader( isr );
        String s = kbd.readLine( );
        System.out.println( "The contents of the file are: " + s );

    } // end of main

} // end of class definition
```

Executing **main** displays the contents of the same file named **myfile.txt**, as follows:

```
The contents of the file are: Hello World
```

Writing to a file is illustrated by the next example, in which a **FileReader** object is used to read from an existing text file and a **FileWriter** object is used to write its contents to another file. The **FileWriter** object is connected to a **PrintWriter** object so that its **println** method can be invoked.

```
import java.io.*;
public class ReadAndWrite {

    public static void main( String[ ] args ) {
        // Read from an existing file.
        BufferedReader fIn = new BufferedReader
                    ( new FileReader( "c:\\myfile.txt" ) );
        String inStr = fIn.readLine( );

        // Write the contents to another initially empty file.
        File file = new File( "c:\\mynewfile.txt" );
        PrintWriter out = new PrintWriter( new FileWriter( file ) );
        out.println( inStr );
        out.close( );
    } // end of main

} // end of class definition
```

Opening the files **myfile.txt** and **mynewfile.txt** demonstrates that the contents of the former file are written to the latter file.

Download free eBooks at bookboon.com

The problem with the simple class **ReadAndWrite** shown above is that writing to the file mynewfile.txt overwrites its existing contents. To *append* to a file, one of the constructors of the **FlieWriter** class takes a boolean which if set to **true** writes text to the end of the file. The modified statement of the class **ReadAndWrite**

PrintWriter out = new PrintWriter(new FileWriter(file, true));

results in the contents of myfile.txt appended to the contents of **mynewfile.txt**.

Random Access Files

The Java statement displayed at the end of the previous sub-section implies that there are constructors for the **FileWriter** class that give the developer s degree of control over writing to a file. The **RandomAccessFile** class is more sophisticated than the **FileWriter** class in that it provides a movable pointer to its content. Methods of the **RandomAccessFile** class provide read/write operations under the control of the pointer and give the developer a high degree of control when working with files that form the persistent data component of an application. The reader is referred to the API for further details about this class.

1.6 Data Streams

Data streams are used to transmit the binary data of primitive data types. A **DataInputStream** object reads primitive data types from an underlying input stream; a **DataOutputStream** object writes primitive data types to an output stream. An application can use data streams in tandem so that a data output stream writes data that can be subsequently read back in to the application by a data input stream.



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

The example that follows is a simple illustration of the use of a pair of data streams. The output data stream writes an **int** value to a file and the input data stream reads the value back in and outputs it to the screen.

```
import java.io.*;
public class DataStreams {

    public static void main( String[ ] args ) {
        // Declare local variables.
        DataOutputStream out = null;
        DataInputStream in = null;

        try
        {
            FileOutputStream fos = new
                FileOutputStream("C:\\mydatafile.dat");
            out = new DataOutputStream( new BufferedOutputStream( fos ) );
            // write out an int
            out.writeInt( 1234 );
            out.close( );
        }
        catch( IOException e ) { }

        try
        {
            FileInputStream fis = new
                FileInputStream( "C:\\mydatafile.dat" );
            in = new DataInputStream( new BufferedInputStream( fis ) );
            // read in the int that was previously written out
            int i = in.readInt( );
            in.close( );
            System.out.println( "The file contents are: " + i );
        }
        catch( EOFException e ) { }
        catch( FileNotFoundException f ) { }
        catch( IOException io ) { }

    } // end of main

} // end of class definition
```

It should be noted that the **writeInt** method matches the **readInt** method in the class above. In general, write and read methods used to write and read primitive data types are matched in this way.

Download free eBooks at bookboon.com

A further point to note is that the **DataInputStream** object has a very convenient way to detect the end of the file that it is reading; its **read** methods catch an **EOFException** exception instead of testing for the return of an invalid value of the primitive being read.

1.7 Summary of Streams

As is evident from the overall content of this chapter, there are very many streams in the **java.io** package. The chapter attempts to summarise most of the main types of streams and provides examples so that the reader can begin to understand how streams are used for specific tasks in an application. There are a number of stream types that are not addressed by the chapter. However, it is to be hoped that the reader gains sufficient understanding of those that are explained and illustrated by example so that further investigation of the API for the **java.io** package can be undertaken with a degree of confidence in the knowledge of the essential principles of using Java streams.

One final point is worth emphasising here: *most of the methods provided by stream classes throw IOException objects*, as indicated by the documentation for stream classes in the API for the **java.io** package. Whilst some of the examples presented in this chapter may have omitted to either declare or catch this type of exception for the sake of brevity, the reader should *always* write code that invokes methods provided by streams to catch **IOException** objects if the documentation states that the methods throws this type of exception.

Before the guide delves into graphical user interfaces (GUIs) in Chapter Three, the next chapter examines a number of data structures that can be used to store primitive data values or object references as an alternative to the humble array.



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

2. Collecting Data II

Chapter Two examines some of the classes of the *Java Collections Framework* in order to explore more flexible data structures than the humble array.

2.1 The Java Collections Framework

Chapter Six in *An Introduction to Java Programming 1: The Fundamentals of Objects and Classesshows* how arrays are used to store primitive data types or object references and, in doing so, highlights one of the drawbacks of using an array to retrieve such data. When an element is required to be retrieved from an array, the position in the array where the element is stored – i.e. its index value plus one – must be established so that the element can be found. If, as is likely, the position of an element in an array is not known in advance, the array must be searched until the element is found. Java's Collections Framework brings together a number of interfaces and classes that can be used to create a variety of data structures, some of which make storage and retrieval of data elements easier than is the case when using arrays for this purpose.

A collection type, sometimes called a *container*, is an object that groups multiple data elements into a single entity. The Java Collections Framework is a unified architecture for collection types, comprising a number of interfaces and implementations.

2.2 The Core Collection Interfaces

The core collection interfaces of the Java Collections Framework form a hierarchy of two separate trees. The following simplified extract from the Java API shows the hierarchy of the core interfaces that implement the **`Collection`** interface, the root interface of the hierarchy.

java.util

Interface Collection<E>

All Known Subinterfaces:

List<E>, Queue<E>, Set<E>, SortedSet<E>

A **`SortedSet`** implements the **`Set`** interface, as indicated in the extract from the API shown on the next page.

Download free eBooks at bookboon.com

java.util

Interface SortedSet<E>

Type Parameters:

E - the type of elements maintained by this set

All Superinterfaces:

Set<E>

The reader will have noticed the presence of an element of Java syntax immediately following the name of the interfaces listed in the extracts from the API shown above. The presence of **<E>** indicates that the interface is *generic*; we will return to this concept later (in this chapter). For the present, it is sufficient to state that when an object that implements one of the collection interfaces is instantiated, the developer should specify the type of object contained in the collection. We will explain the reasoning behind this statement in due course.

The **Map** interface is contained in a separate tree from the **Collection** interface, as shown in the next extract from the API. Although **Map** does not extend from **Collection**, as the hierarchy shows, the concepts and methods of **Map** and **Collection** are sufficiently similar to regard the **Map** interface as a member of the Collections Framework.

java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

SortedMap<K,V>

One of the benefits of implementing the **Map** interface is that the implementing data structure can store name-value pairs. For example, a **HashMap** object implements the **Map** interface and could be used to store object references with an associated **String** key. We will see an example of using name-value pairs in the themed application later in this chapter.

The interfaces listed in the extracts from the API shown above form a collection of abstract data types that are declared as *implementation types* at compile time. For example, the **ArrayList** class implements the **List** interface and is instantiated as shown on the next page.

Download free eBooks at bookboon.com

List< String > list = new ArrayList< String > ( );

The syntax **< String >** determines that the **List** stores **String** objects. Instantiating a collection in this way means that a collection can be manipulated independently from the details of its implementation.

For example, the statement above could be re-written as

List< String > list = new LinkedList< String > ( );

Subsequent code that refers to the object reference **list** does not have to change as a result of changing the implementation type from an **ArrayList** to a **LinkedList**; both of these types implement the **List** interface.

We will explore some of the implementation types in a later section.

2.2.1 The Core Interfaces

The purpose of this sub-section is to describe briefly the core interfaces of the Collections Framework.

The Collection Interface

Java does not provide any *direct* implementations of the **Collection** interface; instead, it provides implementations of the subinterfaces **List**, **Queue** and Set.

The List Interface

A **List** is an ordered collection of elements that can contain duplicate elements. Code that uses a **List** has control over where in the list each element is inserted and accesses elements by their index position in a similar fashion to an array.

The Queue Interface

A **Queue** is typically used to store elements when their order matters.

The Set and SortedSet Interfaces

A **Set** is a collection that cannot contain duplicate elements. A **SortedSet** is a **Set** that maintains its elements in their natural order.

The Map and SortedMap Interfaces

A **Map** maps keys to values; it cannot contain duplicate keys. Each key in a **Map** can only map to one value. A **SortedMap** is a **Map** that maintains its key-value mappings in ascending key order.

The next sub-section examines some of the implementation classes that implement the core interfaces.

Download free eBooks at bookboon.com

2.3 Implementation Types

Concrete implementation types are the actual data structure objects that implement the core interfaces. Figure 2.1 below summarises the types that implement the core interfaces; the columns indicate the storage technology that is used in the concrete implementation.

Inter-faces	**Implementations**				
	Hash table	**Resizable array**	**Tree**	**Linked list**	**Hash table + Linked list**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

Figure 2.1 General-purpose implementations of the core interfaces

The API can be used to find out the details of each implementation. For example, an **ArrayList** is a resizable array implementation of the **List** interface.



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

2.3.1 Constructing a General-Purpose Implementation

As a rule, the Java developer 'programmes' to the interface type, rather that to the implementation type. This approach gives the developer enormous flexibility by working with a reference to an interface, rather than directly with a concrete implementation. The choice of implementation type is usually a function of performance and is informed by the information provided in the API about the classes listed in Figure 2.1.

For example, let us assume that an application requires some kind of list of elements. The declaration of the implementation is assigned to a variable of the corresponding interface type, as in the statement

```
List< E > list = new ArrayList< E >( );
```

Subsequent code that uses the object reference `list` is independent of implementation because the code refers to the collection by its *interface* type, rather than by its *implementation* type. The flexibility that this approach provides can be illustrated by the following statement, where it has been decided to change the implementation type by changing *only* the constructor:

```
List< E > list = new LinkedList< E >( );
```

Only one line of code needs to be changed; subsequent statements that include the reference `list` do not change.

The practice of changing only the line of code that calls the constructor of the implementation class is illustrated in the following example: the relevant statement is in a larger font. The first class creates a HashSet and the second creates a `TreeSet`. In both cases, the elements of the collection are output using two different techniques.

```
import java.util.*;

public class MyHashSet {

    public static void main ( String [ ] args ) {
        // Create a String array.
        String [ ] anArray = { "deep", "fried", "Mars", "bar“ };
        // Instantiate an implementation object.
        Set< String > set = new HashSet< String >( );
        // Add the elements of the array to the collection.
        for ( int i=0; i < anArray.length; i ++ )  {
            set.add( anArray[ i ] );  // Invoke the add method of the
                                      // collection.
        }
        // Output the elements of the collection using a for ... each construct.
        for ( Object o : set ) System.out.print( o + “ “ );
        // Output the elements using an iterator.  (Iterators are discussed later.)
        Iterator< String > i = set.iterator( );  // Get an iterator for the
                                                 // collection.
```


Download free eBooks at bookboon.com

```
            while( i.hasNext( ) ) {  // while there are elements
                System.out.print( i.next( ) + " " );  // get the next element
            } // end while
        } // end main

    } // end of class definition
```

The second, similar class in shown next.

```
    import java.util.*;

    public class MyTreeSet {

        public static void main ( String [ ] args ) {
            String [ ] anArray = { "deep", "fried", "Mars", "bar" };
            Set< String > set = new TreeSet< String >( );
            for ( int i=0;  i < anArray.length; i ++ ) {
                set.add(anArray[ i ] );
            }
            for ( Object o : set ) System.out.print( o + " " );
            Iterator i = set.iterator( );
            while( i.hasNext( ) ) {
                System.out.print( i.next( ) + " " );
            } // end while
        } // end main

    } // end of class definition
```

Executing main for the first class displays the following output:

Mars fried bar deep

Mars fried bar deep

and for the second class

Mars bar deep fried

Mars bar deep fried

It is left as an exercise for the reader to investigate why the output is not in the same order as the `String` array and why traversing the `HashSet` and `TreeSet` collections give different results.

A close examination of the code for the two classes above shows that a number of collection methods are used to produce the output. The next section explores some of the main operations associated with the core interfaces.

Download free eBooks at bookboon.com

2.4 Operations, Methods, Iterators and Algorithms

The core interfaces provide a number of *operations*. These operations carry out basic or bulk tasks associated with the elements of a collection and form a set of similar methods provided for each type. Commonly-used methods typically include methods to add or remove an element from a collection. For example, the **add** method of **Set** is used in the classes discussed at the end of the previous section.

An object that implements the **Iterator** interface allows the elements of a collection to be traversed, typically for searching or output purposes. The **iterator** method of the collection returns an **Iterator**, as illustrated in the following code block in the classes discussed at the end of the previous section.

```
Iterator i = set.iterator( );  // Get the iterator.
while( i.hasNext( ) ) { // Traverse the collection if there are elements.
        System.out.print( i.next( ) + " " );  // Output the next element.
}  // Exit the loop when i.hasNext returns false.
```

The iterator has only three methods: **hasNext**, **next** and **remove**, two of which are illustrated in the code block displayed above. It is important to note that the **remove** method actually removes the last element returned by the iterator. Therefore, care should be taken when invoking the **remove** method on a collection.

Finally, in this sub-section, we will examine the polymorphic algorithms associated with the **Collections** class.




Click on the ad to read more

Download free eBooks at bookboon.com

The polymorphic algorithms take the form of static methods of the **Collections** class that take a collection object as an argument. The reader is advised to refer to the API for further details about the many methods of this class; it is a direct descendant of the **Object** object, as shown below.

java.util

Class Collections

```
java.lang.Object
  └ java.util.Collections
```

The next example illustrates one of the algorithms in action.

```
import java.util.*;

public class TestCollections {

        public static void main ( String [ ] args ) {

        String [ ] anArray = { "deep", "fried", "Mars", "bar" };
        List< String > list = new ArrayList< String >( );
        for ( int i=0; i < anArray.length; i ++ )  {
                        list.add( anArray[ i ] );
        }

                for ( Object o : list ) System.out.print( o + " " );

        System.out.println( "\n" );
        // Reverse the order of the elements in the List.
                Collections.reverse( list );
        for ( Object o : list ) System.out.print( o + " " );

        } // end of main

} // end of class definition
```

The result of executing main is

```
deep fried Mars bar

bar Mars fried deep
```

as expected.

Download free eBooks at bookboon.com

2.5 Generics and the Collections Framework

It is evident from the code and API documentation presented in previous sections that collection types are *generic*, as pointed out in Section 2.2 where the Java syntax **< E >** is first encountered. Before we investigate how generics are used in the Collections Framework, we will digress and establish the basics of generics in general classes in the next sub-section.

2.5.1 Generics in Java Classes

Generics is a relatively new concept in Java. However, as previous sections in this chapter demonstrate, we need to know enough about generics in order to use Java's Collection Framework correctly.

As we saw in Section 3.5 in *An Introduction to Java Programming 2: Classes in Java Applications*, casting an object's type satisfies the Java compiler. However if the *actual* types used at run- time haven't been correctly anticipated by the developer, this is likely to give rise to a run-time **ClassCastException**. For obvious reasons, we wish to avoid run-time errors that derive from incorrectly applying the run-time rules for type casting when writing code. Generics is a mechanism that avoids such run-time exceptions.

Generics are an enhancement to Java's type system; it provides compile-time type safety in that it 'catches' lack of type correctness at compile-time rather than at run-time. Generics eliminates the drudgery of casting and, potentially, making errors in casting types. In short, generics avoids run-time ClassCastExceptions and add stability and robustness to code.

Perhaps an effective way to introduce generics is to compare a straightforward class definition with its generic version. The Java tutorial introduces the concept of a generic class by beginning with a simple, non-generic class named **Box**. This class provides two methods: **add**, which adds an object to the box, and **get**, which retrieves it, as follows:

```
// Source: The Java Tutorial
// http://java.sun.com/docs/books/tutorial/java/generics/generics.html

public class Box {

        private Object object;

        public void add( Object object ) {
                this . object = object;
        }

        public Object get( ) {
                return object;
        }

} // end of class definition
```


Download free eBooks at bookboon.com

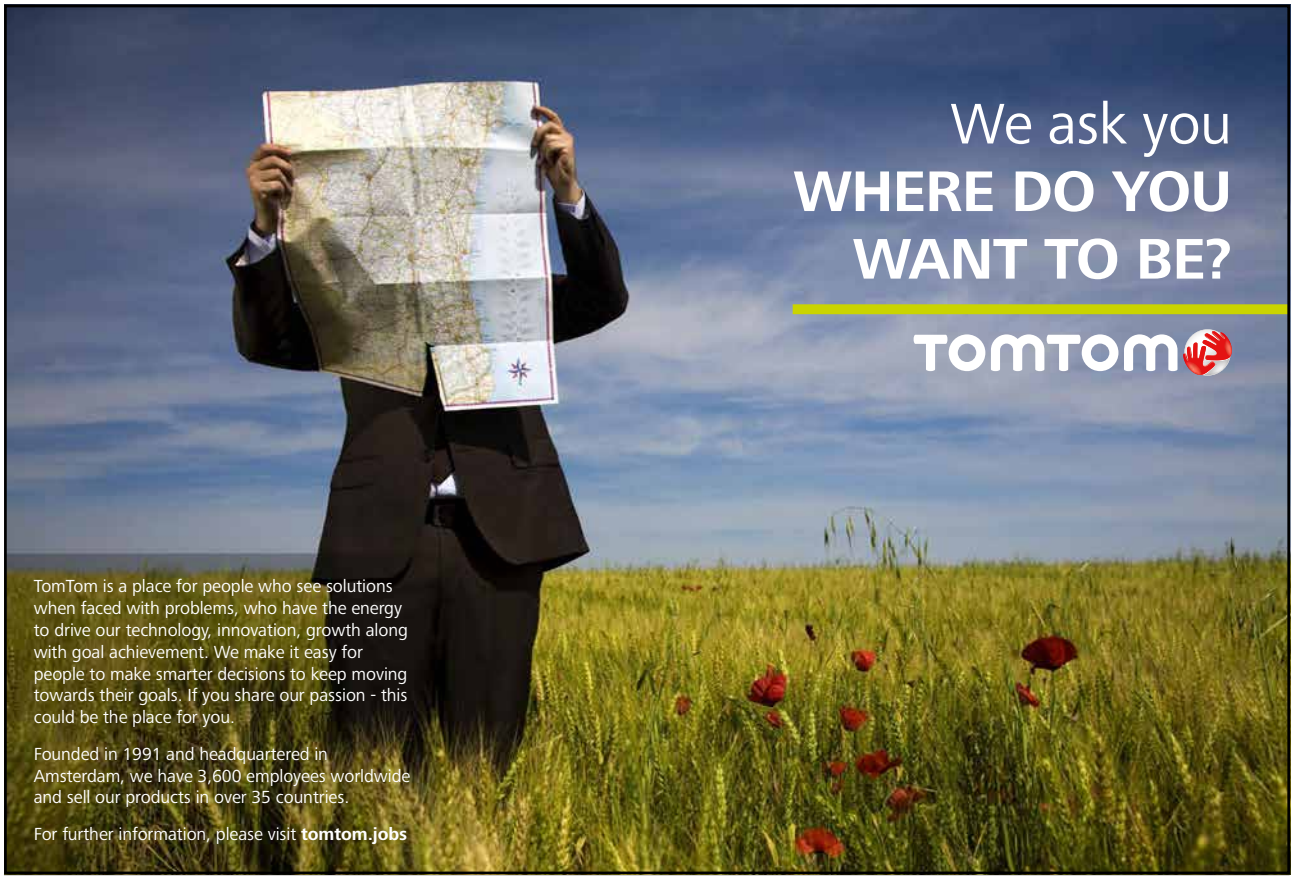
Given that the methods of **Box** accept or return an object of the **Object** type, we are free to pass in an object of *any* type, but not a primitive type. However, should we wish to restrict the type of the attribute with the identifier **object** to a specific type such as an **Integer** object, invoking **get** would require a cast because the compiler can only commit to return an object of the **Object** type. A test programme would look something like the one shown next.

```
public class TestBox {

    // only pass objects of the Integer type into the Box object
    public static void main( String[ ] args ) {
        Box integerBox = new Box( );
        integerBox.add( new Integer( 10 ) );
        // note the cast; get returns an Object object
        Integer someInteger = ( Integer )integerBox.get( );
        System.out.println( someInteger );
    }

} // end of test class
```

The cast from **Object** to **Integer** is correct because we know the return type of the **get** method. The compiler trusts that the cast is correct; it knows nothing about the *actual* type returned when the **get** method is invoked.



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

Let us imagine that a careless programmer passes an object of the wrong type to the **`add`** method. The resulting test class is as follows.

```
public class TestBox {

    public static void main( String[ ] args ) {
        Box integerBox = new Box( );
        integerBox.add( new String( "Hello World" ) );
        // the next statement compiles, but throws an exception at run-time
        Integer someInteger = ( Integer )integerBox.get( );
        System.out.println( someInteger );
    }

} // end of test class
```

The code for the test class will compile but will give rise to a run-time error as follows:

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
```

It could be argued that no sensible programmer would *ever* write a test class such as the second one shown above: the error is too obvious. The purpose of the simple example is to emphasise that in a substantial programme where, for instance, similar **`add`** and **`get`** methods are separated by very many lines of code, it is relatively easy to fall prey to cast errors. The obvious solution is to ensure that compile-time *and* run-time rules for casting types are followed correctly, as illustrated in Section 3.5 in *An Introduction to Java Programming 2: Classes in Java Applications*.

Despite the obvious solution that relies on the developer taking note of method descriptions and documentation to make correct type casts, generics provides an alternative way to avoid a **`ClassCastException`** at run-time. If the **`Box`** class is designed with generics in mind, the error in the second test class is avoided at compile-time and does not cause a run-time exception.

In essence, the non-generic class declaration for a class such as

```
public class SomeClass {
```

is modified so that it becomes a *generic class*

```
public class SomeClass<T> {
```

where **`T`** is a type variable; it can be used anywhere in the body of the class definition. (This technique can also be applied to interfaces.)

Download free eBooks at bookboon.com

The on-line Java tutorial goes on to explain the generic **Box** class by updating the **Box** class to use generics. A *generic type declaration* is created by changing the code "public class Box" to "public class Box<T>". This introduces a *type variable* named **T** that can be used anywhere inside the class.

The tutorial makes the point that we can think of **T** as a special kind of variable, whose 'value' will be whatever type is passed in. This can be any class type, any interface type, or even another type variable; it cannot be any of the primitive data types. In this context, **T** is said to be a *formal type parameter* of the **Box** class. The generic version of the class named **Box**:

```
public class Box< T > {
        private T t; // T stands for "Type"
        public void add( T t ) {
                this . t = t;
        }

        public T get( ) {
                return t;
        }
}  // end of class definition
```

All occurrences of the type **Object** in the non-generic class definition have been replaced with type **T**. To reference this generic class in a test class, a *generic type invocation* is performed, which replaces **T** with some concrete value, such as the **Integer** type, as in

```
Box< Integer > integerBox;
```

We can think of a generic type invocation as being similar to a normal method invocation, but instead of passing an argument to a method, we are passing a *type argument* — **Integer** in this case — to the **Box** class. The statement above does not actually create a new **Box** object; it simply declares that **integerBox** will hold a reference to a "Box of Integer objects", which is how **Box<Integer>** is read.

The statement

```
Box< Integer > integerBox = new Box< Integer >( );
```

instantiates a **Box** object. Once **integerBox** is initialised, we can invoke its **get** method *without* providing a cast, as shown in the following code snippet from **main** in the test class.

```
// call the constructor for Box
Box< Integer > integerBox = new Box< Integer >( );
integerBox . add( new Integer( 10 ) );
// no cast is required in the next statement
Integer someInteger = integerBox.get( );
System.out.println( someInteger );
```


Download free eBooks at bookboon.com

If an incompatible type is added to the **Box**, such as a **String**, *compilation* will fail, alerting the programmer to what previously would have been a run-time error:

```
BoxDemo3.java:5: add( java.lang.Integer ) in Box< java.lang.Integer >
cannot be applied to ( java.lang.String )  integerBox.add("10"); 1 error
```

The purpose of explaining the generic version of the **Box** class is to show that the compiler is alerted to the fact that only **Integer** objects are passed to and returned from its methods. This means that invoking the **get** method in the test class does not require a cast. Furthermore, if a careless programmer passes an incorrect type to the **add** method, this is detected at compile-time rather than at run-time.

We can now return to the use of generics in the Collections Framework.

2.5.2 The Use of Generics in the Collections Framework

Let us consider the following code snippet.

```
List myList = new LinkedList( );
myList . add( new Integer( 10 ) );
Integer i = ( Integer )myList . iterator( ) . next( );
```

As we saw in a previous section, the Java programmer must know what kind of data is in the **List** in order to make the correct cast. This is because the compiler can only guarantee that an **Object** object is returned by the iterator.

If the developer writes the following statement:

```
Double d = ( Double )myList . iterator( ) . next( );
```

the compiler will compile the statement because it has no knowledge of the type of objects returned by the **next** method; it assumes that the developer *does* know this and makes the correct cast. As a result, the statement above will compile but will cause a run-time exception as we saw previously.

The generic version of the code snippet illustrates how generics is applied to the Collection Framework. The purpose of this is to give the programmer the opportunity to express their clear intent to restrict the collection to contain a particular type, as shown on the next page.

Download free eBooks at bookboon.com

```
List< Integer > myList = new LinkedList< Integer >( );
myList . add( new Integer( 10 ) );
Integer i = myList . iterator( ) . next( );
```

The cast is gone because the compiler guarantees that type correctness is checked at compile-time rather than at run-time.

Figure 2.2 below, summarises what the syntax of the code snippet means.

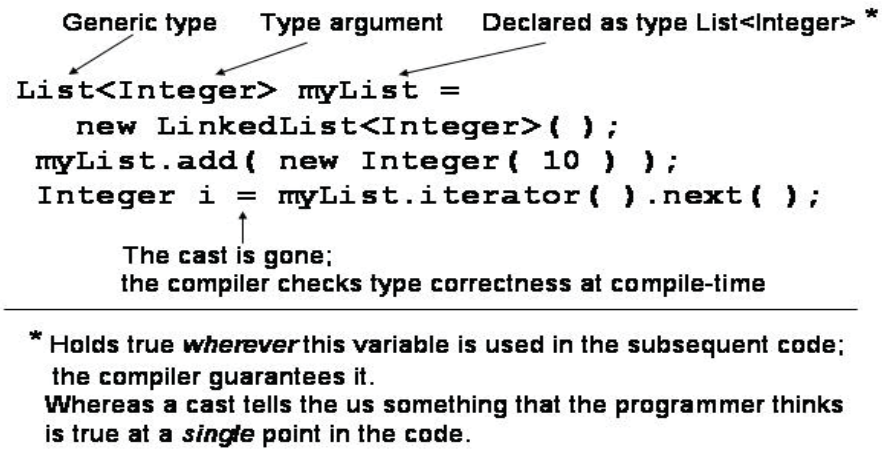


Figure 2.2 The use of generic declarations

INNOVATIVE
LIKE YOU.

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

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

UNIVERSITY OF GOTHENBURG

Click on the ad to read more

Download free eBooks at bookboon.com

Guaranteeing that type correctness is checked at compile-time rather than at run-time is, clearly, a huge advantage over the non-generic version of using collections in that it gives code a degree of robustness that was absent before generics was introduced into the Java language.

2.6 Collections in the Themed Application

One of the requirements of the themed application is to store and retrieve an array of members currently registered as having joined the Media Store. Thus, there is code that adds a new **Member** object to the array and code that writes the array out to a file for subsequent read operations.

The version of the themed application that uses a collection instead of an array includes code that makes it easier to store and retrieve members, by using statements such as the following.

```
// Create a HashMap instead of an array
Map< Integer, Member > members = new HashMap< Integer, Member >( );
// Add a member to it, using a key-value pair of the member's known
// membership number (an int) and the known reference to the Member object
members.put( membershipNumber, member );
// Get a member based on the key; no cast required
Member member = members.get( membershipNumber );
// Remove a member
members.remove( membershipNumber );
```

It should be noted that in the statement

```
members.put( membershipNumber, member );
```

the **Map** expects an **Object** as the key to the object reference for the **Member** object. However, we can pass an **int** to the **put** method because Java automatically converts the **int** with the identifier **membershipNumber** to an **Integer** object.

The table that follows – and continues on the next three pages - compares the version of one of the classes of the themed application that makes use of arrays with the same class in the version that uses a collection. The code is stripped of most of its comments and some of its methods so that the reader can gain an understanding of how the two classes compare.

Download free eBooks at bookboon.com

```
package mediaStore;
/** Array version.
 * @author D. M. Etheridge. */
import java.io.*;

public class MediaStore {

   // Declare instance variables.
   // An array to store Member objects.
   private Member [ ] members;
   // The number of members.
   private int noOfMembers;

   /**
    * This method returns a reference to the
    * array of members.
    * @return members  A reference to the
    * array of members.
    */
   public Member[ ] getMembers( ) {

      return members;

   } // End of method

   /**
    * This method adds a member to the array.
    *
    * @param fName  The member's first name.
    * @param lName  The member's last name.
    * @param uName  The member's user
    * name.
    * @param pWord  The member's password.
    */
   public void addMember(String fName,
     String lName, String uName, String pWord)
   {
      members[ noOfMembers ] = new
        Member(fName, lName, uName, pWord,
        getNextAvailableMembershipNumber( ));
        // Increment the number of members.
        noOfMembers++;
        // Increment the next available
        // membership number.
          nextAvailableMembershipNumber++;
   } // End of method

   /**
    * This method returns the member, given
    * his or her membership number.
```

```
package mediaStore;
/** Collection version.
 * @author D. M. Etheridge.  */
import java.util.*;

public class MediaStore {

   // Declare instance variables.
   // A Collection to store Member objects.
   private Map< Integer, Object > members =
      new HashMap< Integer, Object >( );
   // The number of members.
   private int noOfMembers;

   /**
    * This method returns a reference to the
    * Map of members.
    *
    * @return members  A reference to the Map
    * of members.
    */
   public Map getMembers( ) {

      return members;

   } // End of method.

   /**
    * This method adds a member to the
    * Collection of Member objects.
    *
    * @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)
   {

      // Call the constructor for Member.
      Member member = new Member( fName,
         lName, uName, pWord );
      // The member's membership number is
      // used as the key to the new member in
      // the Collection.
      // Get the membership number.
      int key = getMembershipNumber( );
      // Put this member into the Collection.
```


Download free eBooks at bookboon.com

```
     *
     * @return Member  The member associated
     * with a particular membership number.
     */
    public Member getMember( int
       membershipNumber ) {

       // Declare a local variable.
       Member member = null;

       // Search the array of members for the
       // required membership number.
       for  int i = 0; i < getNoOfMembers( ); i ++ )
       {
          if (membershipNumber==
          members[ i ].getMembershipNumber( ))
          {
             System.out.println( "Found member
             number " + membershipNumber );
             member = members[ i ];
          }
       }

       if ( member != null )
       {
          return member;
       }
       else
       {
          System.out.println("No such member.");
          return member;
       }

    } // End of method

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

       // Read in the array of members.
       try {
          // The String is the path to the .dat file.
          FileInputStream fis = new
             FileInputStream(
              "C://Temp//members.dat" );
          ObjectInputStream ois = new
             ObjectInputStream( fis );
          // Note the cast in the next statement.
          members =
             members.put( key, member );
          noOfMembers++;

       } // End of addMember.

    /** This method returns a member from the
     * Collection.
     *
     * @return member  The member associated
     * with a particular key.
     */
    public Member getMember( int key ) {

       // Get the member from the Collection.
       Member member = members.get( key );
       return member;

    } // End of method

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

       // Read in the collection of members.
       try {
          // The String is the path to the .dat file.
          FileInputStream fis = new
             FileInputStream(
              "C://Temp//members.dat" );
          ObjectInputStream ois = new
             ObjectInputStream( fis );
          // Note the cast in the next statement.
          members =
             ( Map )ois.readObject( );
          ois.close( );
          fis.close( );
       }
       catch ( Exception e ) {
          System.out.println( "Error: " +
             e.getMessage( ) );
       }

    } // End of method

    /**
     * This method writes the array of members
     * to its .dat file.
     */
    public void writeMembers( ) {
```


Download free eBooks at bookboon.com

```
            ( Member [ ] )ois.readObject( );
         ois.close( );
         fis.close( );
      }
      catch ( Exception e ) {
         System.out.println( "Error: " +
            e.getMessage( ) );
      }

   } // End of method

   /**
    * This method writes the array of members
    * to its .dat file.
    */
   public void writeMembers( ) {

      try {
         FileOutputStream fos = new
            FileOutputStream(
            "C://Temp//members.dat" );
         ObjectOutputStream oos = new
            ObjectOutputStream( fos );
         oos.writeObject( getMembers( ) );
         oos.flush( );
         oos.close( );
         fos.close( );
      }
      catch ( Exception e ) {
         System.out.println( "Error: " +
            e.getMessage( ) );
      }

   } // End of method

} // End of class definition
```

```
      try {
         FileOutputStream fos = new
            FileOutputStream(
            "C://Temp//members.dat" );
         ObjectOutputStream oos = new
            ObjectOutputStream( fos );
         oos.writeObject( getMembers( ) );
         oos.flush( );
         oos.close( );
         fos.close( );
      }
      catch ( Exception e ) {
         System.out.println( "Error: " +
            e.getMessage( ) );
      }

   } // End of method

} // End of class definition
```

Table 2.1 Comparison of part of one of the classes of the themed application that uses arrays to one that uses a Map

It is to be hoped that the reader gains an appreciation how a collection is used in the code in the right-hand column to add and remove members from the Media Store using simpler code than its equivalent in the left-hand column.

Download free eBooks at bookboon.com

2.7 Summary of the Java Collections Framework

The Java Collections Framework provides useful data structures that are manipulated by interface types; the developer 'programmes' to an interface type rather than to an implementation type. The various implementation types of each interface are interchangeable, making it very easy to change collection implementations and reuse code.

Whilst the use of generics is not unique to the Collections Framework, its use avoids type casting and results in robust code production.

The next chapter – the penultimate one in this guide – finds out how graphical user interfaces (GUIs) are constructed.



Click on the ad to read more

Download free eBooks at bookboon.com

3. User Interfaces

Constructing a user interface (UI) is one of the most rewarding features of application development using Java; it is highly creative and gives the developer the opportunity to work very closely with users to develop the UI that suits their needs. In this chapter, we will find out how user interfaces are constructed.

3.1 What is a User Interface?

The themed application is referred to from time to time in previous chapters. It is not the intention to appraise the reader about all of the details of the themed application; rather, it is used to illustrate programming concepts in the context of a realistic application. However, we haven't yet explained *how* the classes of the themed application are used in an actual application, although the purpose of using **main** methods to test classes is mentioned and illustrated in previous chapters where relevant or appropriate.

In general, when classes and their methods associated with an application have been thoroughly tested, often with **main** methods specifically written for testing purposes, the process of constructing the UI can proceed. The UI is the component of an application that users use to interact with it in order to carry out the work provided by the business logic of the application in the form of objects that, collectively, provide a service to users. The objects that implement the logic of the application are the *business objects* of the application in that they meet the business requirements of the application as determined by analysis of the business domain under consideration. The notion that the business objects of an application provide a *service* to users means that we can regard the business objects as the *server-side* or *server component* of an application.

Consider, for example, the classes of the themed application shown in Figure 3.1 on the next page.

Download free eBooks at bookboon.com

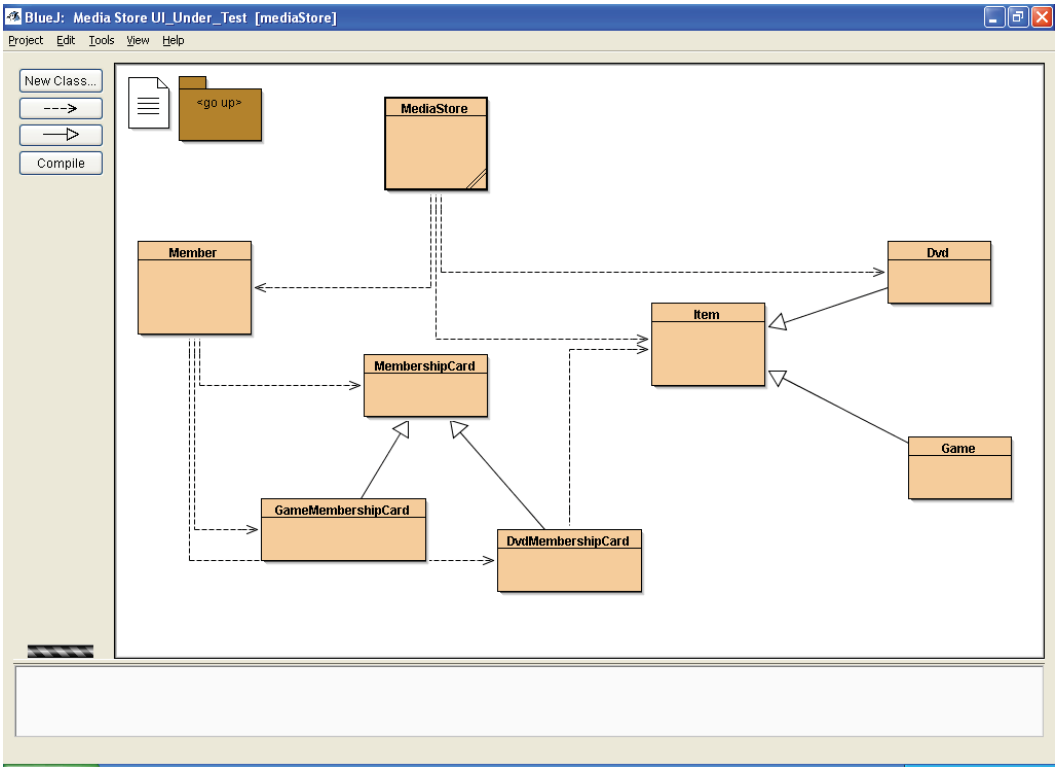


Figure 3.1 Classes of the themed application

The classes shown in Figure 3.1 encapsulate the business requirements of a realistic application that is referred to in this guide as the 'themed application'. The 'has a' and 'is a' associations shown in the figure imply that the class that represents the Media Store has a number of **Member** objects and **Item** objects of either the **Dvd** or **Game** type. Each member is provided with two virtual membership cards, one for recording the details about DVDs on loan (from the Media Store) and another for recording the details about games. Only the **DvdMembershipCard** class is defined in the version of the themed application shown in the figure; hence, the 'has a' link between this class and the **Item** class. For the purposes of the present discussion, the DVD membership card object implements methods to borrow and return DVDs. The state of each **Member** object, along with the graph of objects associated with **Member** – **DvdMembershipCard** and Dvd – are serialized to a data file. The classes shown in the figure comprise the server-side of the application.

The various methods of the classes of the application were tested (by the author of this guide) with, in the first instance, a number of **main** methods that test the correctness of card transactions when taking items out on loan and returning them. When testing was complete, the next stage in the development of the application was to construct the UI. The classes associated with the UI comprise the client-side of the application.

The client-side of the themed application is used – as we will see later in this chapter – to interact with the server-side. For example, there are buttons on the UI that call methods of the **DvdMembershipCard** class to borrow and return DVDs. Clearly, and rather obviously, there is a close coupling between the UI and the business objects with which it interacts. In other words, the server-side of the application is of no value without the client-side of the application and vice versa. In general, whilst the development and testing of

Download free eBooks at bookboon.com

the client and server sides of an application may proceed independently of one another, both sides of an application will eventually be brought together to comprise the complete application. Final testing will test that the UI operates correctly in its interaction with server-side objects.

The combination of the server and client 'sides' of the themed application into a single application results in what is known, in this case, as a *standalone* application in the sense that it runs on one computer: i.e. a standalone application runs in a single address space using a single Java Virtual Machine. Whilst it can readily be argued that the standalone version of the themed application is not realistic in that it is not designed to run on a (computer) network, its purpose in this guide is to provide a source of examples that illustrate how a number of fundamental Java programming concepts can be applied in the context of an application that is appropriate for learners. In short, the development of standalone applications is relevant from a learning point of view.

3.2 Client/Server Applications

The combination of the client and server sides of the themed application (discussed in the previous section) suggests that we can regard *any* standalone application as comprising client and server components that interact with one another. Perhaps not surprisingly, such applications are labelled with the term *client/server*.

Figure 3.2, shown on the next page, illustrates the architecture of a typical client/server application.



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

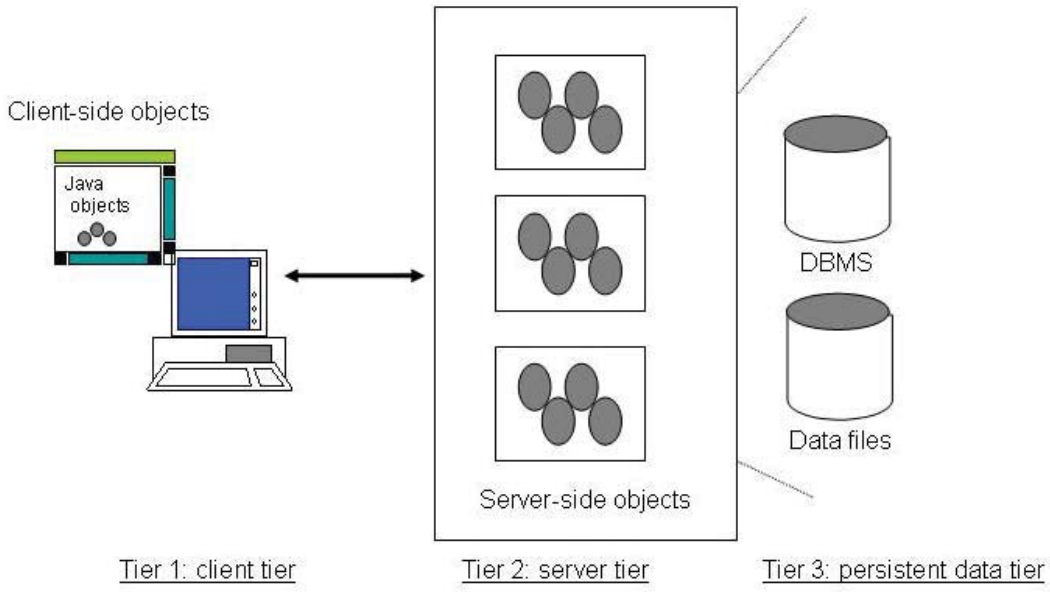


Figure 3.2 The architecture of a typical client/server application

Tier 1, the client tier, comprises the UI; tier 2, the server tier, comprises the business objects of the application; tier 3, the persistent data tier, comprises data storage components such as data files and database tables. In the case of the themed application, the third tier comprises a data file that stores an array of members of the Media Store. In a typical client/server application, the UI interacts with the server-side objects which, in turn, interact with the data tier. In other words, the UI interacts with the server-tier; the UI does not access the data tier directly.

The double-headed arrow (in the figure) that implies that the client and server tiers are connected is, in the case of a standalone application, an indication that the client and server tiers are deployed in the same address space. In the case of a networked application, the connection between the client and server tiers will either be an intranet or the Internet.

Now that we have placed user interfaces in the context of a client/server architecture, the next section moves on to find out how a UI is constructed.

3.3 The Construction of User Interfaces

In broad terms, there are two principal stages in the construction of a UI.

1. Organising UI components;
2. Activating UI components.

Stage 1 involves organising the layout and relative positions of UI components such as buttons, text fields and so on; stage 2 involves activating those components that are designed to call methods implemented in server-side classes in response to the user ‘operating’ the component.

Download free eBooks at bookboon.com

3.3.1 Organising User Interface Components: Some Examples

The principal task associated with graphical user interface (GUI) design involves making decisions about organising UI components (also known as *controls*) into what are known as *containers*.

- A container is an object that can contain components and other containers;
- a component is an object that is (usually) visible when the GUI is displayed;
- a *layout manager* is an object that is used to organise the position of components in their container.

The aim of the examples that follow is to illustrate a structured approach to GUI design. At the end of the set of examples, we will be in a good position to make a number of points about classes used to organise and contain components.

Example One

Figure 3.3 shown a simple GUI; its code is shown on the next page. The reader should read the comments in the code in order to get an initial understanding about how the GUI is created.



Click on the ad to read more

Download free eBooks at bookboon.com



Figure 3.3 A simple GUI

```
// Import packages that contain components, containers and layout managers.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// The GUI class inherits from a container of the JFrame type. The JFrame container will act
// as a top-level, outer container for any inner containers that are added to it.
public class SwingHelloGUI extends JFrame {
        // Declare instance variables.
        // A JTextField component can be used to enter text.
        private JTextField tf;
        // A JButton component is typically used as an active component that the user presses
        // in order to trigger an action to take place.
        private JButton b;
        // A JPanel is an inner container that can be placed inside a JFrame.
        private JPanel p;
        // The constructor does the work of rendering the GUI.
        public SwingHelloGUI( ) {
                // Call the parent constructor.
                super( "Hello GUI" );
                // Initialise tf and p.
                tf = new JTextField( );
                p = new JPanel( );
                // Set the layout manager of the panel to be a grid of two rows and one column.
                p.setLayout( new GridLayout( 2, 1 ) );
                // Initialise b.
                b = new JButton( "Click me..." );
                // Add the button to the first position in the grid.
                p.add( b );
                // Add the text field to the second position in the grid.
                p.add( tf );
                // Add the panel to the current (JFrame) object.
                this.add( p );
                // The pack method ensures that the JFrame is sized to fit the preferred size of
                // its components.
                this.pack( );
```


Download free eBooks at bookboon.com

```
            // Show the JFrame when the constructor is called.
            this.setVisible( true );
        } // End of constructor.
    } // End of class definition.
```

The **main** method is omitted from the code for the sake of brevity. The purpose of **main** is to call the constructor for the GUI; the result of executing **main** is shown in Figure 3.3 above.

The default layout manager for a **JPanel** is **FlowLayout**; this layout manager arranges components in a directional flow from left to right in a container that uses it. As can be seen from the code for Figure 3.3, the default layout has been changed to a grid using the following statement:

```
p.setLayout( new GridLayout( 2, 1 ) );
```

The constructor for a **GridLayout** object is passed to the panel's **setLayout** method in order to change the panel's layout manager to be a grid of two rows and one column.

> When components are added to a grid, they are added in the following order: first row from left to right across the columns, second row from left to right across the columns and so on to the end of the last row.

The next part of the code adds the text field and the button to the **JPanel** in that order, with the result shown in Figure 3.3.

```
// Add the button to the first position in the grid.
p.add( b );
// Add the text field to the second position in the grid.
p.add( tf );
```

These two statements populate the **JPanel** with two components in the correct order. The completed **JPanel** is now ready to be added to its outer container, the **JFrame**, as follows:

```
// Add the panel to the current (JFrame) object.
this.add( p );
```

The populated **JFrame** is now ready to be displayed.

Despite its apparent simplicity, the example above illustrates that there are a number of steps that are associated with GUI design. Thus, the overall approach to GUI design can be summarised as comprising the steps summarised on the next page.

Download free eBooks at bookboon.com

1. Import packages;
2. instantiate the top-level, outer container;
3. instantiate inner containers as required;
4. add controls to inner containers;
5. add inner containers to the top-level container;
6. display the top-level container;
7. make the GUI thread safe.

The example explained above illustrates steps 1 to 6; we will return to Step 7 in due course.

The important point to make about the example and the list of steps is that the overall conception of the *look* of a GUI is a top-down activity, whilst the population of inner containers is a bottom-up activity. In other words, inner containers must be populated in their correct sequence *before* they are added to their top-level container.

An example of one kind of mistake in the bottom-up aspect of rendering a GUI is displayed in the screen shot shown in Figure 3.4; this arises as a result of adding the text field and button to the panel in the wrong order.

SIMPLY CLEVER
ŠKODA


Do you like cars? Would you like to be a part of a successful brand?
As a constructer at ŠKODA AUTO you will put great things in motion. Things that will ease everyday lives of people all around Send us your CV. We will give it an entirely new new dimension.
Send us your CV on
www.employerforlife.com

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



Figure 3.4 Adding components in the incorrect sequence

The code that gives rise to the GUI shown in Figure 3.4 is

```
p.add( tf );
p.add( b );
```

It is left as an exercise for the reader to analyse the code for the remaining examples, in the knowledge of how each GUI looks,. The **main** method that renders the GUI in each example is omitted for the sake of brevity.

Example Two

Figure 3.4 shows a **JFrame** containing two buttons arranged by a **FlowLayout** manager.

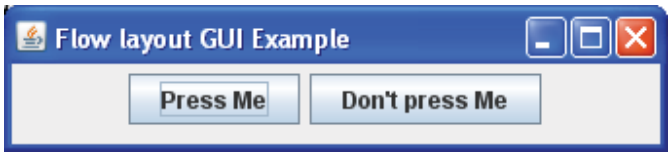


Figure 3.4 The FlowLayout manager

The default layout manager for a **JFrame** is **BorderLayout** (see a later example). To achieve the GUI shown in Figure 3.4, the layout manager for the top-level **JFrame** container is changed to **FlowLayout**, as shown in the code for the GUI that follows next.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FlowLayoutExample {

    private JFrame f;
    private JButton b1;
    private JButton b2;

    private FlowLayoutExample( ) {
```


Download free eBooks at bookboon.com

```
        f = new JFrame( "Flow layout GUI Example" );
        b1 = new JButton( "Press Me" );
        b2 = new JButton( "Don't press Me" );
        f.setLayout( new FlowLayout(  ) );
        f.add( b1 );
        f.add( b2 );
        f.pack( );
        f.setVisible( true );

    } // End of constructor.

} // End of class definition.
```

The GUI shows that **FlowLayout** manager adds components to their container from left to right and, if necessary, from top to bottom. The **FlowLayout** manager aligns components centrally, as can be seen from Figure 3.4.

Example Three

Figure 3.5 shows a **JFrame** that uses a **BorderLayout** manager to arrange a number of buttons.



Figure 3.5 The BorderLayout manager

Given that the default layout manager for a **JFrame** is **BorderLayout**, the buttons are added directly to their top-level container – there being no inner containers in this example – to one of five positions, as shown in the following code.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BorderLayoutExample {

    private JFrame f;
    private JButton bn;
    private JButton bs;
    private JButton be;
    private JButton bw;
    private JButton bc;
```


Download free eBooks at bookboon.com

```
    private BorderLayoutExample ( ) {

        f = new JFrame( "Border layout GUI Example" );
        bn = new JButton( "Button North" );
        bs = new JButton( "Button South" );
        be = new JButton( "Button East" );
        bw = new JButton( "Button West" );
        bc = new JButton( "Button Centre" );
        f.add( bn, BorderLayout.NORTH );
        f.add( bs, BorderLayout.SOUTH );
        f.add( be, BorderLayout.EAST );
        f.add( bw, BorderLayout.WEST );
        f.add( bc, BorderLayout.CENTER );
        f.pack( );
        f.setVisible( true );

    } // End of constructor.

} // End of class definition.
```

The GUI shows that the **BorderLayout** manager adds components to a position determined by one of the static fields of the **BorderLayout** class.

Example Four

Figure 3.6 below displays an example of a **JFrame** that uses a **GridLayout** manager to arrange a number of buttons. The GUI uses a grid with more cells than the one used in Example One above.

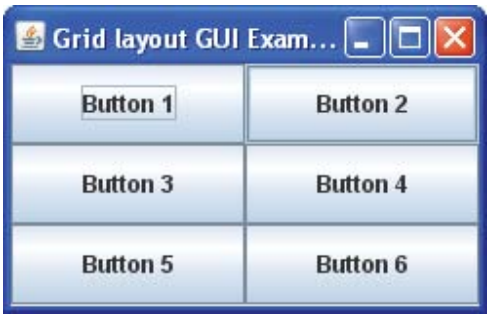


Figure 3.6 The GridLayout manager

The GUI shows that components are added to grid cells from left to right and top to bottom.

Example Five

Figure 3.7 (on the next page) shows an example of a slightly more complex **JFrame** that uses more than one inner container to arrange the components of the GUI.

Download free eBooks at bookboon.com

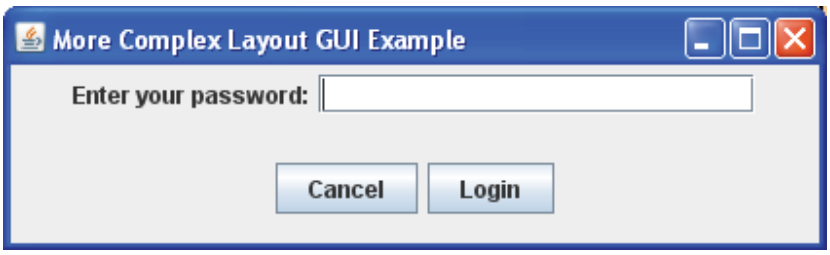


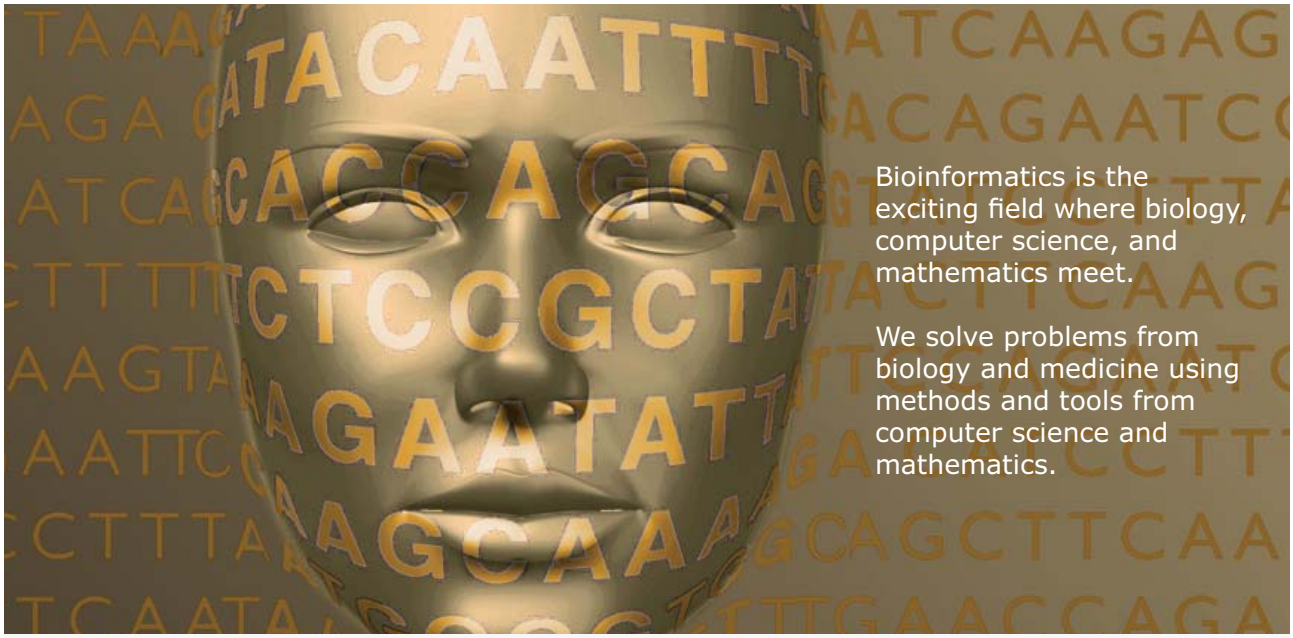
Figure 3.7 Containers within containers

The code for the GUI is next. The reader should note carefully how inner containers are populated and added to the top-level container.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

UPPSALA UNIVERSITET

Develop the tools we need for Life Science
Masters Degree in Bioinformatics



Read more about this and our other international masters degree programmes at www.uu.se/master

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

```
public class ComplexLayoutExample {
        private JFrame f;
        private JButton b1;
        private JButton b2;
        private JLabel l;
        private JTextField tf;
        private JPanel upperPanel;
        private JPanel lowerPanel;

        private ComplexLayoutExample ( ) {
                f = new JFrame( "More Complex Layout GUI Example" );
                upperPanel = new JPanel( );
                lowerPanel = new JPanel( );
                upperPanel.add( l = new JLabel("Enter your password:" ) );
                upperPanel.add( tf = new JTextField( 20 ) );
                lowerPanel.add( b1 = new JButton( "Cancel" ) );
                lowerPanel.add( b2 = new JButton( "Login" ) );
                f.setLayout( new GridLayout( 2,1 ) );
                f.add( upperPanel );
                f.add( lowerPanel );
                f.pack( );
                f.setVisible( true );
        }  // End of constructor.
} // End of class definition.
```

The label and the text field are added to their (**JPanel**) container using the default **FlowLayout** manager for the **JPanel** and the two buttons are similarly added to their own container. The default **BorderLayout** manager for the top-level container (a **JFrame**) is changed to use a **GridLayout** to arrange the two inner containers as shown in the figure. The purpose of adding two separate containers to the grid is to achieve the desired look for the GUI.

If all four components are added to a single container which, in turn is added to the top-level container, the GUI would not achieve the required look. This is illustrated in Figure 3.8 below.

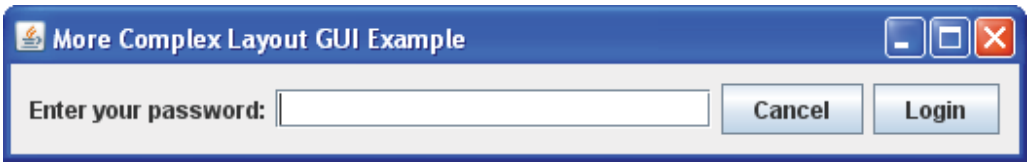


Figure 3.8 The incorrect look for the GUI of Example Five

The required look for the GUI of Example Five is two rows of components. One way to achieve this look is to use two inner containers as shown in the code for the GUI shown in Figure 3.7.

Download free eBooks at bookboon.com

The examples in this sub-section illustrate a few of the many UI components available in the **javax.swing** package. The reader is referred to the API for details about other components and their class hierarchy. There are other layout managers available in the **javax.swing** and **java.awt** packages; again, the reader is referred to the Java API for further details.

3.3.2 Summary of Components and Layout Managers

The examples in sub-section 3.3.1 suggest that we are now in a position to enhance the list of steps that is proposed at the end of Example One above. The enhanced list of steps or activities that are typically included in GUI design is shown on the next page.

UNIVERSITY OF COPENHAGEN



Copenhagen Master of Excellence



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

Come to Copenhagen - *and aspire!*

Apply now at
www.come.ku.dk

Click on the ad to read more

Download free eBooks at bookboon.com

1. Import packages;
2. instantiate the top-level, outer container;
   - typically a JFrame object; (a JFrame is a visible component when it is displayed)
   - the default layout manager for a JFrame is BorderLayout
3. change from the default layout manager to another if required
4. decide on how the outer container will be closed (see below)
5. decide what the look and feel of the GUI should be (see below)
   - instantiate inner containers as required;
   - typically a number of JPanel objects; (a JPanel is not visible when it is displayed)
   - panels can be added to panels; the final, outer panel is often added to the top-level JFrame
   - the default layout for a JPanel is FlowLayout
   - change from the default layout manager to another if required
   - add UI components to their inner containers;
6. add the outer panel (containing inner panels) to the top-level container;
7. display the top-level container;
8. make the GUI thread safe (see below).

Step 3: closing the top-level container

One of the methods of the **JFrame** class is used to set an action to take when the user closes the frame, as in the following statement:

frame.setDefaultCloseOperation(JFrame.<action>);

where action is one of

EXIT_ON_CLOSE
DO_NOTHING_ON_CLOSE
HIDE_ON_CLOSE (default)
DISPOSE_ON_CLOSE

Step 4: decide on the look-and-feel of the GUI

Swing can specify a uniform look-and-feel (L & F) across platforms by means of the static **setLookAndFeel** method of the **UIManager** class, as shown on the next page.

Download free eBooks at bookboon.com

```
try
{
        UIManager.setLookAndFeel( <LookAndFeel> );
}
catch( Exception e ) { }
```

where **LookAndFeel** can be one of

UIManager.getCrossPlatformLookAndFeelClassName( )

UIManager.getSystemLookAndFeelClassName( )

“javax.swing.plaf.metal.MetalLookAndFeel”

“com.sun.java.swing.plaf.windows.WindowLookAndFeel”

“com.sun.java.plaf.motif.MotifLookAndFeel”

Otherwise, the **UIManager** sets a default look-and-feel.

The reader is referred to the API for the **javax.swing** package for further details about closing frames and setting the look-and-feel.

Step 9: make the GUI thread safe

The Java Virtual Machine allows an application to have multiple *threads* of execution running concurrently. The on-line Java tutorial recommends that a GUI executes in its own thread (of execution). This recommendation is reflected in the code for the **main** method that calls the constructor for a GUI. (Threads in Java programmes are discussed in the final chapter of this guide.)

Step 9 is included in a **main** method, usually in a separate, lightweight class with the generalised class definition shown on the next page.

Download free eBooks at bookboon.com

```
public class RenderTheGui {

    public static void main( String[ ] args ) {

        // The next statement is a call to the invokeLater method of the
        // SwingUtilities class of the javax.swing package.
        // The invokeLater method takes a Runnable object as an argument.  In
        // essence, the purpose of the Runable object is to initialise the GUI and
        // schedule it for execution in a thread.
        javax.swing.SwingUtilities.invokeLater( new Runnable( ) {

            // start of anonymous class definition of the Runnable object
            // call the thread's run method
            public void run( ) {

                // call the GUI's constructor
                MySwingClass gui = new MySwingClass( );

            } // end of run method

        } // end of Runnable class definition
        ); // end of call to invokeLater method

    } // end of main

} // end class definition
```

Although there is quite a lot going on in **main**, it is not important that the reader understands its code in full yet. For the present purposes, it is important to note that an *anonymous* **Runnable** object has been created by calling its constructor *within* the brackets of the **invokeLater** method. In effect, the purpose of the class called **RenderTheGui** is to make it thread safe by scheduling the thread's **run** method to execute in its own thread. (We will meet anonymous classes again in a later section of this chapter.)

Before we move on to explain the second principal stage involved in the construction of a GUI – activating UI components – it will be beneficial to the reader to examine briefly one of the widely-used *visual* approaches to the organisation of UI components.

Download free eBooks at bookboon.com

3.4 A Visual Approach to GUI Design

The IDE known as BlueJ referred to in this and previous chapters of this guide was used to write, edit, compile and run the example programmes presented in this guide, including the Swing examples shown in Section 3.3.1. However the look of a GUI developed in a BlueJ project is only evident when its **main** method is executed. Up to the point of calling **main**, the developer relies on his or her skill in following the steps outlined in Section 3.3.2 to render the correct look of the GUI. Thus, the process of display, edit and run is an iterative one in order to render a GUI correctly. It is in this sense that we can think of coding GUIs as *coding by hand*. Coding a GUI by hand is recommended by the author of this guide as the most effective way for a learner to practice the steps set out in Section 3.3.2.

When the learner gains experience in GUI design, he or she can progress to using a *visual approach* to GUI design. A visual approach automates the organisation of components, leaving the developer to concentrate on writing the code to activate UI components (see Section 3.5 later in this chapter).

One such visual design is provided by the IDE known as NetBeans™ (available from http://www.netbeans.org/). NetBeans™ is a highly sophisticated, professional-level IDE that, amongst other things, provides features that facilitate visual design for Java Swing GUIs.

The screen shot displayed in Figure 3.9 on the next page shows a NetBeans™ project that includes a single class that extends form the **JFrame** class.



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

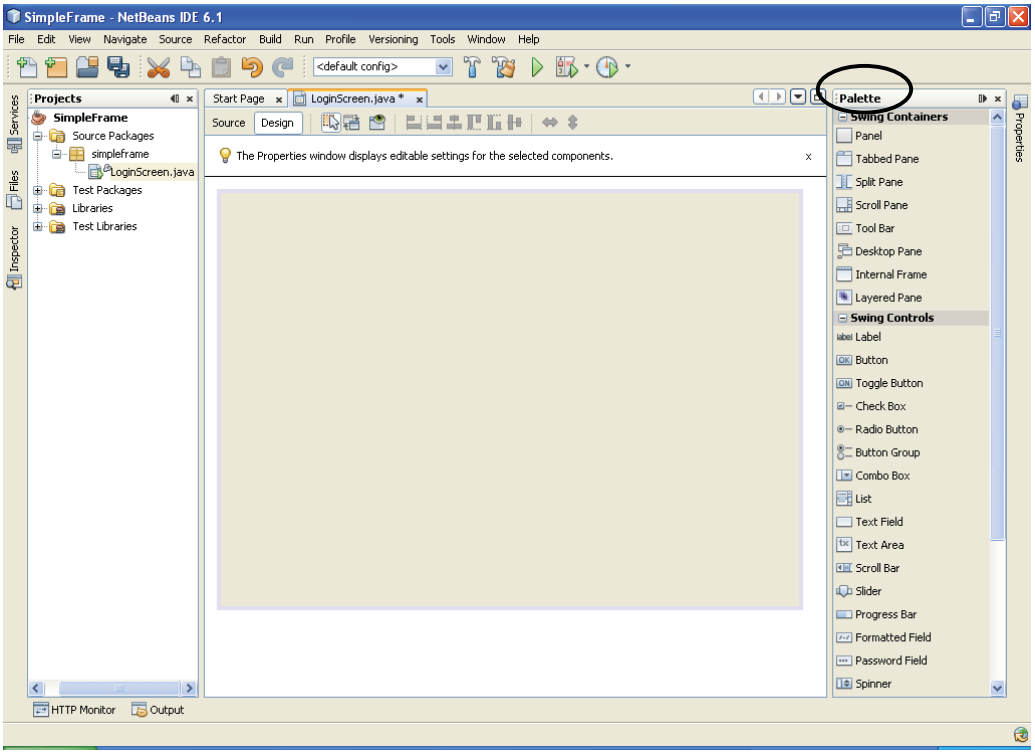


Figure 3.9 A NetBeans™ project containing a 'visual' JFrame

The central pane of the screen displays the design surface, with the palettes of Swing containers and components on the right-hand side.

Components can be dragged from the palette and dropped onto the design surface, as shown in the screen shot, shown on the next page, of a simple login screen for the themed application.

Download free eBooks at bookboon.com

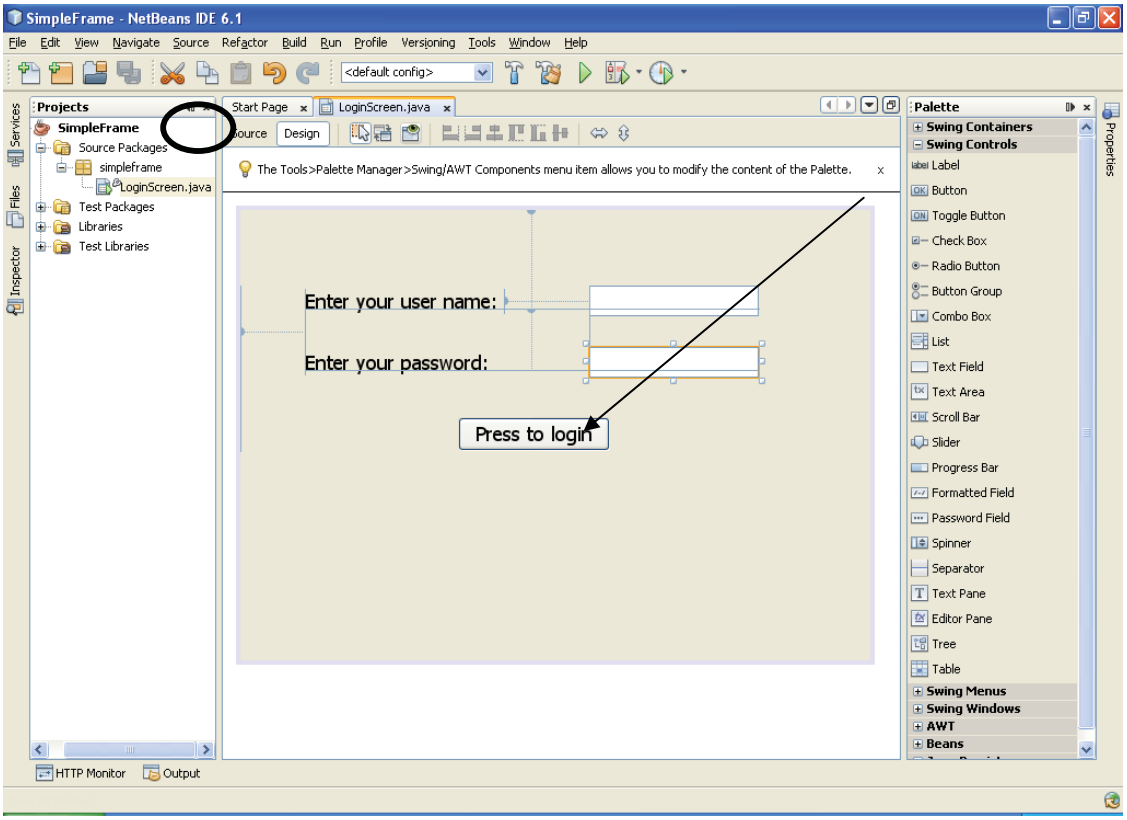


Figure 3.10 Visual design of a simple GUI

Selecting the Source tab (circled in the figure) displays the automatically-generated source code for the GUI. This file includes **main**, rather than placing it in a separate class, and – as expected – the following variable declarations:

```
private javax.swing.JButton jButton1;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JTextField jTextField1;
private javax.swing.JTextField jTextField2;
```

The visual approach adopted by NetBeans™ uses the **GroupLayout** manager that is principally intended for use in IDEs. The principal benefit of using **GroupLayout** is that UI components can be moved around the design surface as required.

In the example shown in Figure 3.10, UI components are added directly to the top-level container in that they are added directly to the design surface. It is equally easy to add Swing containers to the design surface so that they can be subsequently populated by UI components.

The next screen shot shows a **JFrame** containing two empty **JPanel**s.

Download free eBooks at bookboon.com

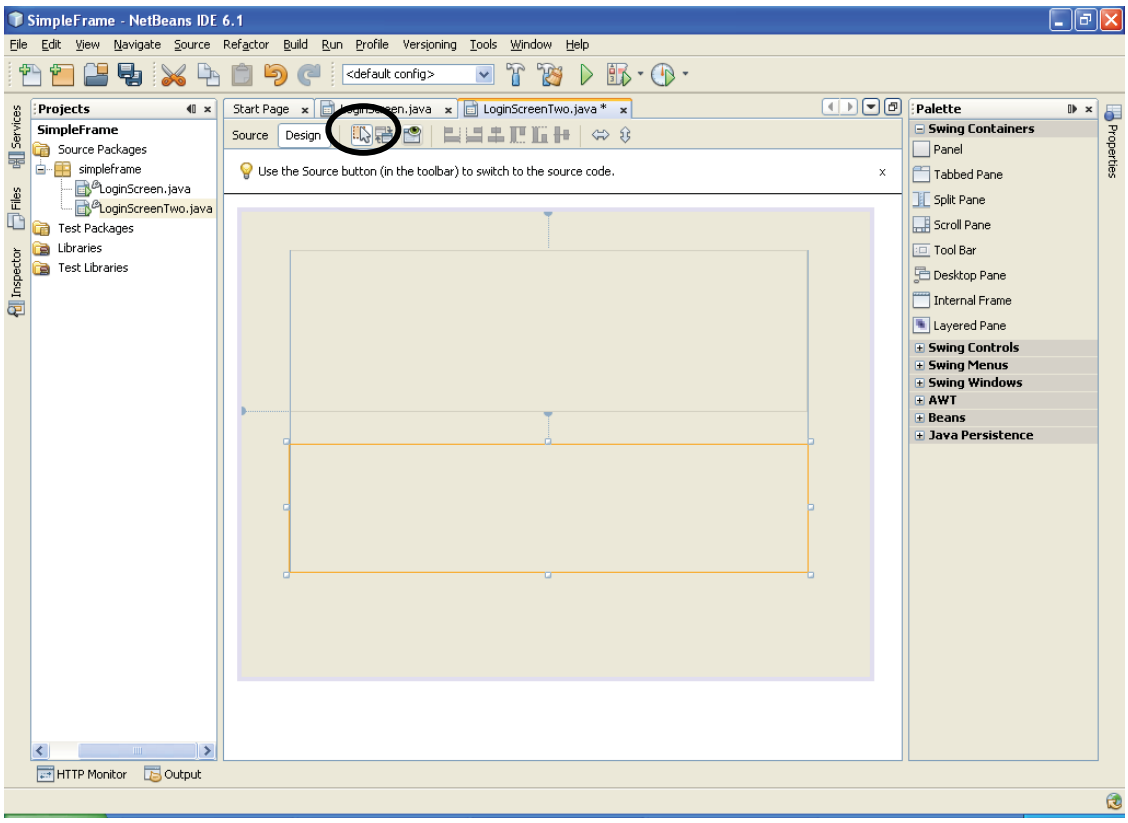


Figure 3.11 A JFrame containing two empty JPanels

Selecting the review button (circled in the figure) shows that panels are opaque when the GUI is displayed, as shown in the next screen shot.

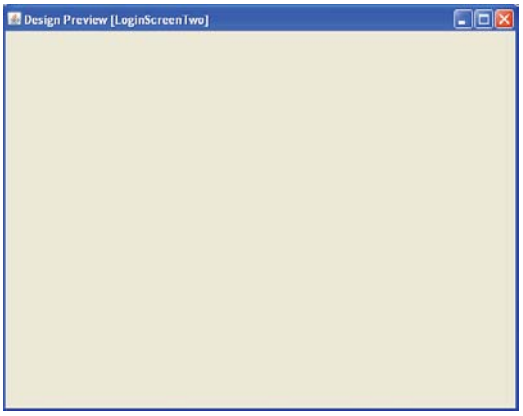


Figure 3.12 The JFrame shown in Figure 3.11

Components could be dragged and dropped onto the containers shown in Figure 3.11 to construct a GUI that comprises components contained in two inner containers that have been added to the top-level **JFrame**.

Download free eBooks at bookboon.com

The main benefit of organising containers and components visually is that the code of the GUI is automatically generated; this leaves the developer to concentrate their efforts on writing code that activates those components that are meant to trigger some action in the context of an application. The second principal stage involved in the construction of a GUI – activating UI components – is explored in the next section.

3.5 Activating User Interface Components

Activating a UI component, such as a button, to carry out processing in the context of an application involves creating an object that represents a user-generated event and writing one or more *event-handler* methods that respond to such events. The fundamental aspects of event handling are explored by means of examples that are presented later in this section. Following the examples, a number of points about event handling are made.

Figure 3.13 displays a hand-coded GUI that is used for existing members of the Media Store – the themed application – to login before they can use their membership card.

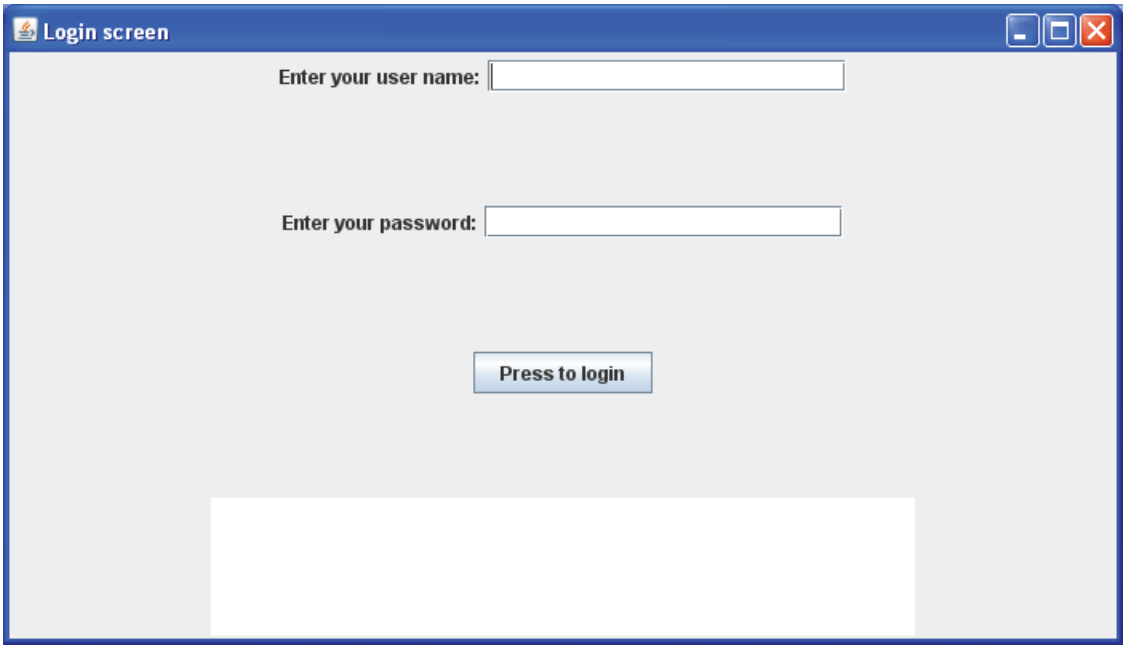


Figure 3.13 The login screen for the themed application

The code for the GUI is shown on this and the next page.

Download free eBooks at bookboon.com

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class LoginScreen {

        private JFrame f;
        private JButton login;
        private JLabel usernameLabel;
        private JLabel passwordLabel;
        private JTextField usernameTextField;
        private JPasswordField passwordField;
        private JTextArea messages;
        private JPanel usernamePanel;
        private JPanel passwordPanel;
        private JPanel buttonPanel;
        private JPanel messagesPanel;

        public LoginScreen( ) {

                f = new JFrame( "Login screen" );
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.setLayout( new GridLayout( 4,1 ) );
```


Trust and responsibility

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

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

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



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

nne pharmaplan®

Click on the ad to read more

Download free eBooks at bookboon.com

```
        usernamePanel = new JPanel( );
        usernamePanel.add( usernameLabel =
                        new JLabel( "Enter your user name:" ) );
        usernamePanel.add( usernameTextField =
                        new JTextField( 20 ) );

        passwordPanel = new JPanel( );
        passwordPanel.add( passwordLabel =
                        new JLabel( "Enter your password:" ) );
        passwordPanel.add( passwordField =
                        new JPasswordField( 20 ) );

        buttonPanel = new JPanel( );
        buttonPanel.add( login =
                        new JButton( "Press to login" ) );

        messagesPanel = new JPanel( );
        messages = new JTextArea( 10, 40 );
        messagesPanel.add( messages );

        f.add( usernamePanel );
        f.add( passwordPanel );
        f.add( buttonPanel );
        f.add( messagesPanel );
        f.pack( );
        f.setSize( 700, 400 );
        f.setVisible( true );

    }  // end of constructor

}  // end of class definition
```

The GUI comprises four panels of components added to a frame's grid of size four rows by one column; comments are omitted from the code. It is left as an exercise for the reader to analyse the code, given the look of the GUI, in order to understand how the GUI is constructed. The third row (from the top) of the GUI comprises a button component, the purpose of which is to enable the member of the Media Store to login. The fourth row from the top comprises a text area, the purpose of which is to output messages that inform the user if he or she has logged in correctly or not.

The fundamental principal of handling user-generated events is to write the class definition for the object that will respond to such events. This category of object is known as a *listener*. The examples that follow use the GUI shown in Figure 3.13 to illustrate a number of ways to provide code to respond to a user who logs in and presses the button.

Download free eBooks at bookboon.com

Example One: an external listener class

In this example, the listener class is separate from the class that displays the GUI. Given that the listener class is designed to respond to button events by outputting messages in the text area of the GUI, the listener class requires access to the text area component. This is achieved by providing a **get** method for the text area in the GUI class and passing the object reference of the GUI to the listener class.

The additional code for the GUI shown in Figure 3.13 – called **ExternalEventHandler** in this example - adds an object that implements the **ActionListener** interface to the button *before* it is added to its panel, as shown on the next page.

This e-book
is made with
SetaPDF
SETASIGN

PDF components for PHP developers
www.setasign.com

Download free eBooks at bookboon.com

Click on the ad to read more

```
buttonPanel = new JPanel( );
        login = new JButton( "Press to login" );
        // 'Register' the button with its listener class.
        login.addActionListener( new ButtonHandler( this ) );
        buttonPanel.add( login );
```

The code for getTextArea is simply:

```
        public JTextArea getTextArea( ) {
                return messages;
        }
```

The **addActionListener** method 'adds' an object that implements the **ActionListener** interface to the button. **ActionListener** is the listener interface for receiving button events. The listener class that processes button events – called **ButtonHandler** in this example - implements this interface and provides code for the **actionPerfomed** method declared in the interface. The object that is created from this class is 'registered' with the button component by invoking its **addActionListener** method.

The code of the **ButtonHandler** class is as follows.

```
        import java.awt.*;
        import java.awt.event.*;
        import javax.swing.*;

        public class ButtonHandler implements ActionListener {

                private ExternalEventHandler gui;

                public ButtonHandler( ExternalEventHandler gui ) {
                        this.gui = gui;
                }

                public void actionPerformed( ActionEvent e ) {
                        gui.getTextArea( ).setText( "Output messages here." );
                }

        } // End of class definition.
```

The object's **actionPerformed** method is invoked when a button event occurs. In this case, a simple message is output to the GUI's text area.

Download free eBooks at bookboon.com

The result of pressing the login button is shown in the next screen shot.

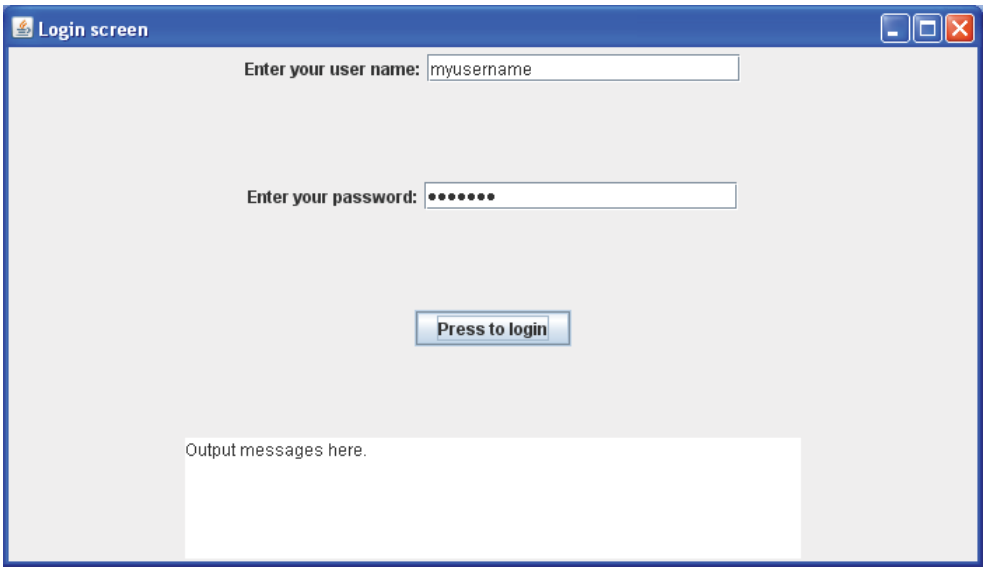


Figure 3.14 A button event for Example One

In the example, and the ones that follow, the body of the **actionPerformed** method declared in the **ActionListner** interface is kept simple in order to illustrate how the listener class is defined. In the themed application, the body of the **actionPerformed** method reads the information entered into the two text fields of the GUI and uses this to scan the array of existing members to find the member who has logged in. A suitable message is output to the text area if the member has logged in correctly or cannot be found in the array. In general, the body of an **actionPerformed** method is likely to comprise many lines of code, depending on how much work it is required to do.

Example Two: the GUI class is its own listener

In this example, the GUI class is its own listener; i.e. a separate listener class is not provided. The code associated with adding the button to its panel is:

```
buttonPanel = new JPanel( );
login = new JButton( "Press to login" );
// Register the current object with the button.
login.addActionListener( this );
buttonPanel.add( login );
```

The **actionPerformed** method is a member of the GUI class; the code for the method is next.

```
public void actionPerformed( ActionEvent e ) {
        messages.setText( "Output messages here." );
}
```


Download free eBooks at bookboon.com

In the case of the GUI class acting as its own listener, it should be noted that the body of the **actionPerformed** method has direct access to private instance variables of the class. Hence, the body of **actionPerformed** has access to the variable with the identifier **messages**.

The result of pressing the login button is shown in the next screen shot; the result of pressing the button is the same as that shown in Figure 3.14.

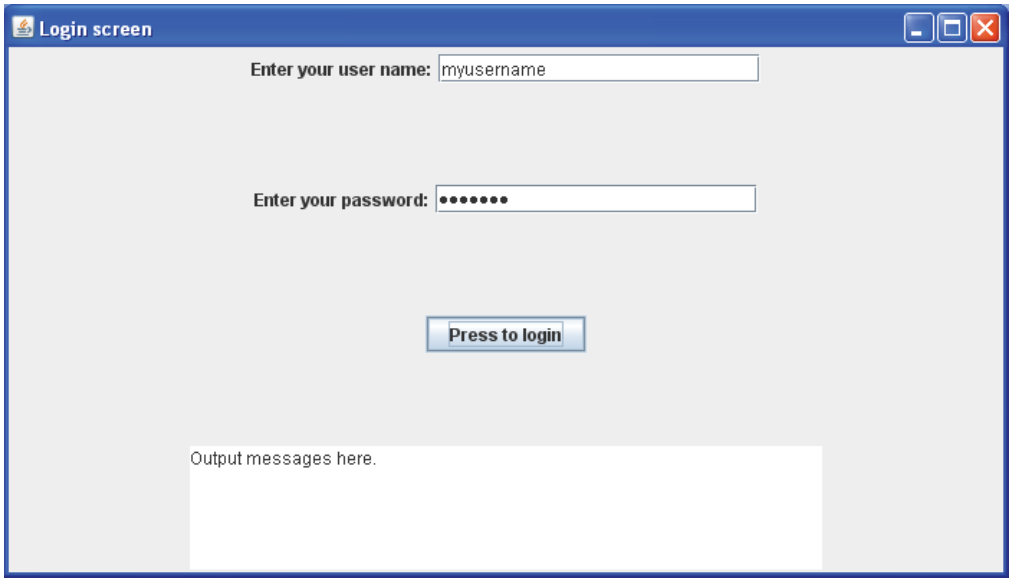


Figure 3.15 A button event for Example Two

FOSS

Sharp Minds - Bright Ideas!

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

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

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

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

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

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

Click on the ad to read more

Download free eBooks at bookboon.com

Example Three: event handling using an inner class

In this example, an *inner class* is used as the listener class instead of an external class (as in Exercise One). An inner class is a complete class definition contained entirely inside an enclosing class definition and is regarded as a member of the enclosing class.

The section of code that adds the button to its panel is shown below:

```
buttonPanel = new JPanel( );
login = new JButton( "Press to login" );
// Register the inner class as the handler class.
login.addActionListener( new InnerButtonHandler( ) );
buttonPanel.add( login );
```

The class definition of the inner class is:

```
class InnerButtonHandler implements ActionListener {

        public void actionPerformed( ActionEvent e ) {
                messages.setText( "Output messages here." );
        }

} // end of inner class definition
```

It should be noted that the inner class definition is not declared to be public; this means that it can be stored in the same file that stores the enclosing class – **InnerClass.java** in this exercise – and is not accessible from outside the outer class. In addition, members of an inner class have direct access to private instance variables of the outer class, as the code for the **actionPerformed** method shows.

The result of pressing the login button is shown in the next screen shot; the result of pressing the button is the same as for exercises one and two.

Download free eBooks at bookboon.com

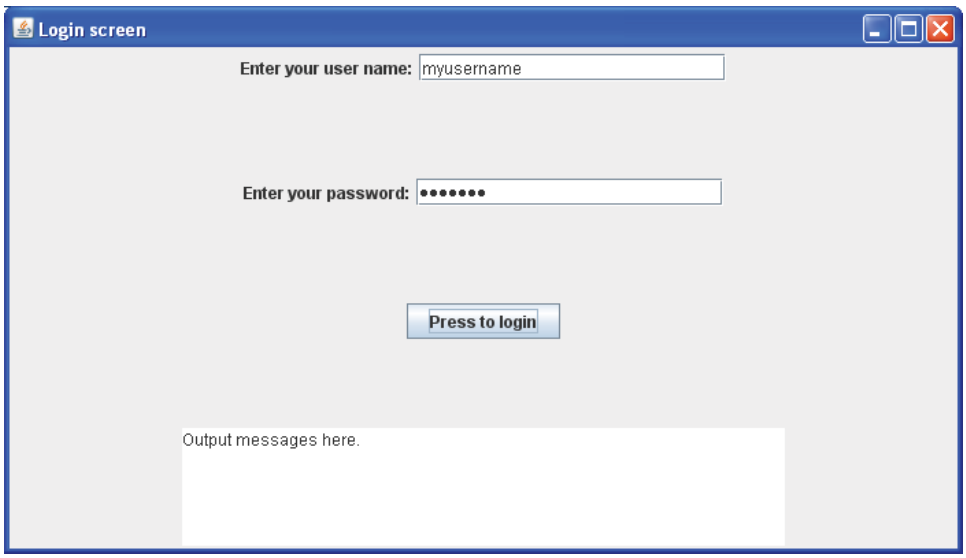


Figure 3.16 A button event for Example Three

Example Four: event handling using an anonymous class

In this example, a category of class known as an *anonymous class* is declared as the listener for the button. An anonymous class is somewhat similar to an inner class, except that it is created and used all at once. The class is 'anonymous' in the sense that it isn't given an explicit object reference; rather, it is created where it is needed. The entire class definition of an anonymous class is contained within the scope of an argument of a method, as shown in the code that adds the button to the panel (for Example Four) that is displayed next.

```
buttonPanel = new JPanel( );
login = new JButton( "Press to login" );
login.addActionListener( new ActionListener( ) { // start of anonymous class definition
        public void actionPerformed( ActionEvent e ) {
                messages.setText( "Hello world" );
        }
} ); // end of anonymous class, then end of addActionListener method
buttonPanel.add( login );
```

The argument to the **addActionListener** method is an object that is an **ActionListener**, i.e. it an object of a class that *implements* the **ActionListener interface**. The invocation of the method **addActionListener** on the button (in the code above) uses special Java syntax to define an anonymous, inner class to create an object of the class that is passed as an argument to the **addActionListener** method. The argument passed to **addActionListener** is a call to **ActionListener( )**; this call begins the definition of the anonymous class that implements the **ActionListener** interface and is shorthand for

Download free eBooks at bookboon.com

```
public class MyHandler implements ActionListener {
```

The closing brace of the anonymous class definition is followed by the closing bracket that holds the argument of the `addActionListener` method call on the button. Given the nature of the special syntax, the reader should be aware that great care must be taken when coding anonymous classes.

As has been pointed out earlier, it is likely that the code for the `actionPerformed` method contains many lines of code, all of which will be included in the body of the definition of the anonymous class which is, in turn, passed as a parameter to the `addActionListener` method of the button. This usually makes it difficult to read the code for event handlers that are members of anonymous classes. Despite this potential disadvantage, an anonymous class has access to the private variables of the enclosing class, as shown in the next statement.

```
messages.setText( "Hello world" );
```

An improved way to code an anonymous listener class is to invoke a method of the enclosing class, as shown in an alternative version of the code that adds the button to the panel (in Example Four) as follows.

```
buttonPanel = new JPanel( );
login = new JButton( "Press to login" );
login.addActionListener( new ActionListener( ) { // start of anonymous class definition
        public void actionPerformed( ActionEvent e ) {
                // delegate event handling to a method of the enclosing class
                buttonAction( e );
        }
} ); // end of anonymous class, then end of addActionListener method
buttonPanel.add( login );
```

The code for the body of the `buttonAction` method is

```
public void buttonAction( ActionEvent e ) {
        // event handling code
        messages.setText( "Hello world" );
}
```

Both versions of Exercise Four produce the same result, as illustrated in the next screen shot.

Download free eBooks at bookboon.com

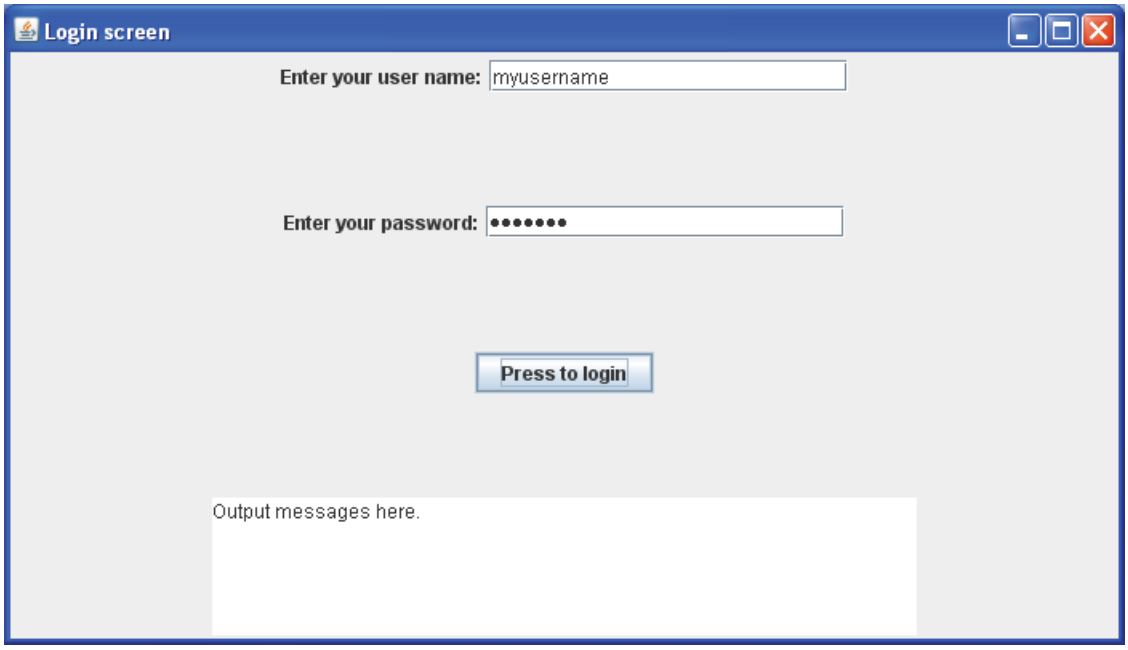


Figure 3.17 A button event for Example Four

The advantage of the second approach illustrated in Example Four is to pass the responsibility for event handling code to a method of the enclosing class. The developer provides event handling code for this method, rather than embedding it within the definition of an anonymous class.



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

Example Five

The alternative approach when using an anonymous class illustrated in Example Four is adopted by the visual design features of NetBeans™.

Double-clicking on the button of the GUI discussed in Section 3.4, shown in design view in the next screen shot automatically inserts event handling code.

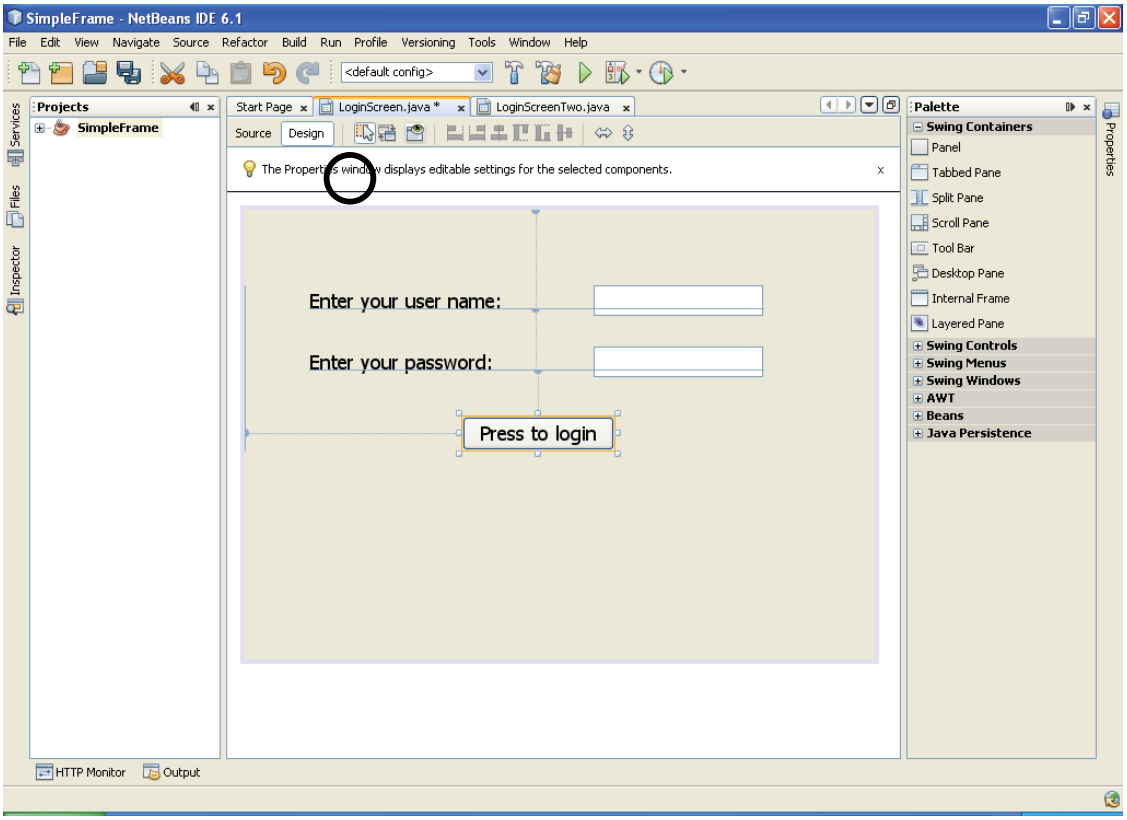


Figure 3.17 The design view of the login screen of the themed application

The event handler method inserted by NetBeans™ is highlighted in the screen shot shown on the next page.

Download free eBooks at bookboon.com

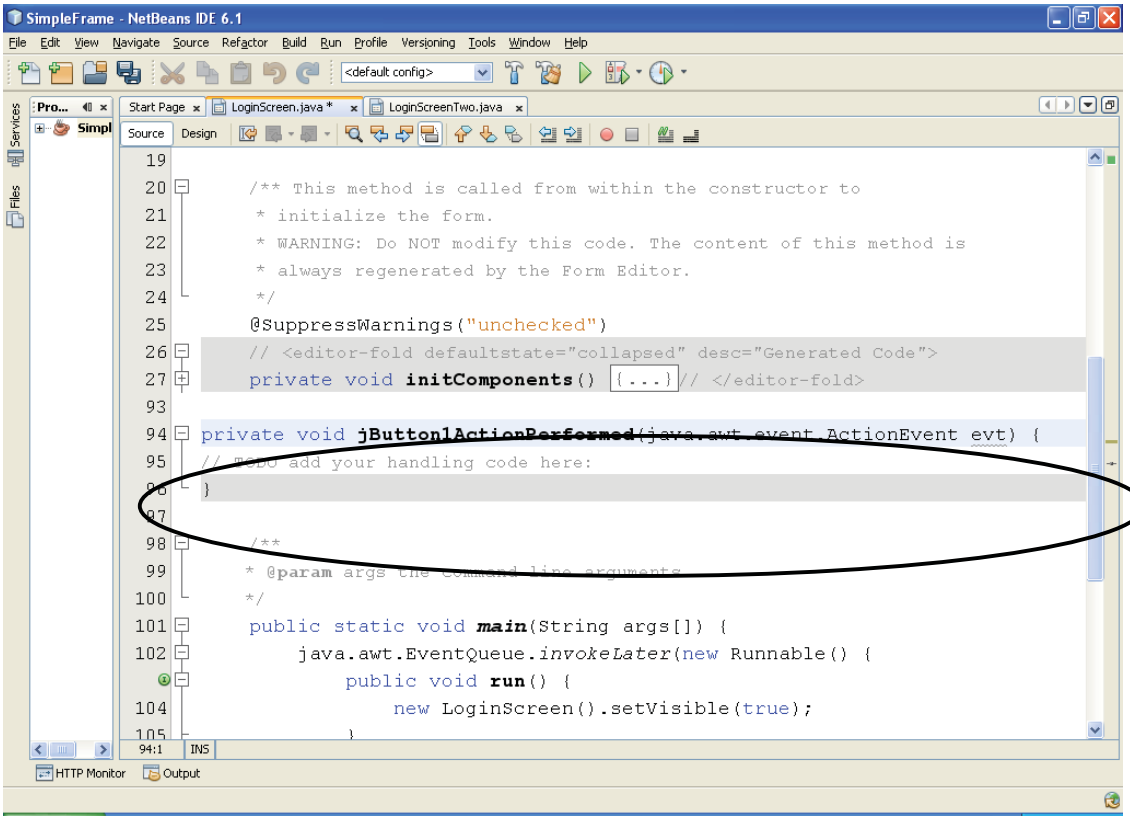


Figure 3.18 The event handling method

Further examination of the code that is automatically inserted by MetBeans™ shows that the method is called from an anonymous class, as highlighted in the screen shot that is displayed on the next page.

Download free eBooks at bookboon.com

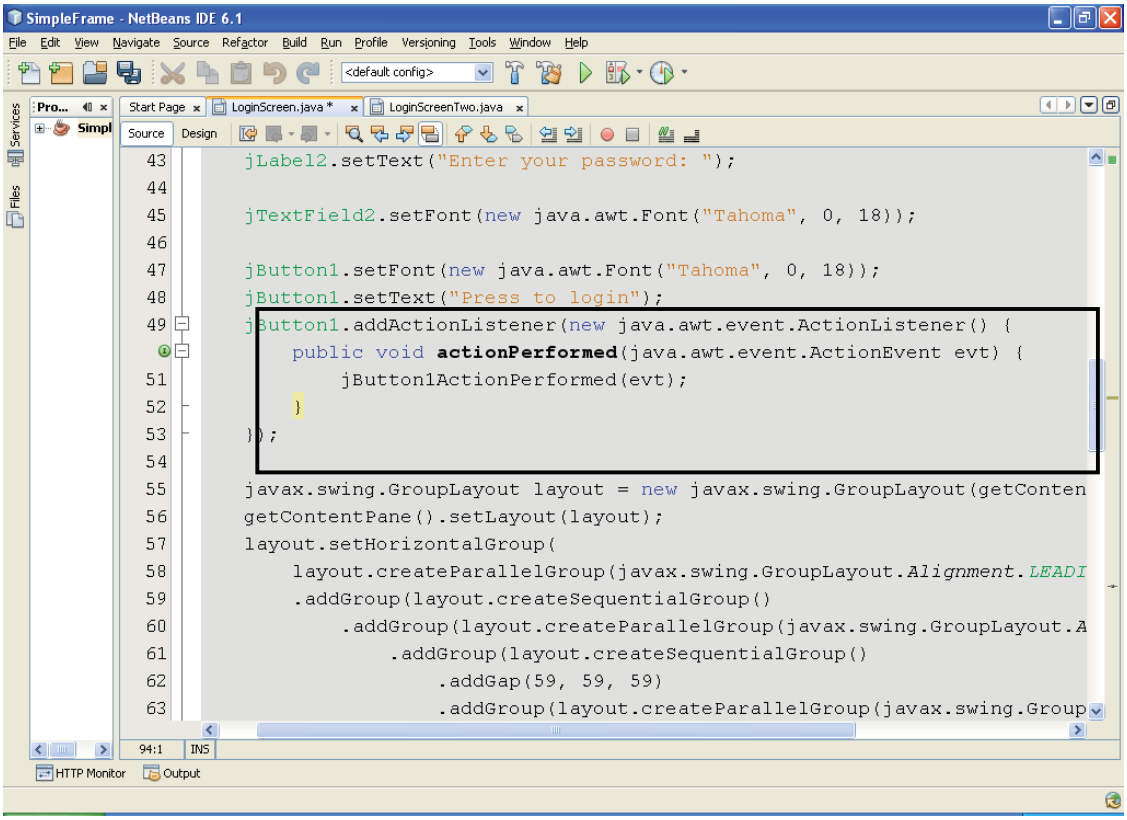
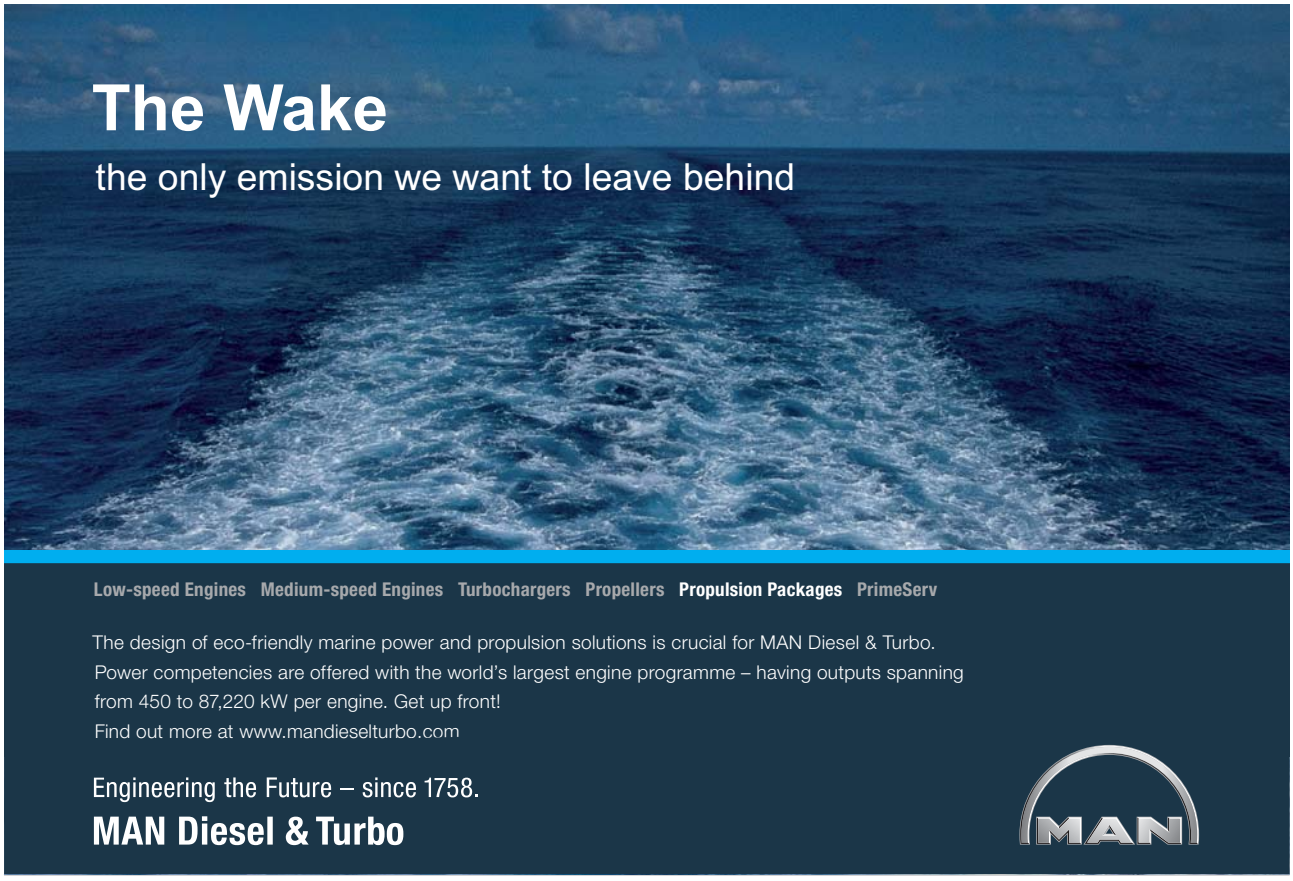


Figure 3.19 An anonymous class calls the event handling method



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

Editing the event handler method as shown in the screen shot seen in Figure 3.20 on the next page produces the same result as before. The result of executing main in the GUI class in the NetBeans™ project is shown in the screen shot shown in Figure 3.21.

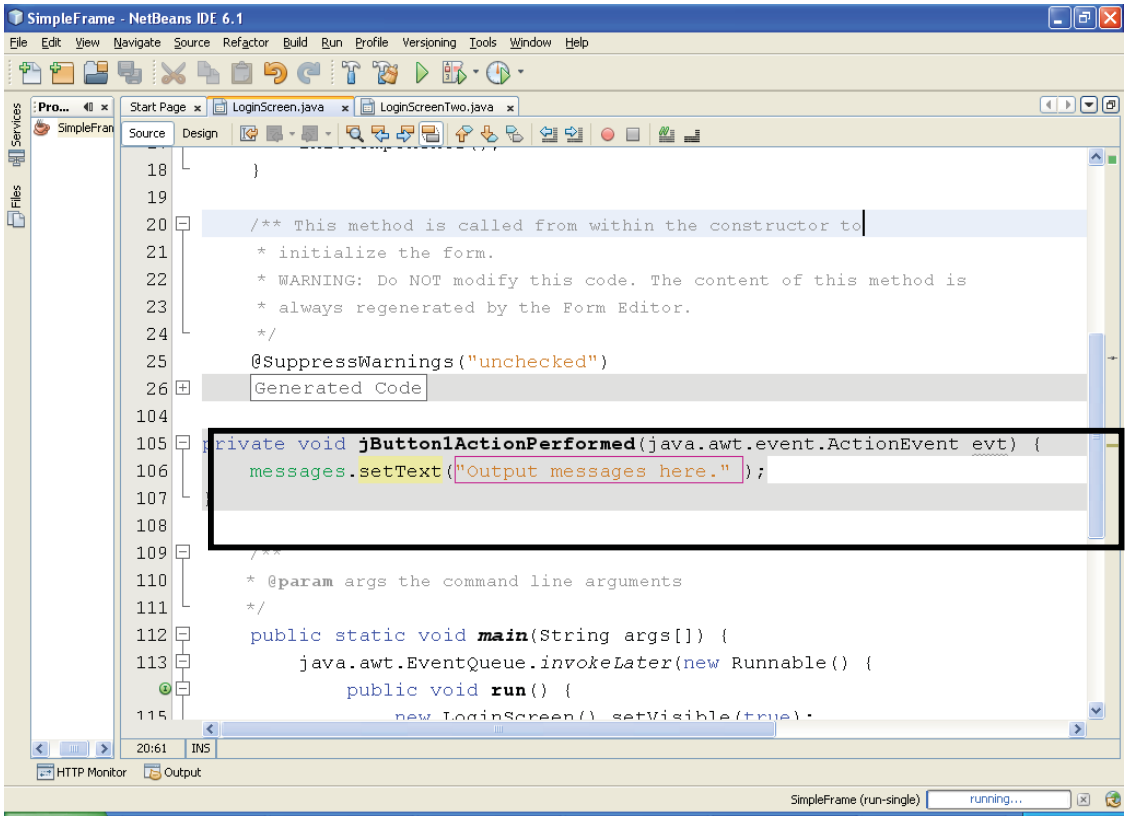


Figure 3.20 Providing the event handling code

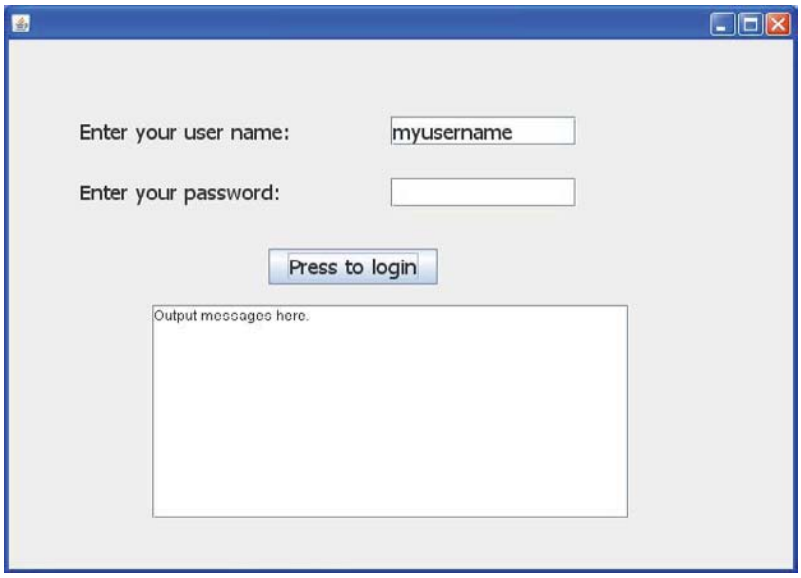


Figure 3.21 A button event for example five

Download free eBooks at bookboon.com

Summary of Listener Classes

The examples above illustrate that there are a number of ways to define the listener class that handles UI events. These include:

1. The use of a separate, external listener class:
   - pass a reference to the GUI class to the constructor of the listener class;
   - this gives the listener class access to GUI components via their 'get' methods;
2. the GUI class is its own listener;
3. use inner class(es);
4. use an anonymous class(es) that invokes an event handler method of the enclosing class.

Methods of the listener classes defined by methods 2, 3 and 4 have direct access to the private variables, including the UI components, of their enclosing class. This is more convenient than Method 1, where 'get' methods have to be provided in the GUI class so that UI components can be accessed by means of the object reference to the GUI object that is passed to the external listener class. Although Method 1 delegates event handling code to a separate class definition – which gives it flexibility and convenience – Method 4 is recommended by the author (of this guide) firstly because event handling code is delegated to dedicated methods that are coded by the developer and, secondly, because many IDEs adopt this approach.

3.6 The GUI for the Themed Application

The complete GUI for the themed application is shown in the series of screen shots that follow on the next two pages. Each part of the GUI is implemented as a **`JPanel`** component that has been added to each of the tabbed panes of a **`JTabbedPane`** container.



Figure 3.22 The welcome tab

Download free eBooks at bookboon.com

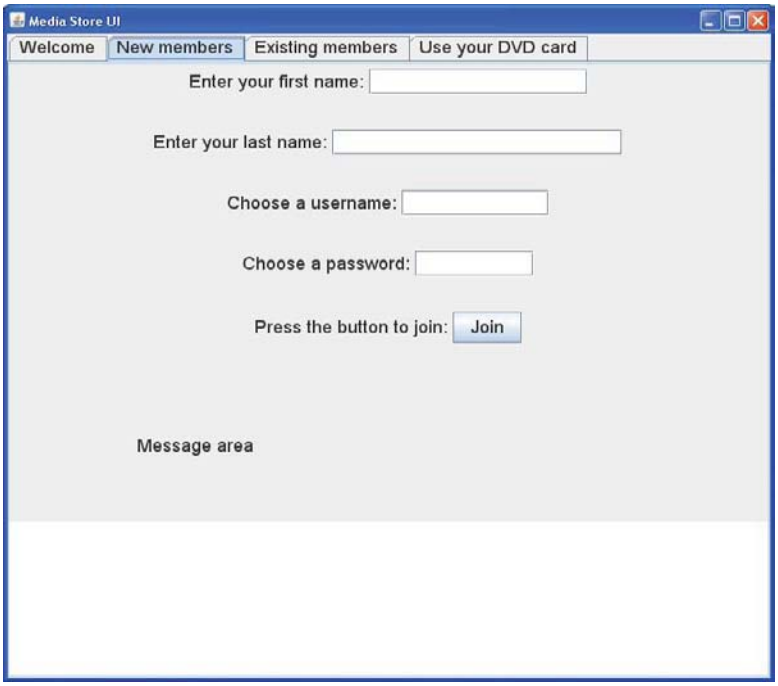


Figure 3.23 The new members tab



Figure 3.23 The existing members tab

Download free eBooks at bookboon.com

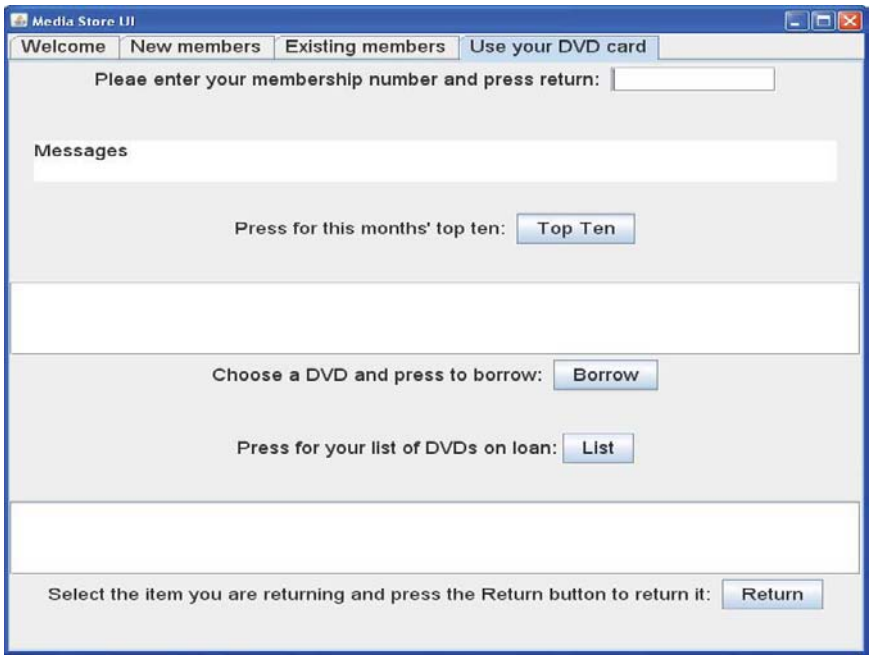


Figure 3.24 The tab for card transactions

The GUI shown in the screen shots above was constructed by hand in the first instance and then it was constructed using the visual design features in NetBeans™. In both cases, the code for the event handler methods is the same. For example, the event handler for the login button scans the array of existing members in order to find the member who is attempting to login. The code for the event handler method is as follows.



Download free eBooks at bookboon.com

```
// Event handler for the 'Login button' for existing members. The purpose of this handler is to
// find the member in the array of existing members.
if ( ( JButton )( ae.getSource( ) ) == loginButton )
{
        // Read in the array of members.
        mediaStore.readMembers( );
        // Capture the existing member's user name and password.
        String userName = userNameFieldExistingMember.getText( );
        // Next, get the contents of the password field.
        char [ ] chars = passwordFieldExistingMember.getPassword( );
        // Declare a local variable to store the password and add the array of characters to a
        // String.
        String password = new String ( chars );
        // Concatenate the user name and password.
        String searchString = userName + password;
        // Search the array of members for the combined user name and password.
        // First, get the array of members.
        Member[ ] existingMembers = mediaStore.getMembers( );
        // 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;
                        messagesAreaExistingMembers.append(
                                "\nThe combined " + name and password is: " +
                                                combinedNameAndPassword );

                        if ( searchString.equals( combinedNameAndPassword ) )
                        {
                                // Found existing member in the array of members.
                                // Output a message.
                                messagesAreaExistingMembers.append(
                                        "\nFound member." );
                                messagesAreaExistingMembers.append(
                                        "\nYour user name" +" is: " +
                                        existingMember.getUserName( ) );
                                messagesAreaExistingMembers.append(
                                         \nYour password" +
```


Download free eBooks at bookboon.com

```
                                    " is: " + existingMember.getPassword( ) );
                            messagesAreaExistingMembers.append( "\nYour" +
                                    " membership number is: " +
                                    existingMember.getMembershipNumber( ) );
                            // Test: find out if the DVD card object has a non-null
                            // reference.
                            messagesAreaExistingMembers.append( "\nYour DVD" +
                                    " card is: " + existingMember.getCard( 0 ) );

                            // Enable the next tab.
                            tabbedPane.setEnabledAt( 3, true );

                            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 )
                {
                        messagesAreaExistingMembers.append( "\nNo such " +
                                " member; please try again." );
                }

        } // end of outer if
```

The purpose of presenting the complete code for the event handler is not to encourage the reader to understand its every statement. Rather, it is intended to show the reader that code for an event handler method can be substantial. In the example shown above, the event handler has work to do when an exiting member attempts to login. If the login is successful, the 'Use Your DVD Card' tab is enabled so that the member can use his or her card to borrow or return DVDs.

3.7 Summary of Event Handling

The examples discussed in Section 3.6 present an opportunity to make a number of general points about the second stage of the construction of a GUI: activating UI components. These are presented in the box below.

Download free eBooks at bookboon.com

- Constructing a GUI that responds to *events* requires an *event handler method* encapsulated in a *listener* object.
- Events are objects that are automatically instantiated when a user interacts with a UI component.
- The UI component sends the event to one or more listener objects which are *registered* with the UI component.
- Every event has a corresponding listener interface which declares which methods must be defined in listener classes that implement them.
- Listener objects, with their handler methods, are *registered* with a UI component that they then observe or listen to.
- A listener contains one or more event handler methods that receive and process events.

The examples in Section 3.6 consider only the **ActionListener** interface implemented by button listeners. As the code shows, this interface declares a single method:

public void actionPerformed(ActionEvent e)

that takes an **ActionEvent** object as its only argument.

The API lists many interfaces and listeners associated with event handling. The tables that follow on the next page lists a few of these merely to indicate their scope to the reader.



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

Interface Summary	
ActionListener	The listener interface for receiving action events.
ItemListener	The listener interface for receiving item events.
KeyListener	The listener interface for receiving keyboard events (keystrokes).
MouseListener	The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component.
MouseMotionListener	The listener interface for receiving mouse motion events on a component.
MouseWheelListener	The listener interface for receiving mouse wheel events on a component.

Table 3.1 Interfaces

Class Summary	
KeyEvent	An event which indicates that a keystroke occurred in a component.
MouseEvent	An event which indicates that a mouse action occurred in a component.
TextEvent	A semantic event which indicates that an object's text changed.

Table 3.2 Listener classes

For example, an `ItemListener` can be registered with a `JCheckBox` component as follows:

myCheckBox.addItemLIstener(new ItemListener( ))

where the `ItemLIstener` interface declares a single method

public void itemStateChanged(ItemEvent e)

that is invoked when an item has been selected by the user of the check box.

In short, the API can be used to find out how to register a UI component with an appropriate listener that implements the relevant interface, depending on the nature of the event or events associated with users when they interact with the component.

Section 3.3.2 includes a list of suggested steps that should be taken when constructing a GUI. Step 9 (in the list) reiterates the recommendation stated in the on-line Java tutorial that a Swing GUI should execute in what is known as a *thread of execution*. The next, and final, chapter of this guide examines how a Java programme can achieve concurrency of execution of tasks by means of threads.

Download free eBooks at bookboon.com

4. Concurrency with Threads

The use of the **Thread** class is often regarded as a relatively advanced topic for a beginners' guide to Java. However the reader may recall, from Chapter Three, that a Swing GUI executes in its own *thread of execution*. This is an outcome of the fact that the Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Despite the relative advanced nature of threads (of execution), it is clear from a study of Chapter Three that the reader should be aware of the fundamental concepts associated with threads in Java applications. Chapter Four briefly examines how more than one task can be carried out concurrently in Java programmes by means of threads.

4.1 An Introduction to Threads

Section 3.3.2 gives the code for the **main** method that renders a Swing GUI in its own thread. The **main** method that calls the constructor for a GUI is usually a member of a lightweight class with the following generalised class definition.

```
public class RenderTheGui {

    public static void main( String[ ] args ) {
        // The next statement is a call to the invokeLater method of the
        // SwingUtilities class of the javax.swing package. The invokeLater
        // method takes a Runnable object as an argument.  In essence, the purpose of
        // the Runnable object is to initialise the GUI and schedule it for execution in
        // a thread.
        javax.swing.SwingUtilities.invokeLater( new Runnable( ) {
            // start of anonymous class definition of the Runnable object;
            // call the thread's run method
            public void run( ) {
                // call the GUI's constructor
                MySwingClass gui = new MySwingClass( );
            } // end of run method
            } // end of Runnable class definition
        ); // end of call to invokeLater method
    } // end of main

} // end class definition
```

An anonymous object that implements the **Runnable** interface is created and passed as an argument to the **invokeLater** method. The effect of calling the **invokeLater** method is to schedule the creation of the GUI to execute in its own thread and causes the thread's **run** method to be called, the purpose of which is to call the constructor for the GUI.

In effect, the purpose of the class called **RenderTheGui** is to schedule the GUI to execute in its own thread. The principal aim of this action is so that the GUI does not 'freeze' when it is used by a user, irrespective of what other tasks that the application is carrying out.

Download free eBooks at bookboon.com

The example shows that the actual work of a thread is done in its **run** method; i.e. that task or tasks that the thread is required to carry out on behalf of the application is coded in the body of the thread's **run** method.

4.2 Creating Threads

One of the tools that are integrated with BlueJ, the IDE used to develop most of the examples presented in this guide, shows that several threads are running when the GUI of the themed application is rendered. Most of these threads are system threads or threads that are associated with running BlueJ as an application. Two of these threads are associated with the themed application, namely the thread that runs **main** and the thread that displays the GUI. The only thread that is created explicitly by the developer is the one that calls the constructor of the GUI, as shown in the code in Section 4.1: the other threads are created automatically by the JVM in order to manage the IDE and various system resources.

The code in Section 4.1 illustrates one of two methods of creating a thread: i.e. create an object that implements the **Runnable** interface.

There are two methods of explicitly creating a thread in a Java programme:

1. Create an object that inherits from the java.lang.Thread class.
2. Create an object that implements the java.lang.Runnable interface.

4.2.1 An Object that Inherits from the Thread Class

A template for the definition of a class that inherits from the **Thread** class follows on the next page.



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

```
public class MyThreadOne extends Thread {

    // declare instance variables

    // constructor for the thread
    public MyThread( ) {
        // code goes here
    }

    // override the run method of the Thread class
    public void run( ) {
        // code executed by thread
    }

    // other members of the class

} // end of class definition
```

The thread object is instantiated by calling the **start** method of the **Thread** class; this invocation places the thread in a *runnable* state, which means that it becomes available for scheduling by the JVM. The **start** method automatically calls the thread's **run** method.

The thread is typically instantiated in a separate, lightweight class that includes a **main** method. A template for this class follows.

```
public class TestMyThreadOne {

    public static void main( String args[ ] ) {
        MyThreadOne mto = new MyThreadOne( );
        mto.start( );
    }

} // end of class definition
```

The body of the **main** method instantiates an instance of the thread by calling its no-arguments constructor; the **start** method of the thread is called on this instance.

4.2.2 An Object that Implements the Runnable Interface

A template for the class definition of a class that implements the **Runnable** interface is as follows.

```
public class MyThreadTwo implements Runnable {

    // declare instance variables
```


Download free eBooks at bookboon.com

```
        // constructor for objects of this class
        public MyThreadTwo( ) {
                // code goes here
        }

        // implement the run method of the Runnable interface
        public void run( ) {
                // code executed by thread
        }

} // end of class definition
```

The next template instantiates objects of the class that implements the **Runnable** interface.

```
public class TestMyThreadTwo {

        public static void main( String args[ ] ) {
                MyThreadTwo mtt = new MyThreadTwo( );
                Thread thread = new Thread( mtt );
                thread.start( );
        }

} // end of class definition
```

In this case, the body of the **main** method instantiates an instance of the class that implements the **Runnable** interface and it passes this object reference to the constructor of the **Thread** class that takes a **Runnable** object as its only parameter. The **start** method of the thread is called, as before.

The outcome of using either of the two methods that can be used to create threads is that the developer provides the body of the thread's **run** method; it is this method that does the work that the thread is required to do in a Java application.

4.3 Using Threads in Java Applications

There are a number of tasks that could be designed to execute in a dedicated thread; these include:

- I/O tasks that require substantial resources or are large in scale;
- large-scale printing tasks where the print driver executes in its own thread;
- synchronised multiple read/write tasks;
- server applications that provide an application service to multiple clients.

The fourth item in the list above identifies applications that require shared access to resources from multiple client applications and implies a high degree of synchronisation in order to maintain the integrity and security of data. The next sub-section explains, with an example, how synchronisation can be achieved.

Download free eBooks at bookboon.com

4.3.1 The synchronized Keyword

It seems reasonable to assert that data that is required to be accessed by multiple client applications must be synchronised to protect the state of the data so that it is consistent from the point of view of client applications that need to use it. Synchronisation logic is required for any server application that provides simultaneous services to multiple clients that require read/write access to shared data. This can be achieved in Java applications by controlling the thread that accesses an object's data values by identifying the critical sections of code that require exclusive access to shared data and 'flagging' such code by using the keyword **synchronized**.

Synchronisation of critical sections of code relies on an entity known as the *intrinsic lock* of an object. A thread 'owns' an object's lock between the time it acquires it and releases it. The Java language provides two synchronising idioms: *synchronised methods* and *synchronised statements*.

When a thread invokes a synchronised method, it acquires the lock for that method's object and releases it when the method returns. Synchronising a method allows exclusive access to code that accesses shared data and ensures that it is not possible for two invocations of the method to interleave and interfere with one another. When a thread invokes a synchronised method of an object, all other threads that invoke synchronised methods of that object are blocked until the first thread has finished with the object and releases its lock. Thus, sharing an object amongst threads is made safe by declaring methods to be synchronised using the keyword **synchronized** as follows:



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

```
public synchronized void aMethod( ){
        // body of method
}
```

Synchronising statements, on the other hand, provides a finer-grained approach than with synchronising methods. When synchronising a block of statements, the block must specify the object that provides the lock, as shown next.

```
public void aMethod( ) {
        synchronized( anObjectReference ) {
        // scope of synchronised block
}
```

When the thread reaches the synchronised block, it examines the object passed as an argument and obtains the object's lock before continuing with the execution of the statements in the block. The lock is released when the thread passes the end of the block.

The example that follows on the next few pages illustrates how methods and statements are synchronised in the thread that runs a banking application. The author (of this guide) uses the example to teach some of the principles of distributed, client/server applications where the client and server run in separate JVMs on a computer network. The outcome of distributing the client and server components of the application means that the synchronised code in the server ensures that only one client at a time can access a customer's account. Some of the code of the bank's server class is omitted in order to allow the reader to identify and study the purpose of the code that is synchronised. The definitions of the **Account** and **BankingException** classes do not need to be shown here.

An object of the class that follows is instantiated in the **run** method of a thread so that its methods can be called from multiple clients.

```
/**
* The server class is based on David Flanagan's code; it implements methods for bank
* account transactions.  Account data is written out to a data a file.
* Modified by David M. Etheridge.
* @version  1.0, dated 17 January 2008.
* Source: David Flanagan, Java Examples in a Nutshell, 2nd Edition, 2000.
*/
import java.util.*;
import java.io.*;

public class BankServer {
        /** This data structure stores accounts.  */
        Map accounts = new HashMap( );
        /** The constructor reads the accounts from a file.  */
```


Download free eBooks at bookboon.com

```
public BankServer( ) {
        try
        {
                readData( );
        }
        catch( Exception e )
        {
                System.out.println( "Error: " + e.getMessage( ) );
        }
} // end of constructor

/** This method opens a bank account with a name and password.  It is synchronised
* to make it thread safe, since it manipulates the data structure of accounts.
* @param name  The name of the account.
* @param password  The account's password.
*/
public synchronized void openAccount( String name, String password )
        throws BankingException, IOException, ClassNotFoundException {
        // find out if there is already an account with this name
        if ( accounts.get( name ) != null )
        {
                throw new BankingException( "Account already exists.\n" );
        }
        // otherwise, if the account doesn't exist: create it
        else
        {
                Account acct = new Account( name, password );
                // put the account in the data structure
                accounts.put( name, acct );
        }
        // write the accounts to a file
        try
        {
                FileOutputStream fos = new FileOutputStream(
                        "C:\\temp\\accounts.dat");
                ObjectOutputStream oos = new ObjectOutputStream( fos );
                oos.writeObject( accounts );
                oos.flush( );
                oos.close( );
                fos.close( );
        }
        catch ( IOException e ) {
                System.out.println( "Error: " + e.getMessage( ) );
        }
} // end of openAccount method
```


Download free eBooks at bookboon.com

```
/** This method cannot be invoked from the client GUI.  Given a name and password,
* it checks to see if an account with that name and password exists.  If so, it returns the
* Account object.  Otherwise, it throws an exception.
* @return  A reference to an existing account.
* @param name  The name of the account.
* @param password  The account's password.
*/
Account verify( String name, String password ) throws BankingException {
        synchronized( accounts )
        { // start of synchronised block; note the cast
                Account acct = ( Account )accounts.get( name );
                if ( acct == null )
                {
                        throw new BankingException( "No such account.\n" );
                }       else        {
                        if ( !password.equals( acct.password ) )
                        {
                                throw new BankingException(
                                "Invalid password.\n" );
                        } // end if
                } // end else
                return acct;
        } // end of synchronized block
} // end of verify method
```



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

```
/** This method closes an account.  It is synchronisd to make it thread safe, since it
* manipulates the data structure of accounts.
* @param name  The name of the account.
* @param password  The account's password.
*/
public synchronized void closeAccount( String name, String password )
        throws BankingException {

        Account acct = verify( name, password );
        accounts.remove( name );
        acct.transactions.add( "Account closed at " + new Date( ) );
        // before changing the balance an account, we first have to obtain a lock on
        // that account
        synchronized ( acct )
        { // start of synchronised block
                double balance = acct.balance;
                // assume a closing balance of zero
                acct.balance = 0;
                // write the accounts to a file as in the openAccount method
        } // end of synchronised block
} // end of closeAccount method

/** This method deposits an amount in the named account.
* @param name  The name of the account.
* @param password  The account's password.
* @param depositAmount  The amount to be deposited.
*/
public void deposit( String name, String password, double depositAmount )
        throws BankingException {

        Account acct = verify( name, password );
        synchronized( acct )
        { // start of synchronised block
                acct.balance += depositAmount;
                acct.transactions.add(
                        "Deposited " + depositAmount + " on " + new Date( ) );
                // write the accounts to a file
        } // end of synchronised block
} // end of deposit method

/** This method withdraws an amount from the named account.
* @param name  The name of the account.
* @param password  The account's password.
* @param depositAmount  The amount to be withdrawn.
*/
```


Download free eBooks at bookboon.com

```
public double withdraw( String name, String password, double
        withdrawAmount ) throws BankingException {

        Account acct = verify( name, password );
        synchronized( acct )
        { // start of synchronized block
                if ( acct.balance < withdrawAmount )
                {
                        throw new BankingException(
                                "Insufficient funds for this withdrawal.\n" );
                }
                else
                {
                        acct.balance - = withdrawAmount;
                        acct.transactions.add(
                                "Withdrew " + withdrawAmount + " on "
                                + new Date( ) );
                }
                // write the accounts to a file

                return withdrawAmount;
        } // end of synchronised block
} // end of withdraw method

/** This method returns the current balance of the named account.
* @return  The account's balance.
* @param name  The name of the account.
* @param password  The account's password.
*/
public double getBalance( String name, String password )
                throws BankingException {

        Account acct = verify( name, password );
        synchronized( acct )
        { // start of synchronised block
                return acct.balance;
        } // end of synchronised block
} // end of getBalance method

/** This method returns a List of strings containing the transaction history of the
 * named account.
* @return  The transaction history of the account.
* @param name  The name of the account.
* @param password  The account's password.
*/
```


Download free eBooks at bookboon.com

```
public List getTransactionHistory( String name, String password )
        throws BankingException, ClassNotFoundException {

        Account acct = verify( name, password );
        synchronized( acct )
        { // start of synchronised block
                return acct.transactions;
        } // end of synchronised block
} // end of getTransactionHistory method

/** This method reads the accounts from a file; it is called from the constructor.  */
public void readData( ) throws IOException, ClassNotFoundException {
        try
        {
                FileInputStream fis = new FileInputStream(
                        "C:\\temp\\accounts.dat");
                ObjectInputStream ois =
                        new ObjectInputStream( fis );
                accounts = ( Map )( ois.readObject( ) );
                ois.close( );
                fis.close( );
                }
                catch (IOException e) {
                        System.out.println( "Error: " + e.getMessage( ) );
                }
} // end of readData method

} // end of BankServer class definition
```

The reader is not expected to understand fully how the bank application works. Rather, the aim of presenting the substantive code for the **`BankServer`** class is so that the reader can gain an understanding how synchronisation is used to synchronise methods and blocks of code in order to meet the requirements of the application in a way that ensures that a named account can be accessed by only one client application at a time.

4.4 Summary of Threads

Chapter Four explains how the **`Thread`** class and the **`Runnable`** interface are used in a Java programme to create threads of execution. An example is used to illustrate how synchronised access to code is achieved in situations that require exclusive access to shared data resources.

Download free eBooks at bookboon.com

The chapter omits any discussion of *thread scheduling*. It is sufficient to say, for the purposes of this guide, that Java threads are *pre-emptive*. This means that the pre-emptive scheduler knows that a number of threads are runnable because their **`run`** method has been invoked implicitly by the JVM or explicitly by the developer's code. However, only one thread is actually running at a time. A running thread continues to run until it ceases to be runnable or another thread of higher priority becomes runnable. In the latter case, the lower priority thread is pre-empted by the higher priority thread. A thread might cease to be runnable (i.e. it become blocked) for a variety of reasons, such as it might have to wait to access a resource. This gives other threads a chance to execute.

Some of the scheduling conditions can be influenced by developer's code; however, the fields and methods of the **`Thread`** class are not considered in this chapter. The reader is referred to the documentation of the **`Thread`** class in the Java API for an initial insight concerning how the developer can influence thread scheduling.

Download free eBooks at bookboon.com