Chapter 7: Java Network Applications

Introduction

Historically, network programming has been thought of as complex and difficult. It was necessary to understand the various network layers involved, along with the data packaging, network connection, data shipping and the handshaking involved. However, the concept of networking is not that difficult to understand - in fact, it is quite similar to the concept of file I/O, except that the file is some black box on another computer/device. The Java programming language has significant network supports and it is structured to hide the lower-level communication from the programmer.

Unless it is behind a firewall/router, each computer connected to the Internet has a unique address called an IP address. Each IP address is a 32-bit number, of the form xx.xx.xx.xx, where xx is an 8-bit number that represents a number between 0 and 255. An IP address can also be represented by a domain name form, such as "www.eeng.dcu.ie". Computers also have a loop-back IP address (127.0.0.1), so that provided you have a TCP/IP stack installed on your computer you can develop network aware applications. The new IP address format is also available under Windows 2000+ (IP-6) that provides addresses of the form xxxx.xxxx.xxxx.xxxx i.e., 128-bit addressing. IPv4 supports 4.2 billion addresses, which is inadequate to even provide one address to each living person (although routed internal network sharing and reuse is possible). IPv6 supports 3.4x10^38 addresses, enough to provide every atom of every person alive on the earth 7 unique IP addresses (assuming 6.5 billion humans, each having 7x10^{27} atoms [12]).

A computer connected to the internet will often provide several services. For example, it could act as both a web server and an e-mail server. To allow multiple services, a **port number** is used that provides a software abstraction, rather than a hardware address. Some common ports for services are 80 (http), 20/21 (ftp data/control), 7 (echo), 22(SSH inc. sFTP), 23 (telnet), 25 (SMTP), etc.

There are several mechanisms that provide network services. There is **UDP** (User-Datagram Protocol) and **TCP** (Transmission-Control Protocol). We will use TCP for the majority of our applications, as we are guaranteed that the messages will arrive at their destination. UDP is less reliable and can often lose packets in transit, but is still useful for applications like broadcasting audio/video. Most protocols (such as HTTP, SMTP, SSH and FTP) are based on TCP.

Java Internet Aware Applications

The InetAddress class

We start a Java Internet application by importing java.net.*, that contains the core networking routines. It contains the InetAddress class, that allows us a simple method of converting between numeric and DNS form addresses.

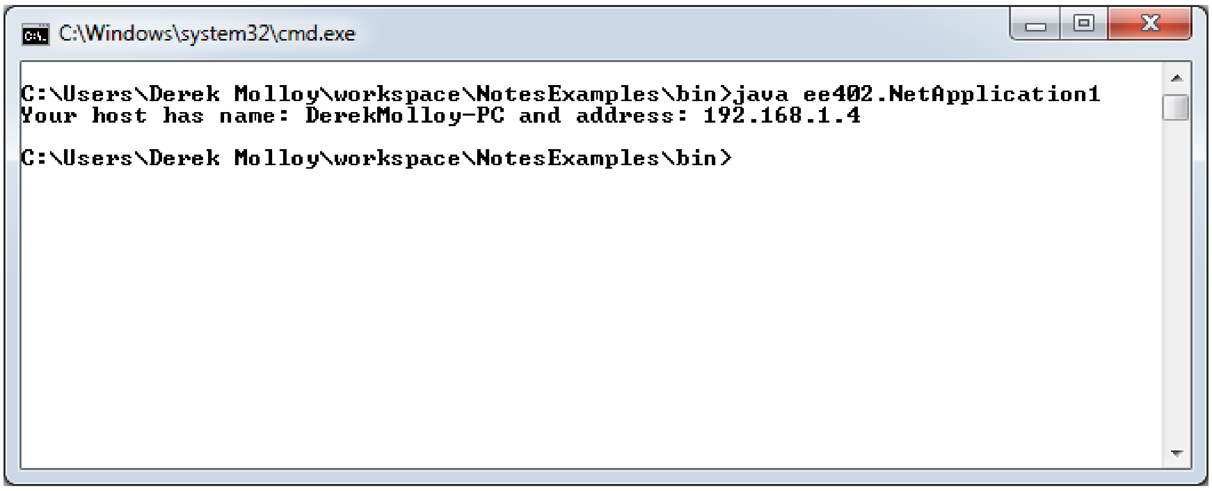
Here is an example of using the InetAddress class. The source code is below:

NetApplication1.java

```java
/* Java Network Application Example - Written by Derek Molloy for the EEN1035 Module
 *
 */

package ee402;

import java.net.*;

public class NetApplication1 {

   public static void main(String[] args) {
      try{
         InetAddress localAddress = InetAddress.getLocalHost();
         String hostname = localAddress.getHostName();
         String address = localAddress.getHostAddress();
             System.out.println("Your host has name:" + hostname + "and address:" +
address);
      }
      catch (UnknownHostException e){
         System.out.println("Cannot detect localhost : " + e.toString());
      }
   }
}
```

This simple application shows us how Java treats networked applications. Again the class structure comes into play where methods associated with the Internet addresses are a part of the InetAddress class. This application is executed by typing **java ee402.NetApplication1**. The output looks like Figure 7.1, "InetAddress Class Example".

Figure 7.1 The InetAddress Class Example



C:\Windows\system32\cmd.exe

```
C:\Users\Derek Molloy\workspace\NotesExamples\bin>java ee402.NetApplication1
Your host has name: DerekMolloy-PC and address: 192.168.1.4

C:\Users\Derek Molloy\workspace\NotesExamples\bin>
```

In this case the local IP address is listed as `192.168.1.4` - and the Microsoft Networks name of my PC is DerekMolloy-PC. This is a fixed IP address behind my home firewall. The local hostname is extracted using the getHostName() method and returns the String object "DerekMolloy-PC".

A Simple "Who Is?" Application

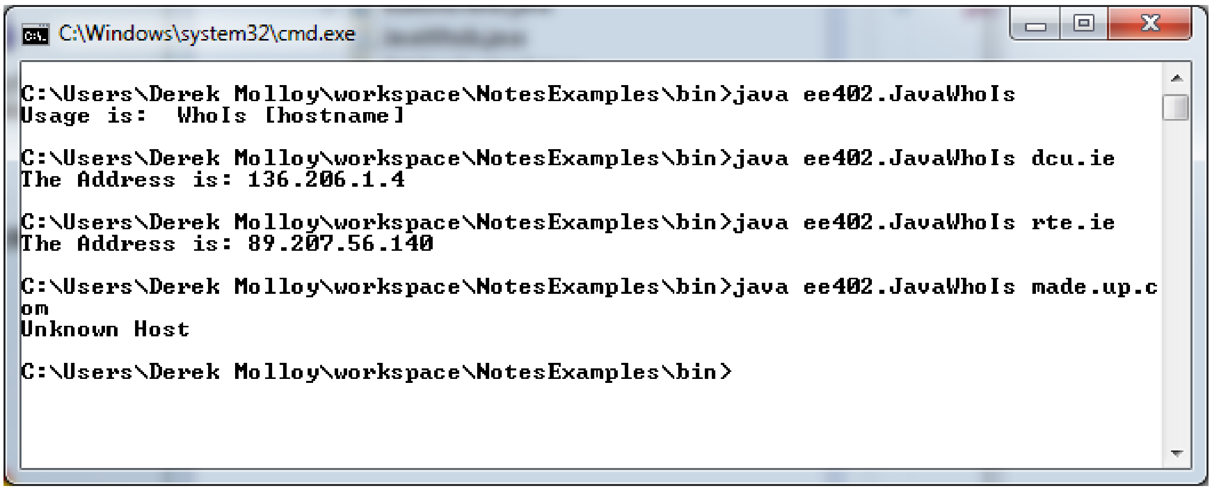
We can modify the code that we just wrote to create a simple "Who Is?" application. If can write this simply by the following code (as below):

JavaWhoIs.java

```java
/* Java WhoIs Example - Written by Derek Molloy for the EE402 Module
* See: ee402.eeng.dcu.ie
*/

package ee402;

import java.net.*;

public class JavaWhoIs {

   public static void main(String[] args) {
      if (args.length!=1){
         System.out.println("Usage is:  WhoIs [hostname]");
         System.exit(0);
      }
      try{
         InetAddress i = InetAddress.getByName(args[0]);
         System.out.println("The Address is: " + i.getHostAddress().toString());
      }
      catch(UnknownHostException e){
         System.out.println("Unknown Host");
      }
   }
}
```

When this application is run it expects the name of a machine, either specified by the full Internet name, or the name of a machine on the same network segment. If you forget to provide the parameters, it prompts you for the correct parameters (See Figure 7.2, “Java WhoIs Example”)

Figure 7.2 The Java WhoIs Example



```
C:\Users\Derek Molloy\workspace\NotesExamples\bin>java ee402.JavaWhoIs
Usage is:  WhoIs [hostname]

C:\Users\Derek Molloy\workspace\NotesExamples\bin>java ee402.JavaWhoIs dcu.ie
The Address is: 136.206.1.4

C:\Users\Derek Molloy\workspace\NotesExamples\bin>java ee402.JavaWhoIs rte.ie
The Address is: 89.207.56.140

C:\Users\Derek Molloy\workspace\NotesExamples\bin>java ee402.JavaWhoIs made.up.c
om
Unknown Host

C:\Users\Derek Molloy\workspace\NotesExamples\bin>
```

In Figure 7.2, “Java WhoIs Example” the first time I ran **java ee402.JavaWhoIs** I did not pass the hostname parameter, so I was prompted to do so. The second time I ran **java ee402.JavaWhoIs dcu.ie** and this worked well, returning the IP address `136.206.1.4`.

In the third case we called **java ee402.JavaWhoIs rte.ie** this returns 89.207.56.140. This could change each time you run the command as there would be several servers for www.rte.ie. The fourth time I made up a server and executed **java ee402.JavaWhoIs made.up.com** and since there is no such host (until someone registers it!) an UnknownHostException occurs and our code prints out the message "Unknown Host".

Remember if your computer is not connected to the Internet you can use the loopback address of 127.0.0.1 which is referred to as the hostname localhost, provided that you have installed a TCP stack to your computer.

Java TCP Client Applications

Introduction

Java removes much of the complexity of writing Network aware applications, by providing a Socket class (java.net.Socket). The socket is the software abstraction, used to represent the terminals of a connection between two machines. The basic concept is that you have two sockets on two machines and you create a connection between these sockets, which you use to transfer data. We don't care how the data gets from one socket to another, even though it might have to traverse many network hardware types (hence the abstraction).

We are going to write a finger client to demonstrate how TCP works in Java.

The Finger Client

Finger is a standard service that allows a remote machine to query a server and ask it for information on a particular user, or on the users that are logged on. Most UNIX systems support the finger service, however on many server systems it is disabled, as it provides a little bit too much information for potential 'hackers'.

The host server is usually located at port 79, where a request is usually made up of a username followed by the '\n' character. The server sends any information back to the client and then terminates the connection.

Here is the source for **Finger.java** that connects to any finger server and receives the output:

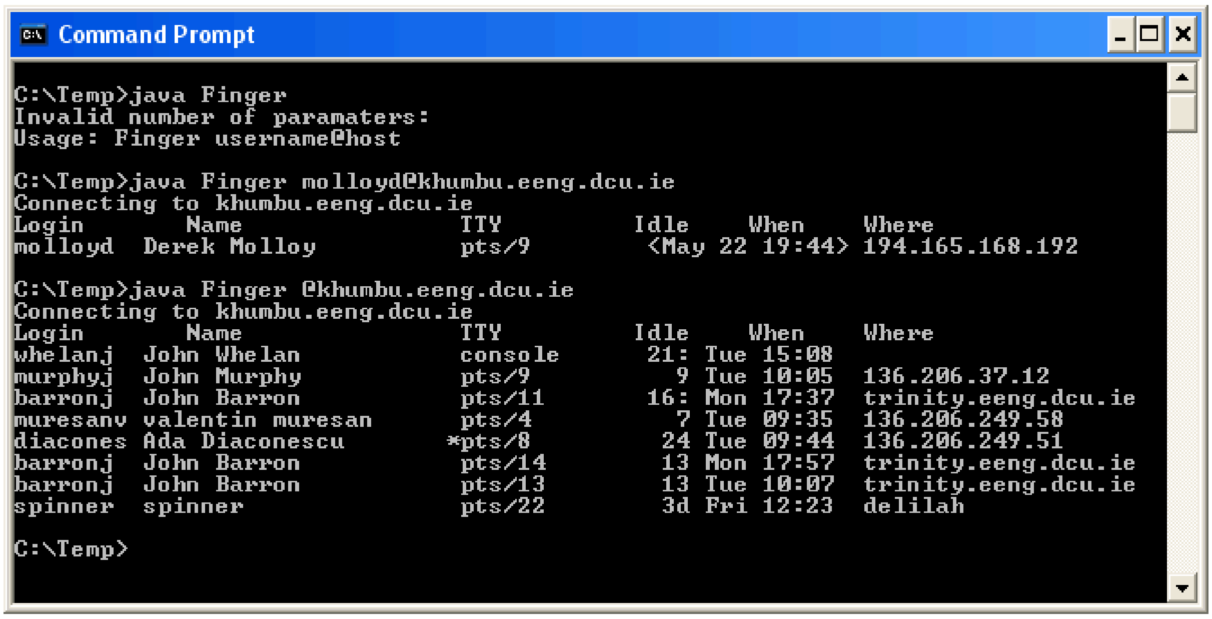
Finger.java

```java
/** TCP finger client - by Derek Molloy ee402.eeng.dcu.ie
Usage: Finger username@host.com
*/

package ee402;

import java.io.*;
import java.net.*;

public class Finger
{

    public static void main (String args[]) {
      // Check command line arguments
      if (args.length != 1) {
           System.err.println ("Invalid number of parameters:");
           System.err.println ("Usage: Finger username@host");
           System.exit(1);
      }
      // Check for existence of @ in argument
      else if (args[0].indexOf("@") == -1) {
           System.err.println ("Invalid parameter : syntax user@host");
           System.exit(1);
      }
      // Split command line argument at the @ character
      String username = args[0].substring(0, args[0].indexOf("@") );
      String hostname = args[0].substring(args[0].indexOf("@") +1, args[0].length());

      try {
          System.out.println ("Connecting to " + hostname);

          // Create a connection to server
          Socket s = new Socket(hostname, 79);

          // Create input and output streams to socket
          PrintStream out = new PrintStream( s.getOutputStream()) ;

          BufferedReader in = new BufferedReader(
                                new InputStreamReader(s.getInputStream()));
          // Write username to socket output
          out.println(username);

          // Read response from socket
          String line = in.readLine();

          while (line != null) {
               System.out.println ( line );
               // Read next line
               line = in.readLine();
          }
          // Terminate connection
```

```
            s.close();
         }
         catch (SocketException e) {
            System.err.println ("Socket error : " + e);
         }
         catch (UnknownHostException e) {
            System.err.println ("Invalid host!");
         }
         catch (IOException e) {
            System.err.println ("I/O error : " + e);
         }
     }
 }
```

When the Finger application is run it expects the name of a user@host, either specified by the full Internet name, or the name of a machine on the same network segment. If you forget to provide the parameters, it prompts you for the correct parameters (See Figure 7.3, "Finger Client Example")

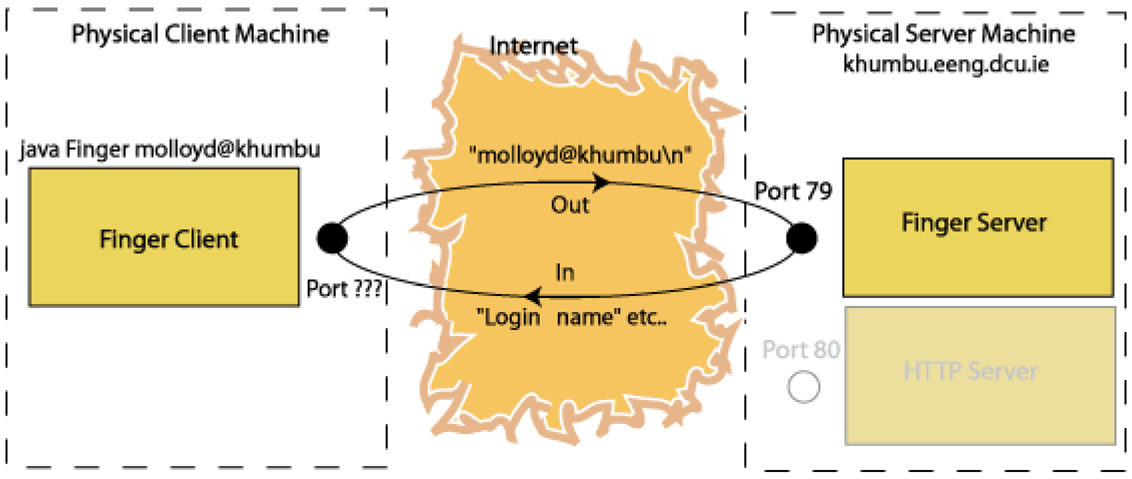
Figure 7.3. Finger Client Example



```
C:\Temp>java Finger
Invalid number of paramaters:
Usage: Finger username@host

C:\Temp>java Finger molloyd@khumbu.eeng.dcu.ie
Connecting to khumbu.eeng.dcu.ie
Login      Name                 TTY         Idle    When     Where
molloyd  Derek Molloy          pts/9        <May 22 19:44> 194.165.168.192

C:\Temp>java Finger @khumbu.eeng.dcu.ie
Connecting to khumbu.eeng.dcu.ie
Login      Name                 TTY         Idle    When     Where
whelanj  John Whelan           console       21: Tue 15:08
murphyj  John Murphy           pts/9          9 Tue 10:05  136.206.37.12
barronj  John Barron           pts/11       16: Mon 17:37  trinity.eeng.dcu.ie
muresanv valentin muresan      pts/4          7 Tue 09:35  136.206.249.58
diacones Ada Diaconescu       *pts/8         24 Tue 09:44  136.206.249.51
barronj  John Barron           pts/14        13 Mon 17:57  trinity.eeng.dcu.ie
barronj  John Barron           pts/13        13 Tue 10:07  trinity.eeng.dcu.ie
spinner  spinner               pts/22        3d Fri 12:23  delilah

C:\Temp>
```

I have run the finger application three times in Figure 7.3, "Finger Client Example". The first time I omitted the parameter by typing **java Finger** and I was then prompted for the correct parameter. In the second case I typed **java Finger molloyd@khumbu.eeng.dcu.ie** and this searches for the username molloyd at the hostname khumbu.eeng.dcu.ie (a Unix server in the school with a finger server installed) and this returns details about this user. In the third case I omitted the username and just typed **java Finger @khumbu.eeng.dcu.ie**. The finger server is designed in this case to return a list of users that are currently logged on, and the locations where they are logged on from. **Note**: Khumbu may not accept connections from clients outside my subnet for security reasons. So the process that occurs to get the output as in Figure 7.3, "Finger Client Example" is summarized in Figure 7.4, "Finger Client/Server Process"

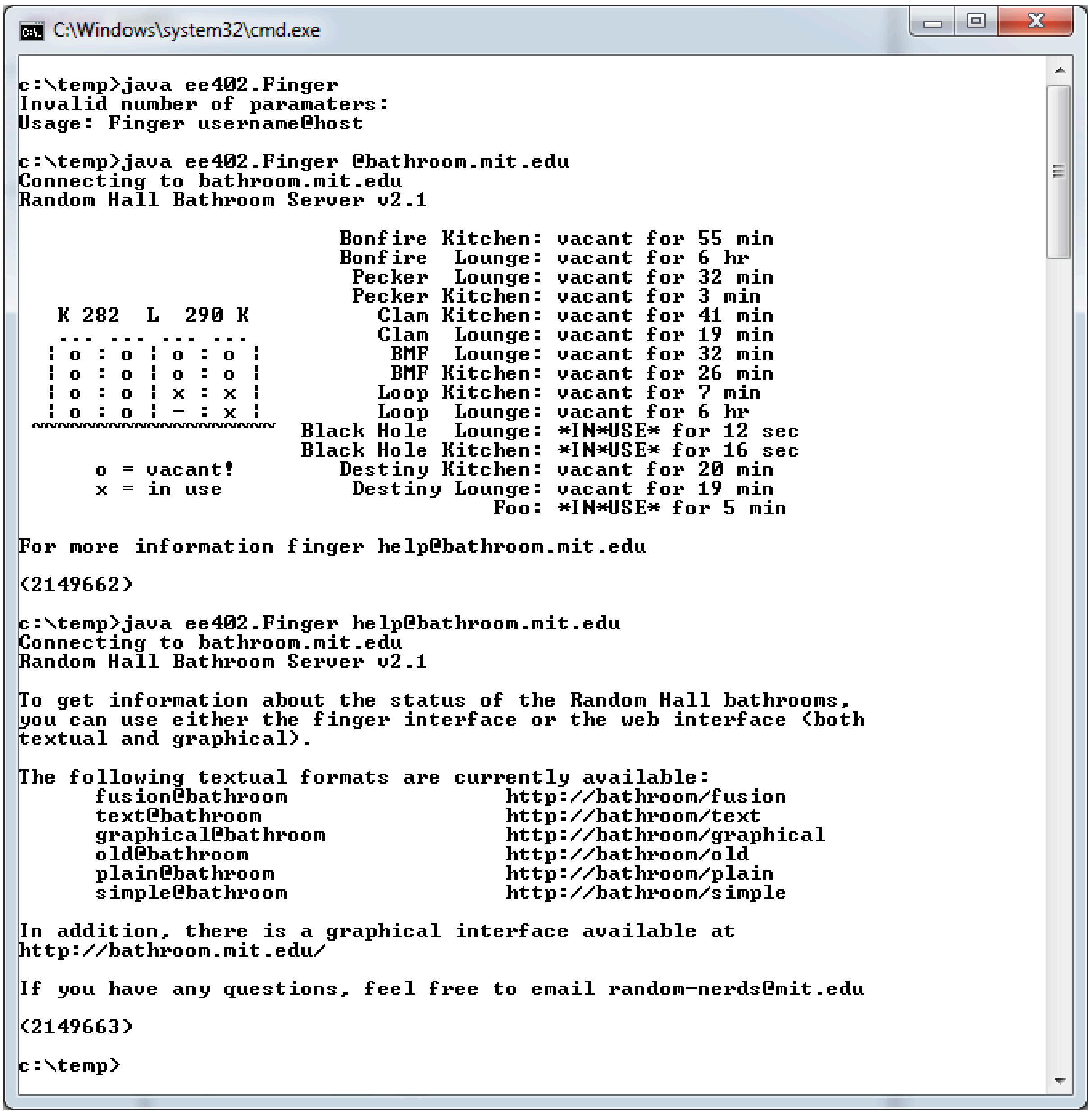
Figure 10.4. Finger Client/Server Process



The Finger application consists of three main steps:

- Import the Network API and I/O packages. Add the main() method that is responsible for all the coding. Receive the parameters from the command line args[] and extract the username and hostname from a String object of the form "user@some.host.com".
- Connect to the server that runs on port 79 (The standard finger protocol port). Try to catch any exceptions that may occur, such as SocketException if the socket does not establish correctly, UnknownHostException if the host is invalid and otherwise if an error occurs during reading/writing a general IOException may occur.
- After establishing the connection. Write the output to the server and then read in the input stream. For the output stream in this case I use a PrintStream that provides convenient functionality over and above the standard OutputStream such as the println() method that we use. To create the PrintStream object we pass the standard OutputStream object to its constructor. For reading the reply from the server we use the BufferedReader class. Before the current JDK we would have used the InputStreamReader but there was an error in the way that the readLine() was used, and so we must use the BufferedReader class. This class provides the extra functionality on top of the standard InputStream such as the readLine() that we need for this application.

Figure 7.4 Executing the Finger Client Application (Executed in 2013)

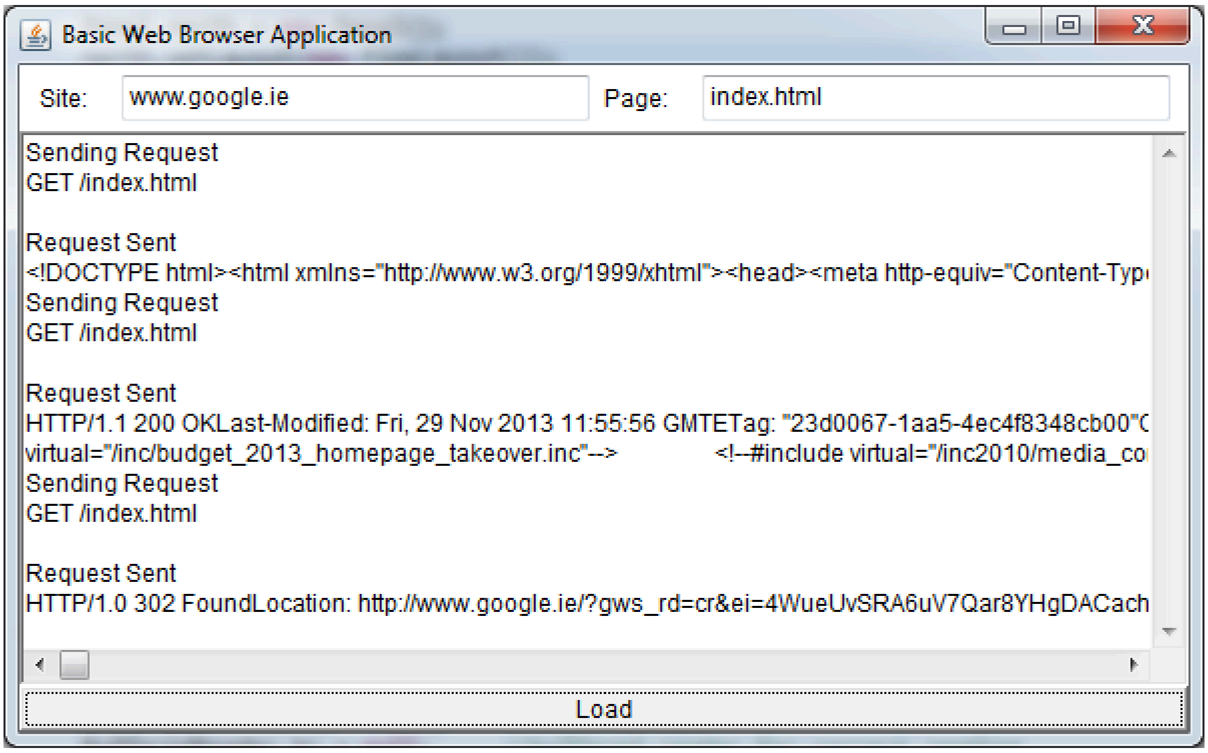


```
c:\temp>java ee402.Finger
Invalid number of paramaters:
Usage: Finger username@host

c:\temp>java ee402.Finger @bathroom.mit.edu
Connecting to bathroom.mit.edu
Random Hall Bathroom Server v2.1

                         Bonfire Kitchen: vacant for 55 min
                         Bonfire  Lounge: vacant for 6 hr
                          Pecker  Lounge: vacant for 32 min
                          Pecker Kitchen: vacant for 3 min
   K 282  L  290 K          Clam Kitchen: vacant for 41 min
   ... ... ... ...          Clam  Lounge: vacant for 19 min
  ¦ o : o ¦ o : o ¦          BMF  Lounge: vacant for 32 min
  ¦ o : o ¦ o : o ¦          BMF Kitchen: vacant for 26 min
  ¦ o : o ¦ x : x ¦         Loop Kitchen: vacant for 7 min
  ¦ o : o ¦ - : x ¦         Loop  Lounge: vacant for 6 hr
 ~~~~~~~~~~~~~~~~~~~~~~ Black Hole  Lounge: *IN*USE* for 12 sec
                      Black Hole Kitchen: *IN*USE* for 16 sec
      o = vacant!        Destiny Kitchen: vacant for 20 min
      x = in use          Destiny Lounge: vacant for 19 min
                                     Foo: *IN*USE* for 5 min

For more information finger help@bathroom.mit.edu

<2149662>

c:\temp>java ee402.Finger help@bathroom.mit.edu
Connecting to bathroom.mit.edu
Random Hall Bathroom Server v2.1

To get information about the status of the Random Hall bathrooms,
you can use either the finger interface or the web interface (both
textual and graphical).

The following textual formats are currently available:
      fusion@bathroom                 http://bathroom/fusion
      text@bathroom                   http://bathroom/text
      graphical@bathroom              http://bathroom/graphical
      old@bathroom                    http://bathroom/old
      plain@bathroom                  http://bathroom/plain
      simple@bathroom                 http://bathroom/simple

In addition, there is a graphical interface available at
http://bathroom.mit.edu/

If you have any questions, feel free to email random-nerds@mit.edu

<2149663>

c:\temp>
```

The Basic Web Browser

Another example of a client application is a basic web browser. This application allows you to connect to any web server (depending on your proxy) and request a web page. The result is returned in HTML and displayed in the text area as shown below

Figure 7.5. The Basic Web Browser Application



```
Sending Request
GET /index.html

Request Sent
<!DOCTYPE html><html xmlns="http://www.w3.org/1999/xhtml"><head><meta http-equiv="Content-Typ
Sending Request
GET /index.html

Request Sent
HTTP/1.1 200 OKLast-Modified: Fri, 29 Nov 2013 11:55:56 GMTETag: "23d0067-1aa5-4ec4f8348cb00"C
virtual="/inc/budget_2013_homepage_takeover.inc"-->           <!--#include virtual="/inc2010/media_co
Sending Request
GET /index.html

Request Sent
HTTP/1.0 302 FoundLocation: http://www.google.ie/?gws_rd=cr&ei=4WueUvSRA6uV7Qar8YHgDACach
```

This application connects to a web site on port 80 and sends the string:

GET /index.html \n\n

where index.html is the page entered in the visual interface. It then reads the response from the server and outputs to the text area.

Here is the source for **BasicWebBrowser.java** that connects to any web server and receives the HTML response.

BasicWebBrowser.java

```java
package ee402;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

@SuppressWarnings("serial")
public class BasicWebBrowser extends Frame implements ActionListener, WindowListener {

    private TextField hostname, page;
    private TextArea returnPage;

    public BasicWebBrowser() {
        super("Basic Web Browser Application");
        this.addWindowListener(this);

        Panel north = new Panel();
        north.setLayout(new FlowLayout());
        hostname = new TextField("www.eeng.dcu.ie",30);
        north.add(new Label("Site:"));
        north.add(hostname);
        page = new TextField("index.html", 30);
        north.add(new Label("Page:"));
        north.add(page);
        this.add(north, BorderLayout.NORTH);

        returnPage = new TextArea(10,40);
        this.add(returnPage, BorderLayout.CENTER);

        Button go = new Button("Load");
        go.addActionListener(this);
        this.add(go, BorderLayout.SOUTH);
        this.setSize(600,350);
        this.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        Socket httpSocket = null;
        DataOutputStream os = null;   //output stream
        DataInputStream is = null;    //input stream
        BufferedReader br = null;            //buffered reader for correct reading

        try {
            httpSocket = new Socket(this.hostname.getText(), 80);   //HTTP port 80
            os = new DataOutputStream(httpSocket.getOutputStream());
            is = new DataInputStream(httpSocket.getInputStream());
            br = new BufferedReader(new InputStreamReader(is));
        } catch (UnknownHostException ex) {
            System.err.println("Don't know host: " + this.hostname.getText());
        } catch (IOException ex) {
            System.err.println("No I/O for the connection to: " +
                                   this.hostname.getText());
        }

        if (httpSocket != null && os != null && is != null) {
           try
           {
               this.returnPage.append("Sending Request\n");
               String theRequest = new String("GET /" + this.page.getText() + "\n\n");
               returnPage.append(theRequest);
                  os.writeBytes(theRequest);

                this.returnPage.append("Request Sent\n");
                // keep reading from/to the socket till we receive "Ok"
                // from HTTP server. Once received then break.

                String responseLine;
                while ((responseLine = br.readLine()) != null) {
                    this.returnPage.append(responseLine);
                    if (responseLine.indexOf("Ok") != -1) {
                      break;
                    }
                }
                this.returnPage.append("End of Response\n");
                os.close();
                is.close();
                httpSocket.close();
           }
           catch (UnknownHostException ex) {
                   System.err.println("Trying to connect to unknown host: " + ex);
           }
           catch (IOException ex) {
                   System.err.println("IOException:  " + ex);
           }
        }
    }

    public static void main(String[] args) {
        new BasicWebBrowser();
    }

    public void windowActivated(WindowEvent arg0) {}
    public void windowClosed(WindowEvent arg0) {}
    public void windowClosing(WindowEvent arg0) {
                System.exit(0);
    }
    public void windowDeactivated(WindowEvent arg0) {}
    public void windowDeiconified(WindowEvent arg0) {}
    public void windowIconified(WindowEvent arg0) {}
    public void windowOpened(WindowEvent arg0) {}
}
```

The TCP Client/Server

Introduction

Some of the general characteristics of clients and server applications are, in general, **client** applications have the following characteristics:

- They are applications that become a client temporarily when remote access is needed, but perform other computation locally.

- They are invoked by a user and executed for one session.
- They run locally on the user's local computer.
- They actively initiate contact with a server and so must know the server details.
- They can access multiple services as required.

In general, **server** applications have the following characteristics:

- They are special-purpose applications dedicated to providing one service.
- They are invoked automatically when a system boots, and continue to execute through many sessions.
- They generally run on a shared computer.
- They wait passively for contact from remote clients.
- They accept contact from clients, but offer a single service.

Note that the word server here is referring to a piece of software. However, a computer running one or more server applications is often also referred to as a server.

We will now write a client/server application pair where:

- The Client will take some data from the user and validate it (as in Finger)
- The Client will then send this data to the Server for processing (as in Finger)
- The Server will process the data and return the results
- The Client will receive the results and display the results in some form.

For our example here We will do this by building a **Date/Time Service**. The client will simply call the server and ask it for the current date and time.

The Date Client/Server Overview

The **Date Server** is possibly the most complex part of this application, in this case consisting of three main classes:

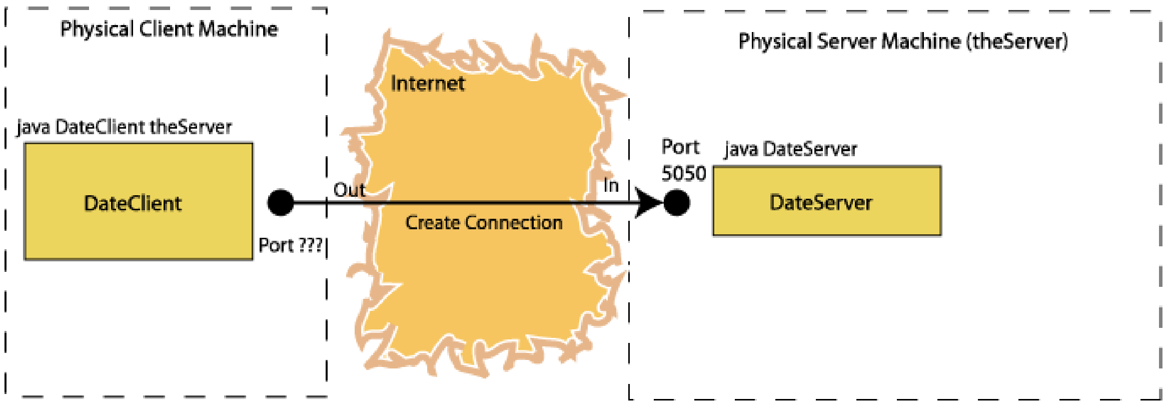
- The DateTimeService class - A basic class to get the current date and time when it is called
- The ConnectionHandler class - This class is used to handle any requests that arrive to the DateServer. This class is specially designed to allow a further transition to a threaded server in later chapters.
- The DateServer class - This is the main server application, that simply listens for connections and creates a ConnectionHandler object when a connection occurs to the server

We will choose the port 5050 as the port to which we send and listen to for our service. Both a client and a server need to create their own sockets. On the server we listen to port 5050.

The **Date Client** is very similar to the Finger client we looked at in the section called "The Finger Client" except that the data transfer uses an Object Stream to send and receive data - we will discuss this later.

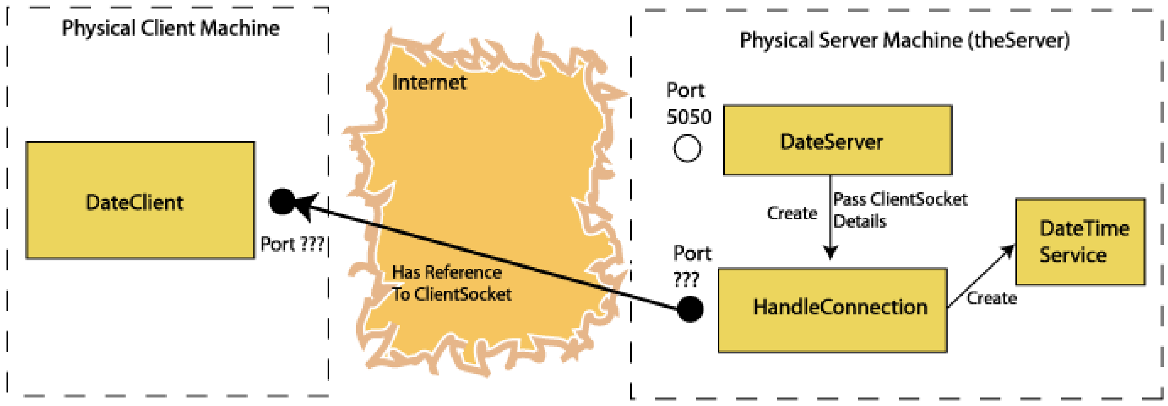
We will first discuss what occurs in the client/server application before looking directly at the code. First off, the server is started by typing the command **java ee402.DateServer** on the BeagleBone/Server machine. The DateServer starts and begins listening for connections on server port 5050. In Figure 7.6, "The Date Client Creates a Connection to the Date Server." the DateClient class is executed by using the command **java ee402.DateClient theServer** where theServer is the hostname of the physical machine on which the DateServer class is located. The DateClient creates a socket and connects to the DateServer server socket running at port 5050. The DateServer should accept this connection provided it is not busy.

Figure 7.6. The Date Client Creates a Connection to the Date Server.



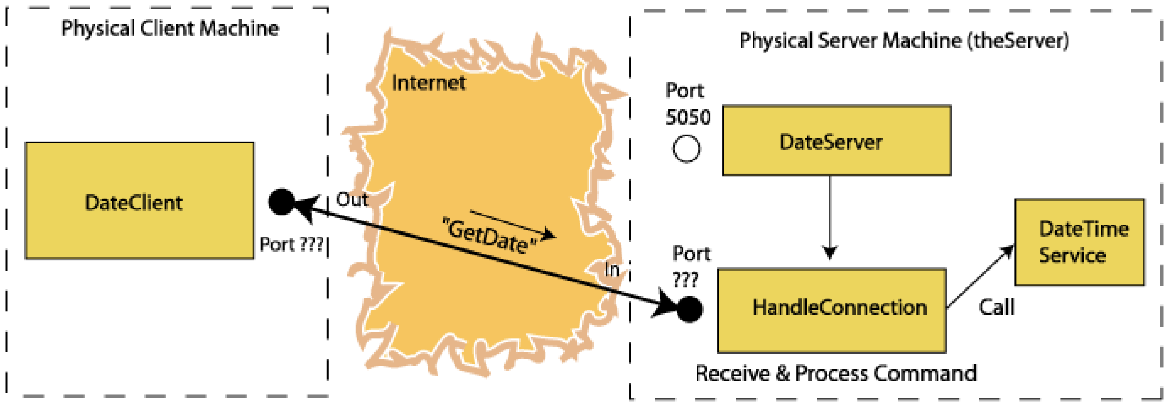
Next, as in Figure 7.7, "The Date Server creates a ConnectionHandler object." the DateServer class creates a ConnectionHandler object to which it passes a reference to the DateClient's socket details. The ConnectionHandler class then creates an instance of the DateTimeService. It also establishes an input/output stream to the DateClient.

Figure 7.7. The Date Server creates a HandleConnection object.



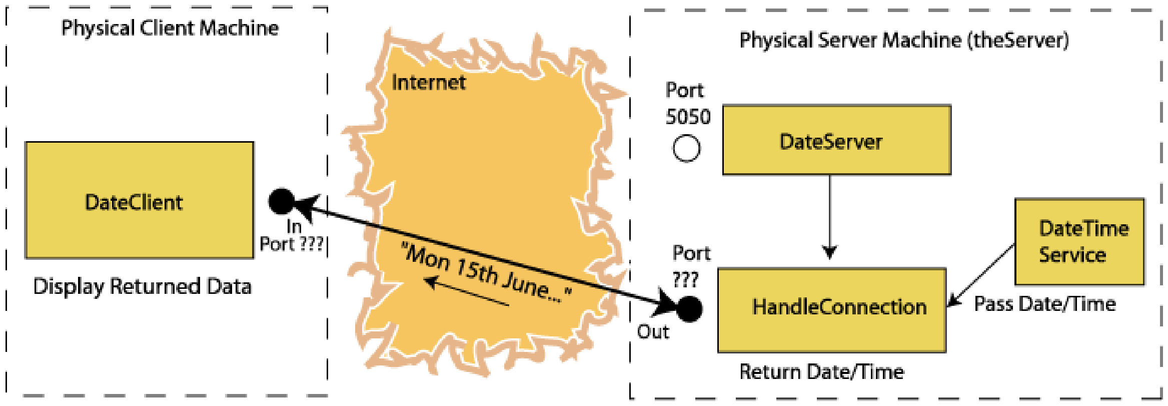
Next, as in Figure 7.8, "The Date Client passes the command to the ConnectionHandler ." the DateClient passes a command to the ConnectionHandler . In this case the command is a String object that contains the text "GetDate". When the ConnectionHandler receives the String object it compares it to see if it is a valid command. In our case the only valid command is "GetDate" so if it is this command then the DateTimeService is called to request the date/time. At this time, since we have not threaded our server application the DateServer is not actually listening to port 5050, rather it is waiting for a response from the ConnectionHandler object.

Figure 7.8. The Date Client passes the command to the HandleConnection.



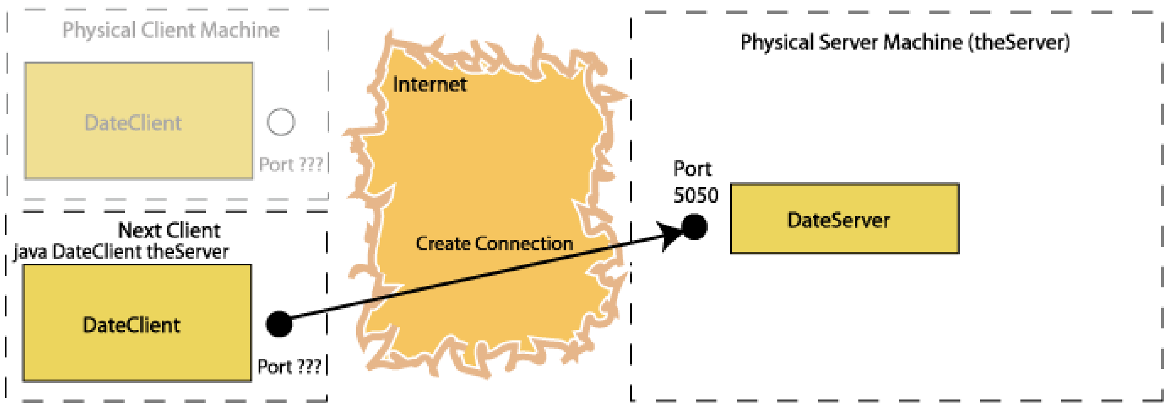
in Figure 7.9, "The ConnectionHandler gets the Date/Time and sends it to the Client." the DateTimeService sends the data/time details back to the ConnectionHandler object where it is then passed on to the DateClient as a String object such as "Mon 15th...". The DateClient will then display the returned data on the client's machine.

Figure 7.9. The ConnectionHandler **gets the Date/Time and sends it to the Client.**



Finally as in Figure 7.10, "The ConnectionHandler shuts down. The DateServer starts listening again." the ConnectionHandler object shuts down connection and is destroyed. Control is returned to the DateServer class and the server once again begins listening to port 5050. The client is now disconnected after having received the data. It now shuts down in our case. The next client is now free to connect to port 5050 on the DateServer and everything happens again. The DateServer will listen forever unless we shut it down by typing CTRL-C in the terminal window.

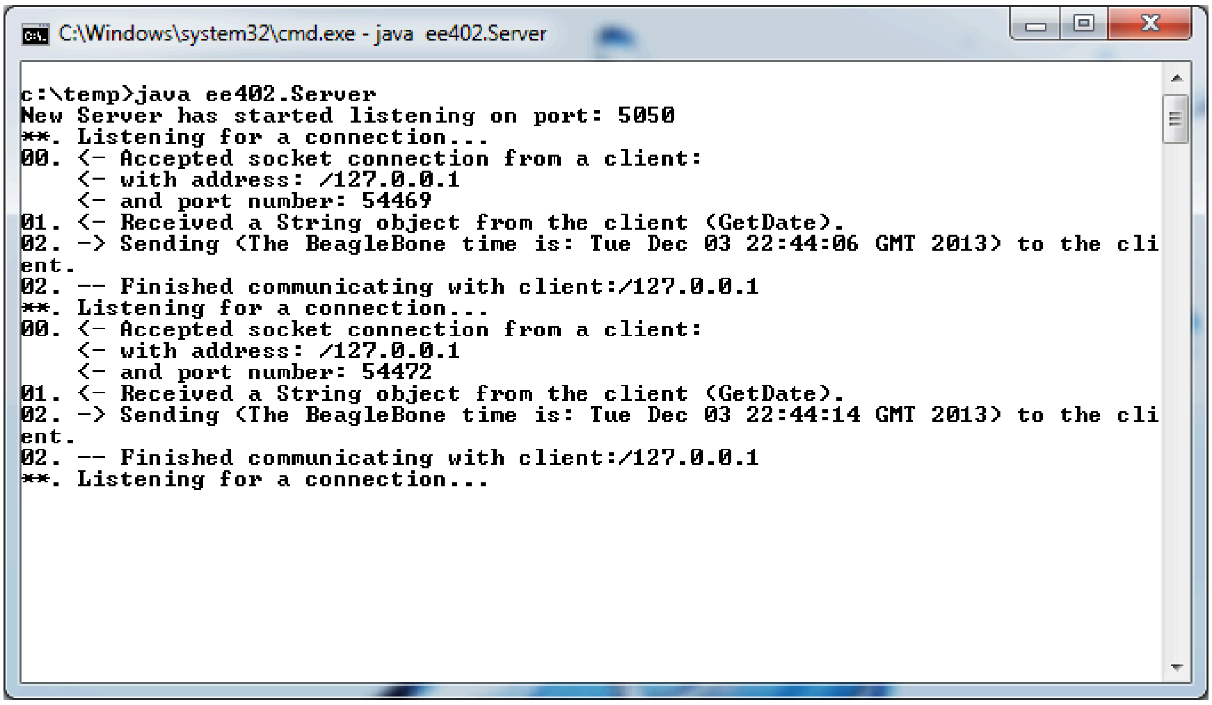
Figure 7.10. The ConnectionHandler **shuts down. The DateServer starts listening again.**



The Running Server/Client Applications

Here you can see the client and server applications working as if you executed them from the command prompt. The DateServer is executed by typing **java ee402.DateServer.** For the assignment you will do this on the BeagleBone Black. There is no need to specify a port number as 5050 is hard coded and there is no need to specify a host name as the code will use the current machine as the server. Figure 7.11, "The DateServer running on "localhost"" shows the DateServer being executed and it stating that it has "started listening on port 5050". The server then accepts different connections from the client application.

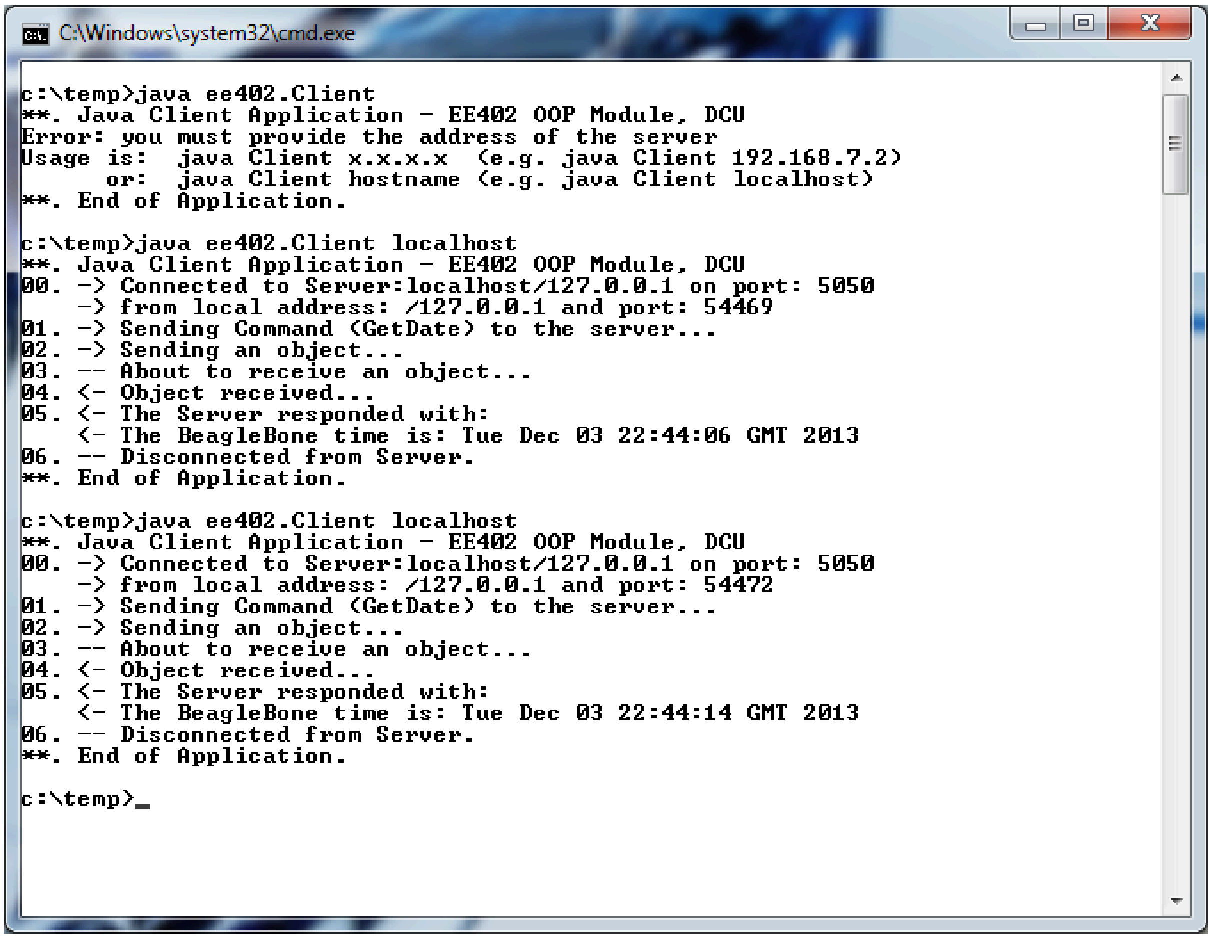
Figure 7.11. The DateServer running on "localhost"



```
c:\temp>java ee402.Server
New Server has started listening on port: 5050
**. Listening for a connection...
00. <- Accepted socket connection from a client:
    <- with address: /127.0.0.1
    <- and port number: 54469
01. <- Received a String object from the client (GetDate).
02. -> Sending (The BeagleBone time is: Tue Dec 03 22:44:06 GMT 2013) to the cli
ent.
02. -- Finished communicating with client:/127.0.0.1
**. Listening for a connection...
00. <- Accepted socket connection from a client:
    <- with address: /127.0.0.1
    <- and port number: 54472
01. <- Received a String object from the client (GetDate).
02. -> Sending (The BeagleBone time is: Tue Dec 03 22:44:14 GMT 2013) to the cli
ent.
02. -- Finished communicating with client:/127.0.0.1
**. Listening for a connection...
```

Figure 7.12, "The Date Client specifying a server at "localhost"" displays the DateClient running multiple times. The first time that the client was run the server name was not specified, so the error was displayed. The next time the client was executed correctly using **java ee402.DateClient localhost** where the server is specified as being on the same machine (we could also have typed 127.0.0.1 instead of localhost).

Figure 7.12. The Date Client specifying a server at "localhost"



```
C:\Windows\system32\cmd.exe

c:\temp>java ee402.Client
**. Java Client Application - EE402 OOP Module, DCU
Error: you must provide the address of the server
Usage is:  java Client x.x.x.x  (e.g. java Client 192.168.7.2)
      or:  java Client hostname (e.g. java Client localhost)
**. End of Application.

c:\temp>java ee402.Client localhost
**. Java Client Application - EE402 OOP Module, DCU
00. -> Connected to Server:localhost/127.0.0.1 on port: 5050
    -> from local address: /127.0.0.1 and port: 54469
01. -> Sending Command (GetDate) to the server...
02. -> Sending an object...
03. -- About to receive an object...
04. <- Object received...
05. <- The Server responded with:
    <- The BeagleBone time is: Tue Dec 03 22:44:06 GMT 2013
06. -- Disconnected from Server.
**. End of Application.

c:\temp>java ee402.Client localhost
**. Java Client Application - EE402 OOP Module, DCU
00. -> Connected to Server:localhost/127.0.0.1 on port: 5050
    -> from local address: /127.0.0.1 and port: 54472
01. -> Sending Command (GetDate) to the server...
02. -> Sending an object...
03. -- About to receive an object...
04. <- Object received...
05. <- The Server responded with:
    <- The BeagleBone time is: Tue Dec 03 22:44:14 GMT 2013
06. -- Disconnected from Server.
**. End of Application.

c:\temp>_
```

The Source Code

There are four classes required for this example, as listed below. Follow these steps:

- Download these classes from github using **git clone git://github.com/derekmolloy/ee402.git** They are in the notes_examples/chapter7 directory.
- Copy the files from ee402/notes_examples/chapter7 to another directory - e.g. c:\temp\ee402
- You can use Eclipse, or compile them by typing **javac *.java** from the c:\temp\ee402 directory.
- You can start the server by going back to the c:\temp directory and typing **java ee402.Server** or it can be run within Eclipse.
- Open a second DOS/Unix command prompt/terminal, go to the c:\temp directory and execute the client by typing **java DateClient localhost** (both applications are on the one machine). If you want to run the client on another PC/BeagleBone Black, simply execute it using **java DateClient the.server.name** where the.server.name is the IP/DNS name of the machine on which the server is running.

Here is the source code for the DateTimeService class that provides the date and time to the ConnectionHandler class - as below:

DateTimeService.java

```java
/* The Date Time Service Class - Written by Derek Molloy for the EE402 Module
* See: ee402.eeng.dcu.ie
*/

package ee402;

import java.util.Calendar;
import java.util.Date;

public class DateTimeService
{
   private Calendar calendar;

   //constructor creates the Calendar object, could use the constructor:
   //   Calendar(TimeZone zone, Locale aLocale) to explicitly specify
   //          the time zone and locale
   public DateTimeService()
   {
       this.calendar = Calendar.getInstance();
   }

   //method returns date/time as a formatted String object
   public String getDateAndTime()
   {
       Date d = this.calendar.getTime();
        return "The BeagleBone time is: " + d.toString();
   }
}
```

Here is the source code for the DateServer class that runs the server application and creates an instance of the ConnectionHandler whenever a connection is made by a client - as below:

Server.java

```java
/* The Date Server Class - Written by Derek Molloy for the EE402 Module
* See: ee402.eeng.dcu.ie
*/

package ee402;

import java.net.*;
import java.io.*;

public class Server
{
    private static int portNumber = 5050;

    public static void main(String args[]) {

        boolean listening = true;
        ServerSocket serverSocket = null;

        // Set up the Server Socket
        try {
          serverSocket = new ServerSocket(portNumber);
           System.out.println("New Server has started listening on port: " + portNumber
);
        }
        catch (IOException e)
        {
            System.out.println("Cannot listen on port:" + portNumber + ",Exception:" +
e);
          System.exit(1);
        }

        // Server is now listening for connections or would not get to this point
        while (listening) // almost infinite loop - loop once for each client request
        {
            Socket clientSocket = null;
            try{
                System.out.println("**. Listening for a connection...");
                clientSocket = serverSocket.accept();
                  System.out.println("00. <- Accepted socket connection from a client:
");
                System.out.println("    <- with address: " +
                                    clientSocket.getInetAddress().toString());
                                System.out.println("       <-  and  port  number:  "  +
clientSocket.getPort());
            }
            catch (IOException e){
                System.out.println("XX. Accept failed: " + portNumber + e);
                listening = false;   // end loop - stop listening for further requests
            }

            ConnectionHandler con = new ConnectionHandler(clientSocket);
            con.init();
            System.out.println("02. -- Finished communicating with client:"
                        + clientSocket.getInetAddress().toString());
        }
        // Server is no longer listening for client connections - time to shut down.
        try
        {
            System.out.println("04. -- Closing down the server socket gracefully.");
            serverSocket.close();
        }
        catch (IOException e)
        {
            System.err.println("XX. Could not close server socket. " + e.getMessage());
        }
    }
}
```

Here is the source code for the ConnectionHandler class that is created by the DateServer class whenever a connection is received. This class is responsible for dealing with each of the client requests:

ConnectionHandler.java

```java
/* The Connection Handler Class - Written by Derek Molloy for the EE402 Module
 * See: ee402.eeng.dcu.ie
 */

package ee402;

import java.net.*;
import java.io.*;

public class ConnectionHandler
{
    private Socket clientSocket = null;                         // Client socket object
    private ObjectInputStream is = null;                        // Input stream
    private ObjectOutputStream os = null;                       // Output stream
    private DateTimeService theDateService;

        // The constructor for the connection handler
    public ConnectionHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
        //Set up a service object to get the current date and time
        theDateService = new DateTimeService();
    }

    // Will eventually be the thread execution method - can't pass the exception back
    public void init() {
         try {
            this.is = new ObjectInputStream(clientSocket.getInputStream());
            this.os = new ObjectOutputStream(clientSocket.getOutputStream());
            while (this.readCommand()) {}
         }
         catch (IOException e) {
            System.out.println("XX. There was a problem with the
               Input/Output Communication:");
            e.printStackTrace();
         }
    }

    // Receive and process incoming string commands from client socket
    private boolean readCommand() {
        String s = null;
        try {
            s = (String) is.readObject();
        }
        catch (Exception e) {    // catch a general exception
            this.closeSocket();
            return false;
        }
        System.out.println("01. <- Received a String object
                    from the client (" + s + ").");

        // At this point there is a valid String object
        // invoke the appropriate function based on the command
        if (s.equalsIgnoreCase("GetDate")){
            this.getDate();
        }
        else {
            this.sendError("Invalid command: " + s);
        }
        return true;
    }

    // Use our custom DateTimeService Class to get the date and time
    private void getDate() {          // use the date service to get the date
        String currentDateTimeText = theDateService.getDateAndTime();
        this.send(currentDateTimeText);
    }

    // Send a generic object back to the client
    private void send(Object o) {
        try {
            System.out.println("02. -> Sending (" + o +") to the client.");
            this.os.writeObject(o);
            this.os.flush();
        }
        catch (Exception e) {
            System.out.println("XX." + e.getStackTrace());
        }
    }

    // Send a pre-formatted error message to the client
    public void sendError(String message) {
        this.send("Error:" + message);        //remember a String IS-A Object!
    }

    // Close the client socket
    public void closeSocket() { //gracefully close the socket connection
        try {
            this.os.close();
            this.is.close();
            this.clientSocket.close();
        }
        catch (Exception e) {
            System.out.println("XX. " + e.getStackTrace());
        }
    }
}
```

Finally the Client application:

Client.java

```java
/* The Client Class - Written by Derek Molloy for the EE402 Module
* See: ee402.eeng.dcu.ie
*
*
*/

package ee402;

import java.net.*;
import java.io.*;

public class Client {
    private static int portNumber = 5050;
    private Socket socket = null;
    private ObjectOutputStream os = null;
    private ObjectInputStream is = null;

        // the constructor expects the IP address of the server - the port is fixed
    public Client(String serverIP) {
       if (!connectToServer(serverIP)) {
          System.out.println("XX. Failed to open socket connection to: " + serverIP);
       }
    }

    private boolean connectToServer(String serverIP) {
       try { // open a new socket to the server
           this.socket = new Socket(serverIP,portNumber);
           this.os = new ObjectOutputStream(this.socket.getOutputStream());
           this.is = new ObjectInputStream(this.socket.getInputStream());
           System.out.println("00. -> Connected to Server:" +
                  this.socket.getInetAddress()+ " on port: " + this.socket.getP ort());
           System.out.println("    -> from local address: " +
                   this.socket.getLocalAddress()
                   + " and port: " + this.socket.getLocalPort());
         }
        catch (Exception e) {
           System.out.println("XX. Failed to Connect to the Server at port: "
              + portNumber);
           System.out.println("    Exception: " + e.toString());
           return false;
        }
        return true;
    }

    private void getDate() {
       String theDateCommand = "GetDate", theDateAndTime;
       System.out.println("01. -> Sending Command (" + theDateCommand
                 + ") to the server...");
       this.send(theDateCommand);
       try{
          theDateAndTime = (String) receive();
          System.out.println("05. <- The Server responded with: ");
          System.out.println("    <- " + theDateAndTime);
       }
       catch (Exception e){
          System.out.println("XX. There was an invalid object
                     sent back from the server");
        }
        System.out.println("06. -- Disconnected from Server.");
    }

    // method to send a generic object.
    private void send(Object o) {
        try {
           System.out.println("02. -> Sending an object...");
           os.writeObject(o);
           os.flush();
        }
        catch (Exception e) {
            System.out.println("XX. Exception Occurred on Sending:" +  e.toString());
        }
    }

    // method to receive a generic object.
    private Object receive()
    {
        Object o = null;
        try {
            System.out.println("03. -- About to receive an object...");
            o = is.readObject();
            System.out.println("04. <- Object received...");
        }
        catch (Exception e) {
                       System.out.println("XX.  Exception  Occurred  on  Receiving:"  +
e.toString());
        }
        return o;
    }

    public static void main(String args[])
    {
        System.out.println("**. Java Client Application - EE402 OOP Module, DCU");
        if(args.length==1){
           Client theApp = new Client(args[0]);
           theApp.getDate();
        }
        else
        {
           System.out.println("Error: you must provide the address of the server");
           System.out.println("Usage is:java Client x.x.x.x (e.g.
                                java Client 192.168.7.2)");
           System.out.println("      or:  java Client hostname
                                (e.g. java Client localhost)");
        }
        System.out.println("**. End of Application.");
    }
}
```

Serialization

Serialization is best described by its absence. In the previous section, we passed String objects from the client to the server and back again and we were not concerned with the transport system.

That is a correct approach, however, things get a little bit more difficult when we want to send objects that consist of a number of states (i.e. is-a-part-of values). Before the advent of serialization we (as the programmer) were required to break an object into its component parts and send them one-by-one over the stream, where they would then be re-constructed one-by-one.

So for example, if an object Fish contained a name and a type String states, these would have to be written one at a time to the stream using a method like writeUTF(). The server would then have to read these String objects one at a time and re-construct the Fish object. However, this is prone to error; Suppose the order of string reading was reversed on the server, then the fish would have the name of type and the type of name.

Object serialization makes this task straightforward, as we simply add to the code the fact that the Fish class implements Serializable and almost everything else is taken care of.

The serialization mechanism used is capable of handling a wide variety of situations. When you serialize an object you save all the states of the object. This even includes states marked private. However, there are times when you don't want a state to be persistent, especially if the object is tied to resources that are specific to this session of the virtual machine, it does not make any sense to serialize the object for later use. Fortunately, the Java programming language includes the declaration transient. A state marked transient means that the state is not saved when an object is serialized. Static data is also not serializable, and so is ignored.

If the private states in your class include sensitive data, e.g. credit card numbers, passwords etc. you should not serialize them as this data can be plainly read in the stream. The simplest way to remove this data is to use private transient.

```java
import java.io.*;               // contains the Serializable interface

 public class Fish implements Serializable
 {
      private String name;
      private String type;

      public Fish(String name, String type)
      {
             this.name = name;
             this.type = type;
      }

      public String getName() { return this.name;   }

      public String getType() { return this.type;   }
 }
```

The Fish class implements the Serializable interface, which allows the data of the class to be ordered in a serial form, to send it across a serial network (of sorts). The Serializable interface is simple to implement, as we do not have to write any additional code. The Serializable interface does not have any methods, as it is simply a "signal" interface that indicates to the JVM that you want to use the default serialization mechanism.

On the client-side we can create a fish object (say myFish) and send it using the send(Object o) method discussed previously. In the ConnectionHandler class we can read in each object, check to see what class it is, and then cast convert the object to a Fish.

So, for example:

```
if (o.getClass().getName().equals("Fish"))
 {
      Fish aFish = (Fish) o;
 }
```

Note: If the Fish class is in the ee402 package, then you may have to include the package name in the cast, for example:

```
Fish aFish = (ee402.Fish) o;
```

Once we have converted to the Fish object we can simply call the usual methods:

```
System.out.println("\nFish name is: "+ aFish.getName());
System.out.println("\nFish type is: "+ aFish.getType());
```

Once an object is written to a stream, its state is fixed. In effect, a copy of the object is created on the client side and sent to the server side. If this object changes state on the client side, while the server is busy processing the copy, then the copy is no longer up-to-date. This can be problematic. One solution is to lock the client object on the client side, so that it cannot be altered while the copied object is in transit to or from the server - but what happens if the server or communication fails?

The use of serialization also allows easy persistent storage through the use of files. For example if we wished to store the Fish objects to a file we could use:

```
FileOutputStream out = new FileOutputStream("Storage.file");
ObjectOutput s = new ObjectOutputStream(out);
s.writeObject("Some example string");            // store an example String
s.writeObject(new Fish("Jim", "Guppy"));  // store an example Fish
s.flush();
```

That allows us to store a String object or a Fish object using the same writeObject() method. To read in from this file Storage.file we could use:

```
    FileInputStream in = new FileInputStream("Storage.file");
    ObjectInputStream s = new ObjectInputStream(in);
    String theString = (String) s.readObject();
    Fish myFish = (Fish) s.readObject();
```

As the communication becomes more complex, the serialization task becomes even more difficult. Serialization, although providing a huge advantage over manually streaming data, still requires the programmer to maintain the protocol for communication, distributing the protocols and making sure that both the client and server are aware of the serialized object class (in this case Fish), allowing the casting to take place.

Exercise

Task: Modify the code from the section called "The TCP Client/Server" to send a Fish object across the network and display it on the server side. Use these code samples:

- **Server.java**
- **ConnectionHandler.java**
- **Client.java**

Chapter 8 - Threads and Networking

This section assumes that you have completed Chapter 7 - Java Network Applications. The client-server application that was discussed in that chapter worked well; however, there was one main difficulty. The server could only handle one client connection at a time, i.e., a connection is made and the server is tied to that client. The service is performed and then when complete, the client is released and the server is made available for more client connections. In this section we want to develop a threaded server application that will allow multiple client connections at the same time.

Threads - An Introduction

Introduction

We often require to break our application into a number of independent sub-tasks, called threads (threads of execution). We write each thread in an individual manner and assume that there is some mechanism for dividing up the CPU time and sharing it between these threads. Threads enhance performance and functionality, by allowing a program to perform multiple tasks simultaneously.

Why would we want to do this? Well, take the example of a web browser. It allows us to download a web page and at the same time interact with the window, and even scroll through the text reading what has already started downloading. Threading is supported directly in the Java language, allowing us to write platform independent threaded applications (There are some difficulties when dealing with the different operating systems as there are slight behaviour differences).

Multiprocessing vs. Multithreading

Multiprocessing refers to multiple applications, executing 'apparently' concurrently, whereas **multithreading** refers to one or more tasks within an application executing 'apparently' concurrently. The word 'apparently' is used as the platform usually has a single CPU that must be shared. If there are multiple CPUs it is possible that processes might actually be executing concurrently. This is the case with more recent Intel processors, such as the i3 to i9 where there are multiple cores.

Multiprocessing is a heavyweight process, under the control of the operating system. The different applications have no relationships with each other. Multithreading can produce programs that are very efficient, since work can be performed during low CPU actions, such as waiting for the user to enter data, or waiting for a file to load. Multithreaded programs can be concurrently executed in a multiprocessing system. They can exist concurrently with each other.

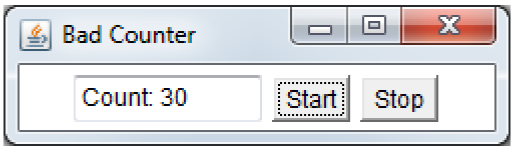
Why use threads?

Suppose we wish to write an application that performs a CPU-intensive operation. It is possible that this application will become unresponsive as the application ignores user input. For example, this counter application simply counts forever, outputting the count in a Textfield component.

Figure 8.1, "The Bad Counter Application Running" displays a capture of an application that does not work correctly. You can press the Start button to start the count but the Stop button will not work to stop the application from counting.

Note: This application does not work correctly - It may even have to be killed by using your task manager. Under Windows press CTRL-SHIFT-ESC together to open the task manager.

Figure 8.1 The Bad Counter Application Running



Here is the source code for this example. If you have not studied threads before then it will not be immediately obvious why this application does not work correctly.

BadCounterApp.java

```java
/** Bad Counter Application Example - Derek Molloy, Dublin City University
 *   This code will not work correctly and the question is why?
 *   I will correct this in the next version.
 */

package ee402;

import java.awt.*;
import java.awt.event.*;

@SuppressWarnings("serial")
public class BadCounterApp extends Frame implements ActionListener {

	private int count = 0;
	private Button start, stop;
	private TextField countText;
	private boolean running = true;

	public BadCounterApp(){
		super("Bad Counter");
		this.setLayout(new FlowLayout());
		this.countText = new TextField(10);
		this.add(countText);
		this.start = new Button("Start");
		this.add(start);
		this.start.addActionListener(this);
		this.stop = new Button("Stop");
		this.add(stop);
		this.stop.addActionListener(this);
		this.pack();
		this.setVisible(true);
	}

	public void go(){
		while(running){
			this.countText.setText("Count: " + count++);
			try{
				Thread.sleep(100);
			}
			catch(InterruptedException e){
				System.out.println("Thread was Interrupted!");
			}
		}
	}

	public static void main(String[] args) {
		new BadCounterApp();
	}

	public void actionPerformed(ActionEvent e) {
```

```
                if (e.getSource().equals(start)){
                        this.go();
                }
                else if (e.getSource().equals(stop)){
                        this.running = false;
                }
        }
}

```

So why does it not work? When you run the application you will see that it starts counting as expected with a delay between each number - so what is wrong? Well if we look at the code step by step:

- First off, the application creates an instance of Frame, draws the buttons, textfield and is working perfectly. The application is now waiting for a button to be pressed. No problems so far.
- Next off, the "Start" button is pressed by the user. This event is sent to the BadCounter's actionPerformed() since we have added a listener to the "Start" button using the addActionListener(). No problems so far.
- The actionPerformed() method calls the go() method and awaits a response. But this is where are problem is - the go() method has a loop - that says "while running is true sleep for a while (100ms) and then update the value in the countText TextField". **Since this loop goes around forever control will never be returned to the actionPerformed() method from the call to go().**
- The only thing that is happening at the moment is that the loop is looping - The application has stopped listening for events as it is trapped within the loop, within the call to the go() method, within the actionPerformed() method.
- Even though the code is correct for the "Stop" button **it can never be pressed** - You will notice that in the running application the buttons don't even go up and down when you press them.
- We need some way of having a loop (infinite or other) and also listening for events - we need threads!

Threads - A better way?

We want to re-write the application in the previous section so that we can do this count operation, while still providing access to the user interface. This is not much different than most threaded applications, where we might define several threads:

- One thread to handle the keyboard input.
- One thread to handle a file load.
- One thread to handle some complex computation.

The computation thread could be given priority once the load or input threads are blocked, waiting for communication from the keyboard or the disk drive. Sequential programs do not have this facility. They cannot perform any operations while the user is deciding what to type.

We have two ways of creating threads in Java. We can extend the Thread class, or we can implement the Runnable interface.

Extending the Thread class

The first way is to extend the Thread class to create our own Thread class, and then providing specific functionality by overriding the run() method - such as:

```java


public class MyThread extends Thread
{
        public void run()
        {
                //add your implementation here
        }
}

```

Implementing the Runnable Interface

Implementing the Runnable interface allows us more flexibility than extending the Thread as we our new class is still able to inherit from any parent class, without that parent having to be Thread. The Runnable interface has one method that we must implement - run(). So our thread might look like:

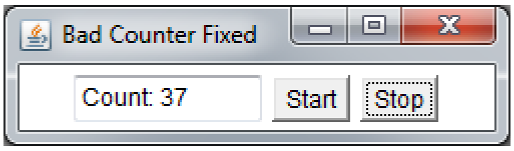
```java

public class MyThread extends WhateverClass implements Runnable
{
        public void run()
        {
                //add your implementation here
        }
}


```

A Working Counter

This section fixes the counter applet discussed in the section called “Why use threads?” to allow it to work correctly. Figure 8.2, “The Bad Counter Fixed Application Running” displays a capture of a working version of the application.

You will notice that when you press the "Start" button the counter starts, but you are still able to press the "Stop" button and it does indeed stop the counter. I have not written the code to allow you to restart the counter - for simplicity at this point.

Figure 8.2 The Bad Counter Fixed Application Running



The source code is below, with the main changes highlighted in yellow:

BadCounterAppFixed.java

```java
/** Bad Counter Fixed Application Example - Derek Molloy, Dublin City 
University
*   This code fixes the previous non-working version by adding threading
*/

package ee402;

import java.awt.*;
import java.awt.event.*;

@SuppressWarnings("serial")
public class BadCounterAppFixed extends Frame implements ActionListener, 
Runnable {

        private int count = 0;
        private Button start, stop;
        private TextField countText;
        private boolean running = true;
        private Thread thread;

        public BadCounterAppFixed(){
                super("Bad Counter Fixed");
                this.setLayout(new FlowLayout());
                this.countText = new TextField(10);
                this.add(countText);
                this.start = new Button("Start");
                this.add(start);
                this.start.addActionListener(this);
                this.stop = new Button("Stop");
                this.add(stop);
                this.stop.addActionListener(this);
                this.pack();
                this.setVisible(true);
                this.thread = new Thread(this);
        }

        public void run(){
                while(running){
                        this.countText.setText("Count:  " + count++);
                        try{
                                Thread.sleep(100);
                        }
                        catch(InterruptedException e){
                                System.out.println("Thread was Interrupted!");
                        }
                }
        }

        public static void main(String[] args) {
                new BadCounterAppFixed();
        }

        public void actionPerformed(ActionEvent e) {
                if (e.getSource().equals(start)){
                        this.thread.start();
                }
                else if (e.getSource().equals(stop)){
                        this.running = false;
                }
        }
}
```

So why does this version work? Well I have made a few changes:

- A new state is added to the class, thread Thread object. This is our reference to the new thread object that we are about to create.
- The BadCounterAppFixed class implements the Runnable interface, and so was required to write a run() method. This means that every class that implements Runnable **must** have written a run() method.
- In the init() method we create an object for the thread reference using new Thread(this);, which creates a new Thread object out of the current object this. The Thread constructor that we use is Thread(Runnable target). The only reason that we are allowed to pass this to the constructor is because this class implements the Runnable interface.
- Instead of calling go() in the previous case, this time the actionPerformed() method calls theThread.start() method when the "Start" button is pressed. **Note that we call the start() method and NOT run() method that was just discussed.** The call to start() asks the thread manager to invoke this thread's run() method 'when it gets the chance'. This means that when start() is called - control is returned **immediately** to the the next line of actionPerformed()and the thread manager invokes a separate thread of execution at the same time for the while(running) method.

Threads - More Aspects

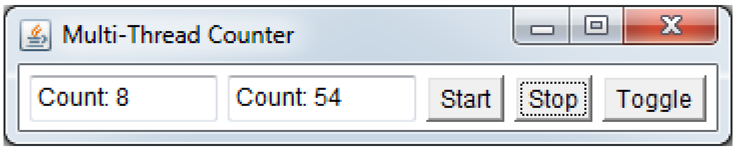
Suspending and Resuming Threads

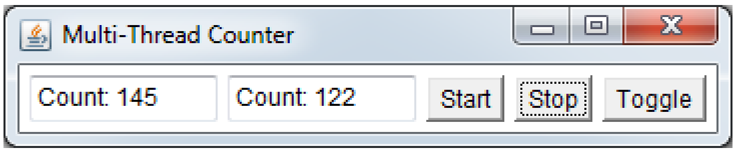
Suspending a thread is when we temporarily pause a thread, rather than stop and destroy it. This thread can be resumed at a later stage.

When Java was first introduced threads were a part of the language and worked fine. There were some difficulties with multi-Platform versions. With the introduction of Java 2 threads were updated to prevent certain race and lock conditions that occurred. In the Thread class there were suspend() and resume() methods, but these have been deprecated for JDK1.2+ and so should not be used - you will get deprecated warnings from the Java compiler if used, and they will lead to unpredictable threaded code if these warnings are ignored.

Figure 8.3, "A Multi-Threaded Counter" displays a capture of an application running in which there are two TextFields objects that are controlled by the same buttons, but they are different threads that start at different points 0 and 50 and count at different rates. The first one counts every 100ms, whereas the second counts every 200ms (i.e. at half the speed).

Figure 8.3 A Multi-Threaded Counter (a) shows the count values at the start, (b) shows the count values after a few seconds, where the first counter has passed the second counter.





The source code is as below:

CounterApplication1.java

```java
/** The Multi-Thread Counter Example - Derek Molloy, Dublin City University
*
*/

package ee402;

import java.awt.*;
import java.awt.event.*;

class Counter extends Thread{
	private int count = 0, delay;
	private boolean running = true, paused = false;
	private TextField outputField;

	public Counter(TextField t, int delay, int startAt){
		this.outputField = t;
		this.delay = delay;
		this.count = startAt;
	}

	public void run(){
		while(running){
			this.outputField.setText("Count: " + count++);
			try{
			   Thread.sleep(this.delay);
			   synchronized(this){
			      while (this.paused) wait();
			   }
			}
			catch (InterruptedException e){
			   System.out.println("Counter was Interrupted!");
			   this.running = false;
			}
		}
	}

	public void stopCount() { this.running = false; }

	public void toggleCount() {
		synchronized(this){
			this.paused = !this.paused;
			if (!this.paused) this.notify();
		}
```

```java
        }
}

@SuppressWarnings("serial")
public class CounterApplication1 extends Frame implements ActionListener{

        private Counter count1, count2;    //two separate threads
        private TextField count1Field, count2Field;
        private Button start, stop, toggle;

        public CounterApplication1() {
                super("Multi-Thread Counter");

                this.count1Field = new TextField(10);
                this.count2Field = new TextField(10);
                // delay 100ms start at 0
                this.count1 = new Counter(count1Field, 100, 0);
                // delay 200ms start at 50
                this.count2 = new Counter(count2Field, 200, 50);
                this.start = new Button("Start");
                this.stop = new Button("Stop");
                this.toggle = new Button("Toggle");

                this.start.addActionListener(this);
                this.stop.addActionListener(this);
                this.toggle.addActionListener(this);

                this.setLayout(new FlowLayout());
                this.add(this.count1Field);
                this.add(this.count2Field);
                this.add(this.start);
                this.add(this.stop);
                this.add(this.toggle);

                this.pack();
                this.setVisible(true);
        }

        public void actionPerformed(ActionEvent e) {
                if (e.getSource().equals(start)){
                        this.count1.start();
                        this.count2.start();
                }
                else if (e.getSource().equals(stop)){
                        this.count1.stopCount();
                        this.count2.stopCount();
                }
                else if (e.getSource().equals(toggle)){
                        this.count1.toggleCount();
                        this.count2.toggleCount();
                }
        }

        public static void main(String[] args) {
                new CounterApplication1();
        }
}


```

I have changed the code from the previous section to have two separate classes - the application and the counter. I did this to prevent any confusion between the Frame class and the thread itself, and to allow for the creation of two separate thread objects. The Frame class creates three buttons as in Figure 8.3, "A Multi-Threaded Counter Example" and two TextField objects. So how does this code work?

- The CounterApplication1 creates two threads count1 that passes a reference to the count1Field, sets the counter to 0 and the delay between counting to 100ms and count2 that passes a reference to the count2Field and sets the counter to 50 and the delay between counting to 200ms.
- The actionPerformed() method handles the start button, that calls the start() method of both Counter objects. The "Stop" button and the "Toggle" button call the stopCount() and toggleCount() methods of both Counter objects.
- The Counter class extends the Thread class and overrides the run() inherited from Thread.
- The Counter class has a single constructor that requires a TextField reference, an int delay and an int starting count value.
- There are two boolean states running and paused. running is a state that defines if the run() method is looping. The paused state defines if the wait() is to be called during the loop.
- To stop the counter the stopCount() can be called from outside the class. The stopCounter simply sets the running state to false, so that the next time the loop runs to completion, the while(running) no longer is true and so the loop ends. Once the thread has been stopped it has run to completion and cannot be re-started as the object has been destroyed.
- The toggleCounter() is a good bit more complex. The idea is straightforward - when the toggleCounter() is called the paused state changes from true to false and then back again the next time it is called. When the paused state is true then the Thread's wait() method is called and so the loop pauses at this very point. In addition, if the paused state is true when the "toggle" button is pressed, then we must unpause the counter. To do this we must call the Thread's notify() method to let it know that there has been a change in the states of the class. We must also use the synchronized modifier on the wait() and notify() methods. This is required by the language, and ensures that wait() and notify are properly synchronized, eliminating the race conditions that could cause the "suspended" thread to miss a notify and remain suspended indefinitely. Rather than place the synchronized on the entire run() method, we have localised it to the wait() method. We will discuss **synchronization** shortly.

The wait() method (from the Object class) causes a thread to release the lock it is holding on an object - allowing another thread to run. It can only be invoked from within a block of synchronized code and should be wrapped in a try block as it throws an IOException. There are three different wait() methods:

- wait() wait for ever!
- wait(long timeout) with a timeout

- wait(long timeout, int nanos) with the time measured in nanoseconds

The wait() can only be invoked by the thread that currently owns the lock on the object. Once the wait() is called the thread becomes disabled for scheduling and is dormant until one of the following things happens:

- Some other thread invokes the notify() method for this object and the scheduler runs the thread.
- Some other thread invokes the notifyAll() method for this object and the scheduler runs the thread.
- Some other thread interrupts this thread.
- If a time was provided, and it has elapsed.

Once one of these things happens the thread then becomes available to the scheduler.

The notify() and notifyAll() methods are defined in the Object class. Like the wait() method, they can only be used within synchronized code. The notify() method wakes up a single thread which is waiting on the object's lock. If there was more than one thread waiting then on the object's lock then the choice of which waiting thread should be chosen is arbitrary - notifyAll() awakens all threads waiting on the object's lock and the scheduler will decide which one to run. If you call notify() on an object with no waiting threads, then the call will just be ignored.

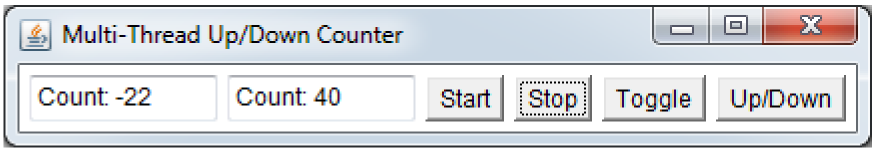
Try this yourself! - An Up/Down Counter

Task: Modify the code as shown in Figure 8.3, "A Multi-Threaded Counter Example" to add the following functionality:

- Add a new button called Up/Down that when pressed causes the two threaded counters to count the opposite way.

You can see a screen-grab of my solution in Figure 8.4, "My Up/Down Counter Example".

Figure 8.4 The Up/Down Counter Example, for you to try yourself.



Solution: The solution is here - UpDownCounterApp.java but please do not look at it until you have had a good attempt yourself.

Scheduling of Threads

Java has thread scheduling that monitors all running threads in all programs and decides which thread should be running. There are two main type of thread:

- **Priority Threads** - Are regular, user-defined threads.

- **Daemon Threads** - Are low-priority threads (often called service threads) that provide services to programs, when the load on the CPU is low. The Garbage Collector Thread is an example of a daemon thread. It is possible for use to convert a user thread into a daemon thread, or vice-versa, using the setDaemon(boolean) Thread method. If there are only daemon threads running, the scheduler will exit.

There are two main forms of scheduler: (i) **Preemptive** that gives a certain time slice to each thread. The scheduler sets up the order that the threads run in. (ii) **Non-preemptive** that runs a thread until it is complete. Each thread has control of the processor for as long as it requires.

New threads inherit the priority and daemon flag from the thread that created it.

Thread Priorities

The scheduler decides which thread should be running based on a priority value assigned to the thread. The priority number has a value between 1 and 10 and a thread runs more often if it has a higher priority value. There are three pre-defined priorities:

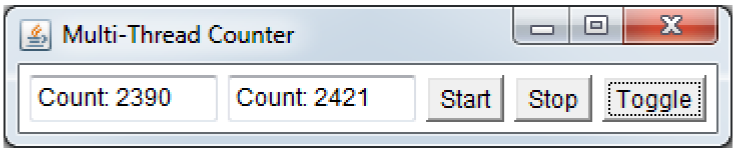
- Thread.MIN_PRIORITY - has the priority value of 1.
- Thread.NORM_PRIORITY - Normally a thread has a priority value of 5.
- Thread.MAX_PRIORITY - has the priority value of 10.

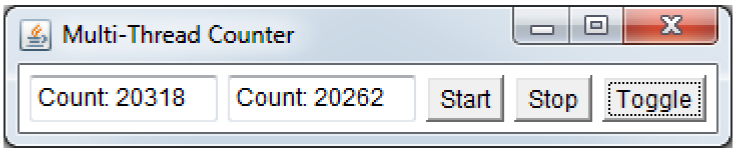
You can use the setPriority() to set the priority level of a thread, so if you wished to set a Thread object called testThread to be running at the highest priority possible, you could call testThread.setPriority(Thread.MAX_PRIORITY); The method getPriority can be used to return the priority level of a given thread. It returns an int value, as per the list above.

An Example of Priorities and Threads

I have written a short example based on the code in the previous section to show you the effect of priorities on threads. This example sets the first Counter object to have the maximum priority and the second Counter object to have the minimum priority. The first counter starts with a value of 0 and the second counter starts with the higher value of 200. I set the delay between counting to be 1ms, in the hope that my machine will not be able to handle that speed of counting and so will give more priority to the first counter. You can see the results in Figure 8.5, where a screen-grab of this example running shows that the first counter has caught up with the second counter after ~20300ms, i.e. around 20 seconds.

Figure 8.5 The Counter Example, with different thread priorities (a) you can see that count 1 has a lower value but at a later time in (b) you can see that counter 1 has a higher value than counter 2.





The code for this example is below, with the changes highlighted in yellow (please note, I am using the same Counter class, I have had to rename it in order that I can distributed the notes in a single ee402 package):

CounterApplication2.java

```java
...

@SuppressWarnings("serial")
public class CounterApplication2 extends Frame implements
ActionListener{

        private Counter2 count1, count2;    //two separate threads
        private TextField count1Field, count2Field;
        private Button start, stop, toggle;

        public CounterApplication2() {
                super("Multi-Thread Counter");

                this.count1Field = new TextField(10);
                this.count2Field = new TextField(10);
                // delay 1ms start at 0
                this.count1 = new Counter2(count1Field, 1, 0);
                this.count1.setPriority(Thread.MAX_PRIORITY);
                // delay 1ms start at 50
                this.count2 = new Counter2(count2Field, 1, 50);
                this.count2.setPriority(Thread.MIN_PRIORITY);
                this.start = new Button("Start");
                this.stop = new Button("Stop");
                this.toggle = new Button("Toggle");

                this.start.addActionListener(this);
                this.stop.addActionListener(this);
                this.toggle.addActionListener(this);

                this.setLayout(new FlowLayout());
                this.add(this.count1Field);
                this.add(this.count2Field);
                this.add(this.start);
                this.add(this.stop);
                this.add(this.toggle);

                this.pack();
                this.setVisible(true);
        }

        public void actionPerformed(ActionEvent e) {
                if (e.getSource().equals(start)){
                        this.count1.start();
                        this.count2.start();
                }
                else if (e.getSource().equals(stop)){
                        this.count1.stopCount();
                        this.count2.stopCount();
                }
                else if (e.getSource().equals(toggle)){
                        this.count1.toggleCount();
                        this.count2.toggleCount();
                }
        }

        public static void main(String[] args) {
                new CounterApplication2();
        }
}
```

Synchronisation of Threads

Introduction

It can be difficult to control threads when they need to share data between the threads and a common object. It is difficult to predict when data will be passed, often being slightly different each time the application runs. In certain situations it is vital that we **synchronize** threads to prevent this unpredictable behaviour and incorrect results.

A Good Bad Example

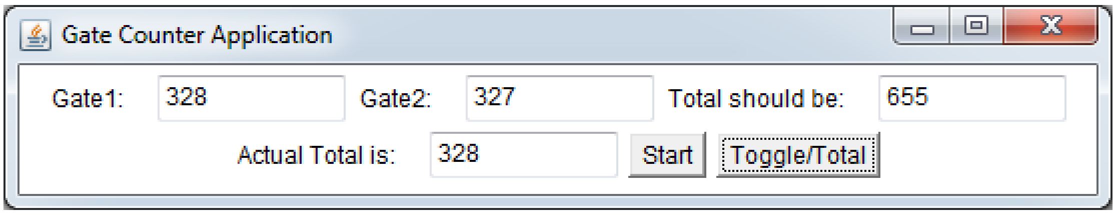
Suppose we have two gates into a stadium and that each time that a person goes through a gate a message is sent to a central counter to calculate the total number of people in the stadium. It would be vital for selling admittance to ensure that the total number of people in the stadium is less than the total capacity of the stadium.

Each gate in this example is operating independently and so resembles a thread. They share the central counter, which in this case resembles a suitable object.

Figure 8.6, "A Non-Synchronized Gate Counter Example" shows my implementation of this example. The first two TextField components are the independent gate counters (counting the number of people arriving at the stadium). The third TextFieldobject is the sum of the individual gates, so this is what the total **should be**. The "Actual Total" TextField is what our shared object (central counter) believes the total to be. As you can see there is a significant difference between what the total is, and what it should be. Why is this?

Well the reason is that the StadiumDetails "central counter" is not synchronized. When the first gate counter calls the spectatorEntered() method, it is entirely possible that the thread manager decides that that thread has had enough CPU cycles, and passes control to the second gate counter thread. You will notice that the first thread would have stored the current total in tempInt before it was stopped by the CPU, but the second thread would have updated the numberSpectators state while this thread was paused. When control is given back to the first thread, it would continue with the spectatorEntered() method, but it would be working with the old total, as stored in tempInt. I have purposely made it more difficult for the "central counter" to work correctly, by adding a sleep(5) call to force a delay of 5ms in each thread's counting cycle. See Figure 8.7, "A Non-Synchronized Gate Counter Example (Program Flow)" to see the program flow of the two threads. Remember that the two threads are sharing the **same StadiumDetails object**.

Figure 8.6. A Non-Synchronized Gate Counter Example


Gate Counter Application

Gate1: 328 Gate2: 327 Total should be: 655

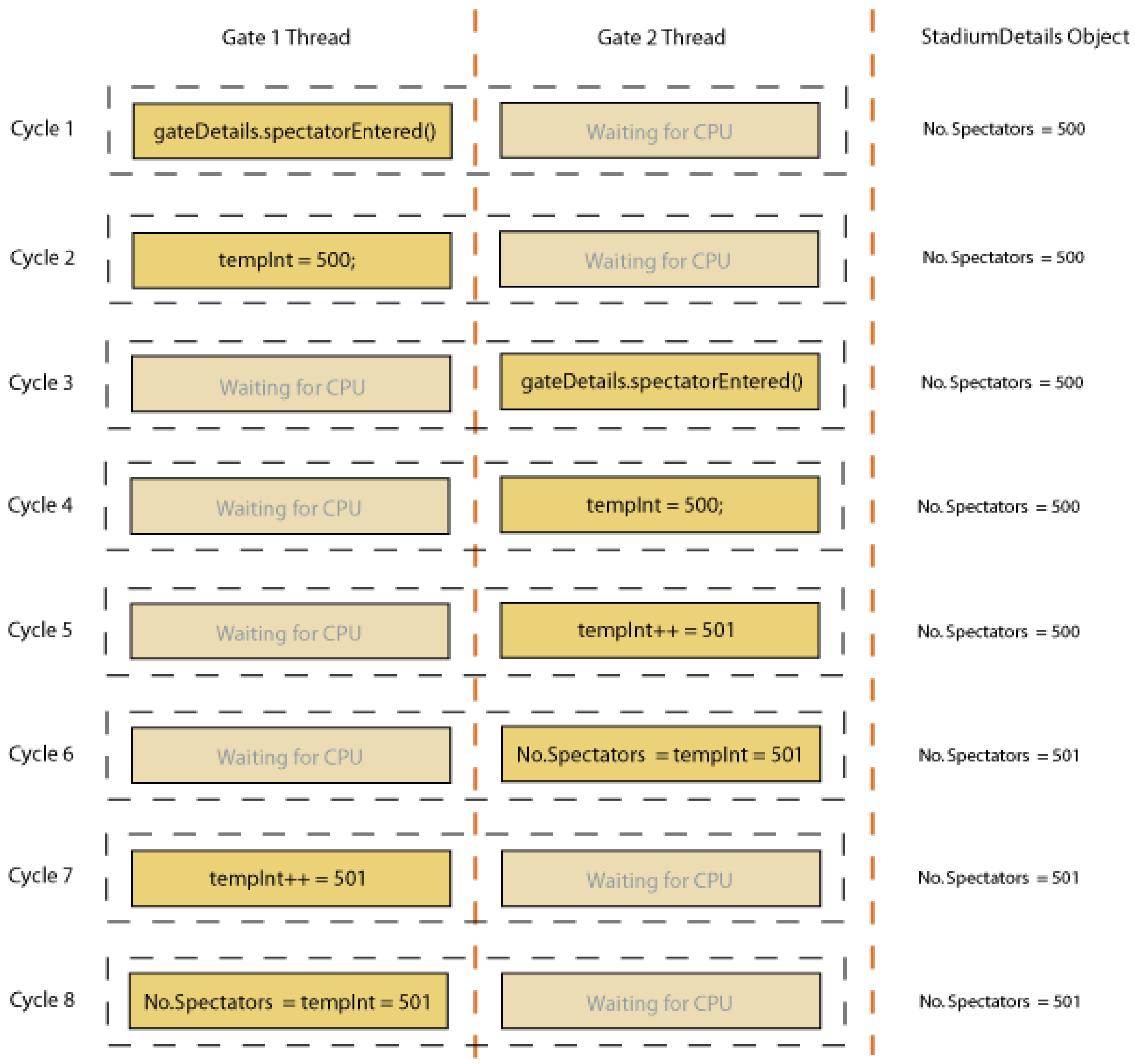
Actual Total is: 328 Start Toggle/Total

The entire code for this non-synchronized example as shown in Figure 8.6, "A Non-Synchronized Gate Counter Example" is in GateCounterApp.java The code segment for this example is:

GateCounterApp.java

```
class StadiumDetails
{
        private int numberSpectators;

        public void spectatorEntered() {
                int tempInt = this.numberSpectators;
                try {
                        Thread.sleep(5);  //added to cause problems.
                }
                catch (InterruptedException e) {
                        System.out.println(e.toString());
                }
                tempInt++;
                this.numberSpectators = tempInt;
        }

        public int getTotalSpectators() { return numberSpectators; }
}
```

Figure 8.7. A Non-Synchronized Gate Counter Example (Program Flow)

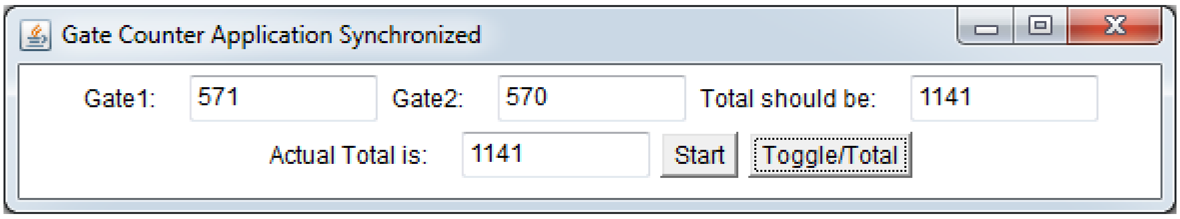


In this example of the problem of non-synchronized code I have made the problem worse, by inserting a quite unnecessary delay of 5ms. I did this to encourage the thread manager to change from execution of thread 1 thread 2 and vice-versa. If this delay was not there the problem would not have been so pronounced, instead of almost 50% of the spectators not being counted, maybe 1 in 1000 would not be counted, depending on your CPU conditions, the number of gates etc. The point is that it is unpredictable and should be fixed.

So how do we fix it? Well the answer is straightforward enough, we use the synchronized keyword, but the implementation is not quite as easy - when do we use it?

The synchronized keyword can be used to group a set of instructions that should not be interrupted by the thread manager, in other words a synchronized block of code should run to completion. In this example we can fix the StadiumDetails class to work correctly by adding the synchronized keyword to the spectatorEntered() method and to the getTotalSpectators() (for safety). Figure 8.8, “A Synchronized Gate Counter Example” shows a screen-capture of the application running correctly and Figure 8.9, “A Synchronized Gate Counter Example (Program Flow)” displays the program flow in the situation where the code has been modified. Note that I have still left in the delay to prove that it works correctly.

Figure 8.8. A Synchronized Gate Counter Example



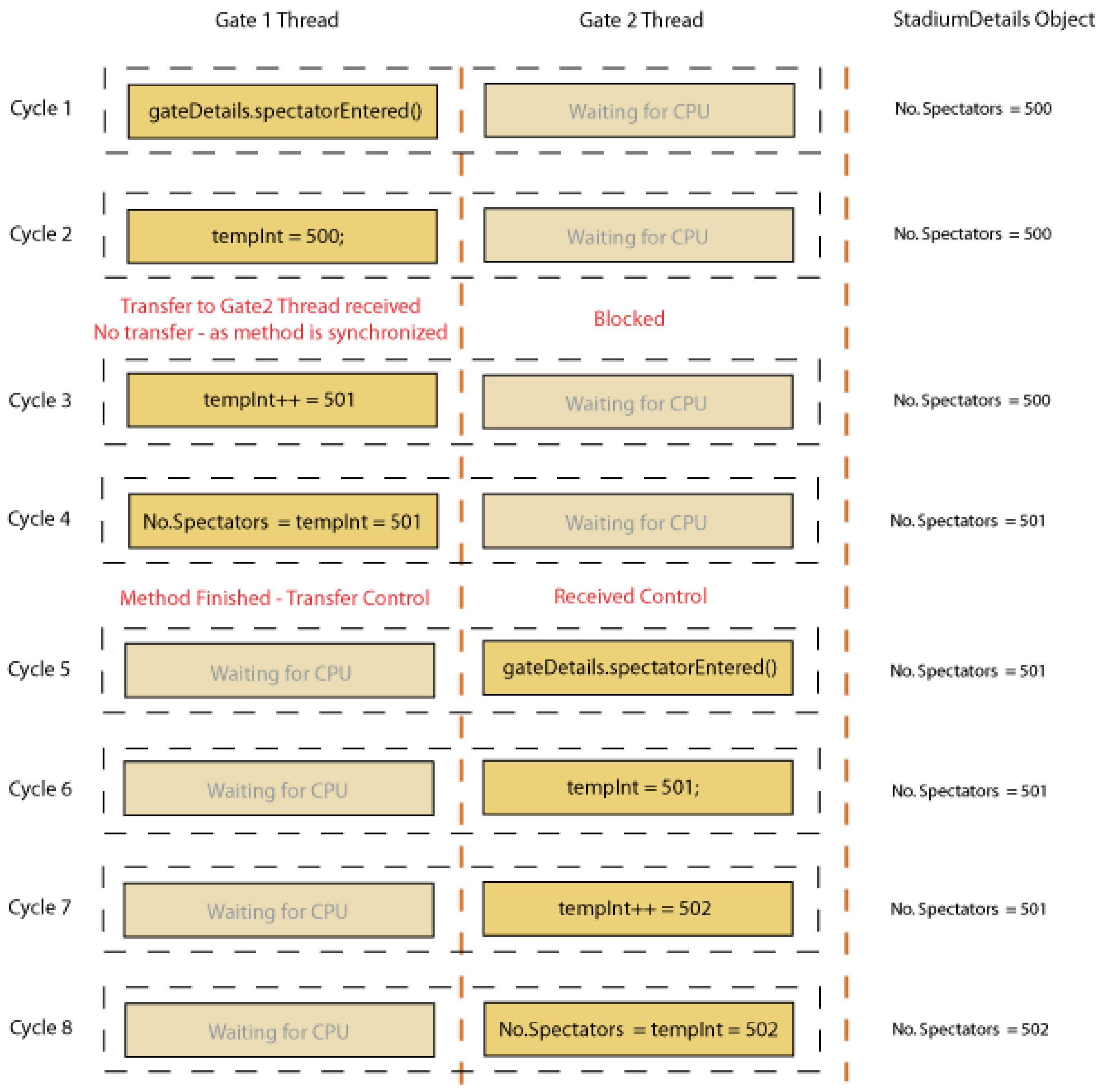
The entire code for this synchronized example as shown in Figure 8.8, "A Synchronized Gate Counter Example" is in GateCounterAppSync.java The fixed code in this example is:

GateCounterAppSync.java

```java
class StadiumDetailsSync
{
	private int numberSpectators;

	public synchronized void spectatorEntered() {
		int tempInt = this.numberSpectators;
		try {
			Thread.sleep(5);  //added to cause problems.
		}
		catch (InterruptedException e) {
			System.out.println(e.toString());
		}
		tempInt++;
		this.numberSpectators = tempInt;
	}

	public synchronized int getTotalSpectators() { return numberSpectators; }
}


```

As you can see in Figure 8.9, "A Synchronized Gate Counter Example (Program Flow)" the first thread executes as it has previously until it receives a request from the thread manager to transfer control to the next thread - but since the spectatorEntered() has been tagged with the synchronized keyword then this method must run to completion. So the second thread must wait until this method is finished before it can be loaded into the CPU and executed. Because of this, the total number of spectators is incremented correctly to 501 before the second thread begins. This means that the second thread starts counting from 501 and correctly increments the total to 502.

Figure 8.9. A Synchronized Gate Counter Example (Program Flow)



Adding Synchronisation to Code

We add synchronization to our code by either modifying the methods that we use to share this data like:

```java
public synchronized void theSynchronizedMethod()
{

}
```

Or we could select a block of code to synchronize and use:

```
synchronized(anObject)
{

}
```

This works like a lock on objects. When two threads execute code on the same object, only one of them acquires the lock and proceeds. The second thread waits until the lock is released on the object. This allows the first thread to operate on the object, without any interruption by the second thread.

Again, synchronization is based on objects:

- Two threads call synchronized methods on different objects, they proceed concurrently.
- Two threads call different synchronized methods on the same object, they are synchronized.
- Two threads call synchronized and non-synchronized methods on the same object, they proceed concurrently.

Static methods are synchronized per class. The standard classes are multithread safe.

It may seem that an obvious solution would be to synchronize everything!! However this is not that good an idea as when we write an application, we wish to make it:

- **Safe** - We get the correct results.
- **Lively** - It performs efficiently, using threads to achieve this liveliness.

These are conflicting goals, as too much synchronization causes the program to execute sequentially, but synchronization is required for safety when sharing objects. Always remove synchronization if you know it is safe, but if you are not sure then synchronize.

A Multi-Threaded TCP Server

Introduction

In the section called "The TCP Client/Server" we developed a full client/server application. The server we developed could only handle one connection at a time. It was structured to create a `ConnectionHandler` object for each connection that was made to the server, however, since the DateServer waits for the ConnectionHandler object to deal with the DateClient then cannot handle any further connections at the same time. You can see this if you run the server and then connect to it with two clients at the same time - the server will refuse the connection as it is busy serving the current client on port 5050 and has not started listening again.

However, the Server has been structured correctly to make it easily threaded by keeping the connection handler separate from the Server. All that has to be done is to make this connection handler (the HandleConnection class) threaded. Once it is threaded the server can create a separate thread for every client that connects, where the thread deals completely with that client. When the client disconnects this thread can finish.

To make the server threaded we can take the ConnectionHandler class from the section called “The TCP Client/Server” and modify it, by making it inherit from the Thread class and overriding the run() method of the Thread class.

The DateServer has to be modified slightly to create an object of our new ThreadedConnectionHandler class and to call the start(), method rather than calling the thread's run() method directly.

If we do this the ThreadedServer code should look like:

```java
...

public class ThreadedServer
{
    private static int portNumber = 5050;

    public static void main(String args[]) {
      ...

        // Server is now listening for connections or would not get to this point
        while (listening) // almost infinite loop - loop once for each client request
        {
            Socket clientSocket = null;
            try{
                System.out.println("**. Listening for a connection...");
                clientSocket = serverSocket.accept();
                System.out.println("00. <- Accepted socket connection
                                  from a client: ");
                System.out.println("    <- with address: " +
                                   clientSocket.getInetAddress().toString());
                System.out.println("    <- and port number: " +
                                   clientSocket.getPort());
            }
            catch (IOException e){
                System.out.println("XX. Accept failed: " + portNumber + e);
                listening = false;   // end loop - stop listening for client requests
            }

            ThreadedConnectionHandler con =
                  new ThreadedConnectionHandler(clientSocket);
            con.start();
            ...
        }
  ...
```

And the ThreadedConnectionHandler code should look like:

```java
/* The Connection Handler Class - Written by Derek Molloy for the EE402 Module
* See: ee402.eeng.dcu.ie
*/

package ee402;

import java.net.*;
import java.io.*;

public class ThreadedConnectionHandler extends Thread
{
    private Socket clientSocket = null;              // Client socket object
    private ObjectInputStream is = null;             // Input stream
    private ObjectOutputStream os = null;            // Output stream
    private DateTimeService theDateService;

        // The constructor for the connection handler
    public ThreadedConnectionHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
        //Set up a service object to get the current date and time
        theDateService = new DateTimeService();
    }

    // Will eventually be the thread execution method - can't pass the exception back
    public void run() {
         try {
            this.is = new ObjectInputStream(clientSocket.getInputStream());
            this.os = new ObjectOutputStream(clientSocket.getOutputStream());
            while (this.readCommand()) {}
         }
         catch (IOException e)
         {
            System.out.println("XX. There was a problem with
                      the Input/Output Communication:");
            e.printStackTrace();
         }
    }

...
```

To make it easier for you to see this update, I have placed all the source files here:

- ThreadedServer.java - The **new** Threaded Server class
- ThreadedConnectionHandler.java - The **new** Handle Connection Thread class
- Client.java - The same client, no changes are necessary.
- DateTimeService.java - The same Date/Time Service class.

Chapter 9 - Swing Lightweight Components

Introduction

In the last few chapters we built user interfaces using the AWT (Abstract Windowing Toolkit) classes. We will now examine the use of advanced components, building a user interface using the JFC (Java Foundation Classes) Swing API.

The Java Foundation Classes (JFCs) are named like the Microsoft Foundation Classes (MFCs) and are equivalent in nature. The JFC is a group of packages that provide GUI classes for Java applets/applications. The AWT is the foundation of JFC, but JFC includes APIs such as Swing, 2D API (for 2D graphics) and the Accessibility API (for ease of access for people with disabilities).

Swing is an ever expanding library of components used to build Graphical User Interfaces. The AWT components we discussed previously are heavyweight components, relying on the operating system to render them. Swing components are lightweight components, completely independent of any operating system. Heavyweight and lightweight components can be mixed, however I would not advise it as it sometimes leads to difficulties in repainting. We will still use the AWT layout managers in creating our Swing based applications. Traditionally the Swing API was downloaded separately to the core API (in Java 1.x) - now it is a part of the core API (as of Java 2).

Some of the features of Swing Components are:

- **They are Lightweight** - Most Swing components are written in Java and so do not depend on the host operating system to draw them. Because of this we can develop complex visual components.
- All Swing components support the **Accessibility API**. In addition we can add tool-tips to a Button to provide further description of its function.
- Since the components are lightweight we can change the **Look-And-Feel** of an application - even during run-time.
- All Swing components can have their own borders, allowing us to design **advanced layouts**.

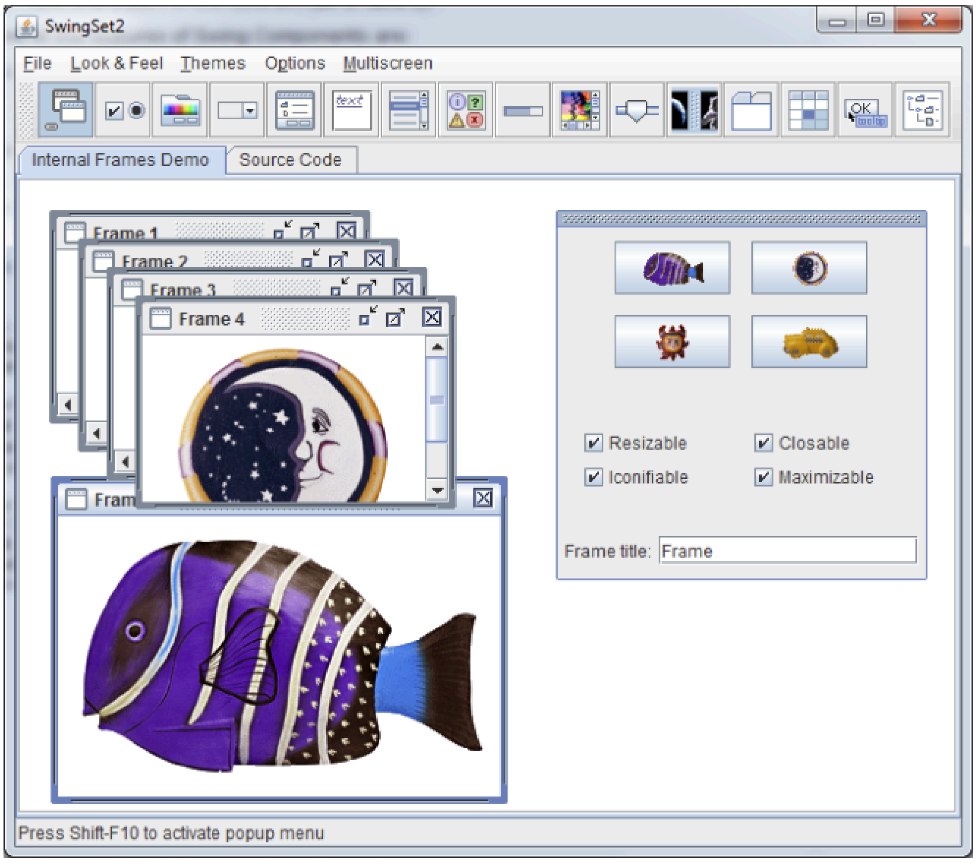
We will use Swing to develop applications. The JDK (Java JDK 7+) comes packaged with various demonstration programs. If you look in the directory C:\Java\jdk1.7.0\demo\jfc you will see a collection of Swing Applications.

The first one to run is in SwingSet2, where it is stored as SwingSet2.jar i.e. a Java Archive that contains all the .class files for the example. You can execute this example by typing java -jar SwingSet2.jar in this directory. If you have your windows environment set up correctly

You will see an application as in Figure 9.1, “The Swing Set 2 Example”. You can see from this figure that there is a huge selection of Swing components that you can use.

Note that you can select the "source code" tab to see how the code in the window was written. This can be very useful if you see a component that you would like to use in the same format.

Figure 9.1. The Swing Set 2 Example

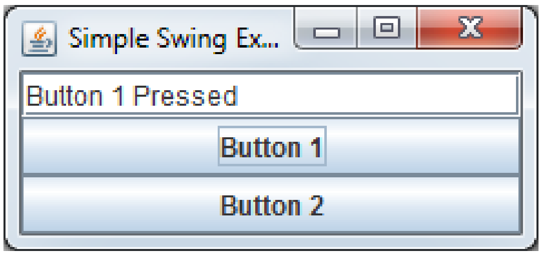


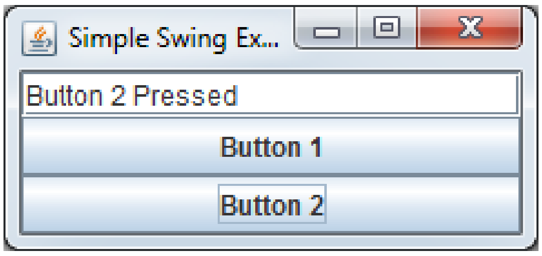
Developing with Swing

A Basic Swing Example

To write a simple Swing application we do not have to make too many changes to a standard AWT application. Figure 9.2, "A Simple Swing Application" shows a simple Swing button application that should provide you with a template for writing your own applications.

Figure 9.2. A Simple Swing Application





The source code for this application is below: SimpleSwingApplication.java

```java
/** Java Swing Examples - Written by Derek Molloy, Dublin City University,
 */

package ee402;

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

@SuppressWarnings("serial")
public class SimpleSwingApplication extends JFrame implements ActionListener {

       private JButton button1, button2;
       private JTextField status;

       public SimpleSwingApplication() {
         // call the parent class constructor - sets the frame title
         super("Simple Swing Example");

         this.status = new JTextField(20);
         this.button1 = new JButton("Button 1");
         this.button2 = new JButton("Button 2");

         this.button1.addActionListener(this);
         this.button2.addActionListener(this);

         // default layout is border layout for Frame/JFrame
         // note: since Java 1.5 you no longer require getContentPane() below
         this.getContentPane().add("North",this.status);
         this.getContentPane().add("Center",this.button1);
         this.getContentPane().add("South",this.button2);

         this.pack();
         this.setVisible(true);
```

```
        }

        public void actionPerformed(ActionEvent e) {
          if (e.getActionCommand().equals("Button 1")) {
            this.status.setText("Button 1 Pressed");
          }
          else if (e.getActionCommand().equals("Button 2")) {
            status.setText("Button 2 Pressed");
          }
        }

        public static void main(String args[]) {
           new SimpleSwingApplication();
        }
}
```

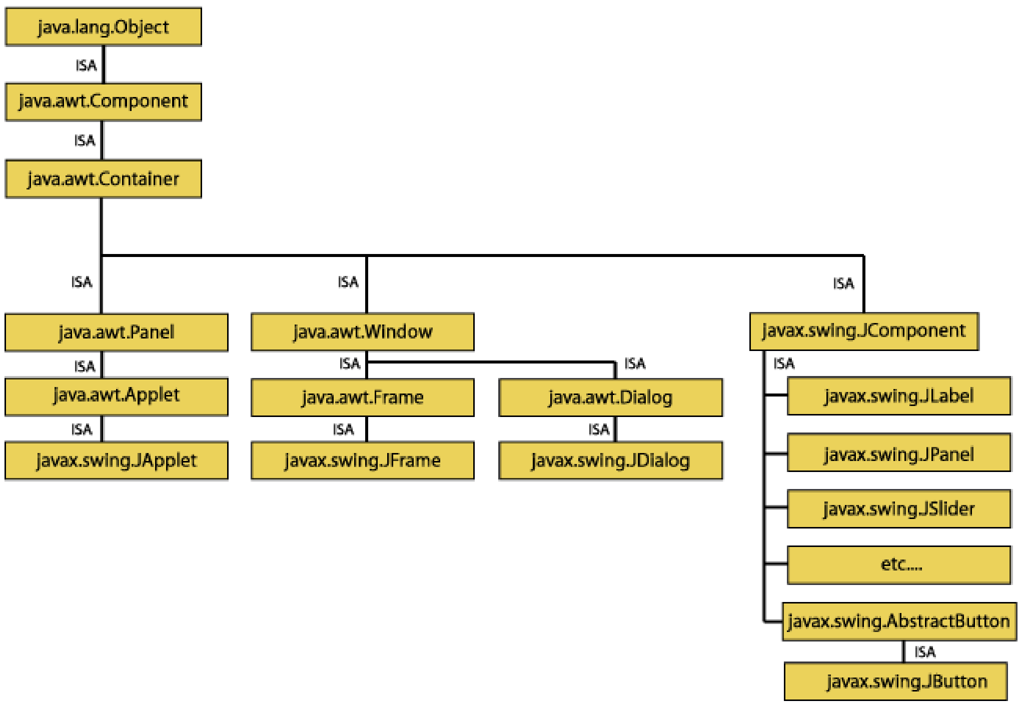
this.pack() sets the JFrame object to its components preferred sizes. this.setVisible(true) makes the JFrame object visible on the screen. You usually make this call last as all layout has been carried out and the frame will not be changing sizes and moving after it is displayed. You can also have refresh problems if you do not do this last.

The Swing equivalent components for the AWT components that we have discussed previously are:

- JButton - as discussed replaces the AWT

Button has a few changes. For one as in Figure 9.3, “The Swing Component Hierarchy” the parent class is AbstractButton which defines a common behaviour for all buttons, for example there is a two-state

Figure 9.3 The Swing Component Hierarchy



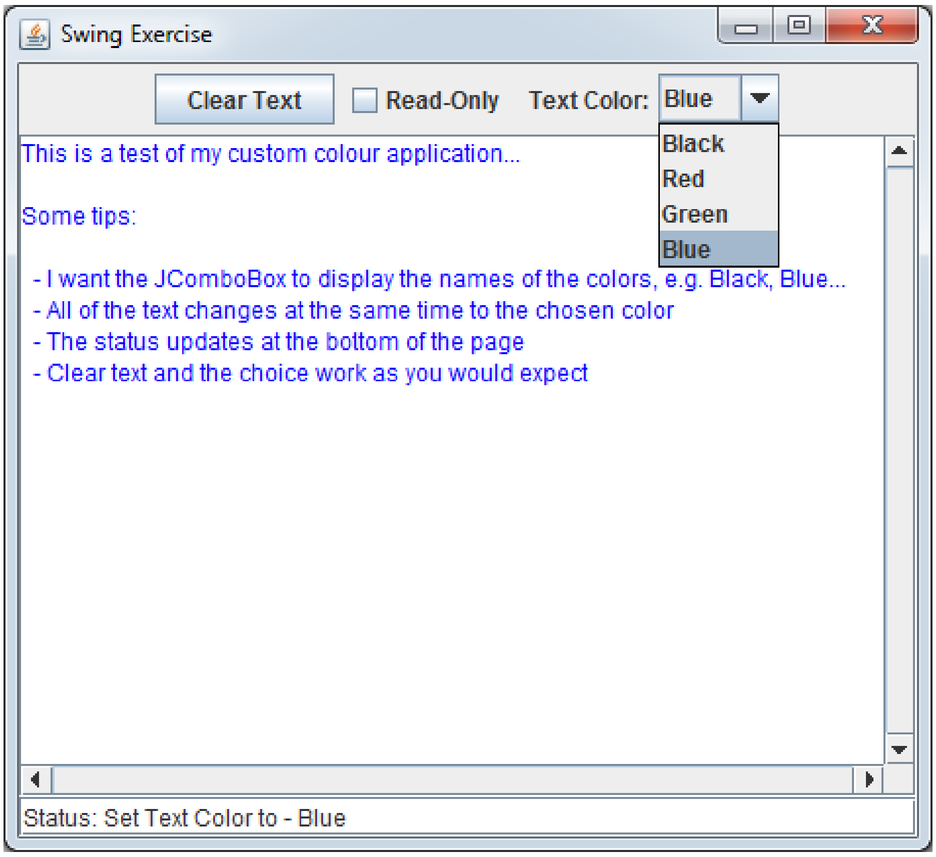
- **JToggleButton** class also inheriting from AbstractButton. Button objects can also have an image icon added to the button area.
- **JTextField** - replaces the TextField class and remains essentially the same. There is a slight difference in the event structure and we have the added benefit of some derived sub-classes, such as JPasswordField which allows a password to be entered using a character mask, and JFormattedTextField to allow for the entry of formatted text, such as a date of birth.
- **JTextArea** - replaces the TextArea class and is slightly different. For one thing, the JTextArea does not handle scrolling of the text to allow you to have more control over its functionality. To add a scrolling capability to a JTextAreacomponent you can add it to a JScrollPane component. By default, the line wrapping property is set to false. You can enable line wrapping by using the

Exercise - Swing Components

Task: Write the application as shown in Figure 9.6, “A Swing Exercise for You!”, where:

- The JTextArea component in the centre should have the ability to scroll (with Scrollbars always visible).
- See Figure 9.6, “A Swing Exercise for You!”
- See the JTextArea class for more information.

Figure 9.6. A Swing Exercise for You!



```
This is a test of my custom colour application...

Some tips:

 - I want the JComboBox to display the names of the colors, e.g. Black, Blue...
 - All of the text changes at the same time to the chosen color
 - The status updates at the bottom of the page
 - Clear text and the choice work as you would expect
```

Status: Set Text Color to - Blue

Hint: Remember to use the API documentation for method calls etc. You will probably have to use the setForeground(Color c) method to change the text colour of the TextArea, unless you can work out a better way.

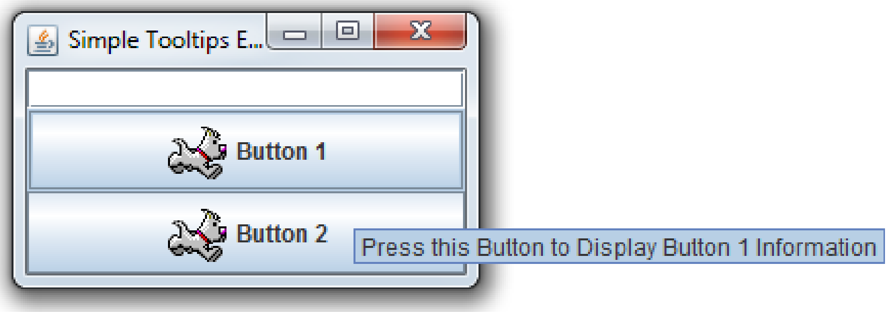
Solution: My solution is here, but please don't look at it until you have had a good attempt at it - SwingExercise.java

Other Properties added to Swing Components

ToolTips can be added to components to allow a further description of the functionality of the component. You can use the setToolTipText() method of the JComponent class. When you float over the JButton object the ToolTip will appear as in Figure 9.7, "An Example of Swing ToolTips". You can see the source code in SwingTooltips.java

```
 this.button1 = new JButton("Button 1", icon);
this.button1.setToolTipText("Press this Button to Display Button 1 Information");
this.button2 = new JButton("Button 2", icon);
this.button2.setToolTipText("Press this Button to Display Button 2 Information");
```

Figure 9.7. An Example of Swing ToolTips

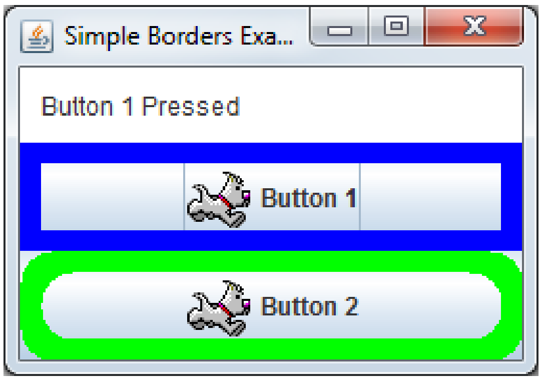


Swing also provides borders for JComponent components. Border is an interface that has many default implementations that you can use, or even create your own. Figure 9.8, "An Example of Swing Borders" shows three different border types that you can use. The JTextField component is surrounded by a EmptyBorder with a colour of white, the first JButton component is surrounded by a blue MatteBorder and the last JButton component is surrounded by a LineBorder with a line width of 10 pixels, a color of green and rounded corners set to true. You must import the MatteBorder and the last JButton component is surrounded by a LineBorder with a line width of 10 pixels, a color of green and rounded corners set to true. You must import the javax.swing.border.* package to use these pre-defined borders. The code used for the example shown in Figure 9.8, "An Example of Swing Borders" is as below

```
this.status = new JTextField(20);
this.status.setBorder(new EmptyBorder(10,10,10,10));
this.button1 = new JButton("Button 1", icon);
this.button1.setBorder(new MatteBorder(10,10,10,10,Color.blue));
this.button1.setToolTipText("Press this Button to Display Button 1 Information");
```

```
  this.button2 = new JButton("Button 2", icon);
  this.button2.setBorder(new LineBorder(Color.green, 10, true));
  this.button2.setToolTipText("Press this Button to Display Button 2 Information");
```

Figure 9.8. An Example of Swing Borders



The code for this example can be found here: SwingBorders.java

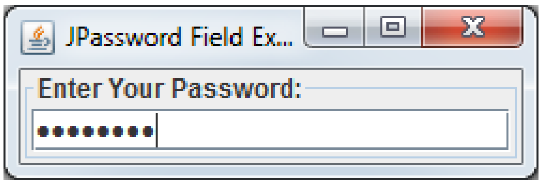
Other Useful Swing Components

General Components

The JPasswordField Component

The JPasswordField component is a child class of JTextField and so has similar functionality, except that it masks the entered text with a character, the default mask being '*'. Figure 9.9, "A Password Field Example" shows this example running. When the text is entered (without being visible) the user can press the enter key and an ActionEvent object will be generated. This ActionEvent can be passed to the actionPerformed() method, where it can be identified and then the password field can be interrogated using getPassword() that returns an array of char. The use of the char array is for security reasons and each element of the array should be set to blank after the password has been validated. You can use the setEchoChar('X') to change the echo character to whatever is required.

Figure 9.9. A Password Field Example



The source code for this example is as below and as in JPasswordFieldExample.java

```
 package ee402;
 import javax.swing.*;
```

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

@SuppressWarnings("serial")
public class JPasswordFieldExample extends JFrame implements ActionListener {
        JLabel l1 = new JLabel("This is a Test Swing Application");
        JButton b1, b2;
        JPasswordField pwd;

        public JPasswordFieldExample()
        {
                super("JPassword Field Example");
                JPanel p = new JPanel();
                p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
                p.setBorder(new TitledBorder("Enter Your Password:"));
                pwd = new JPasswordField(20);
                pwd.addActionListener(this);

                p.add(pwd);
                this.getContentPane().add(p);
                this.pack();        // set the size automatically
                this.setVisible(true);
        }

        public void actionPerformed(ActionEvent e)
        {
                if (e.getSource().equals(pwd))
                {
                        char[] thePassword = pwd.getPassword();
                        String s = new String(thePassword);
                        System.out.println("Password is " + s);
                }
        }

        public static void main(String[] args)
        {
                new JPasswordFieldExample();
        }
}
```

The JSlider Component

The JSlider component is a replacement for the Scrollbar AWT component. It is mentioned here, because there is quite a range of new functionality available with this component. For example if you look at the second JSlider object from the top (the middle one) in Figure 9.10, “A JSlider Example”, you will see that it has a title, tick marks for intervals. The code for this component is shown below:

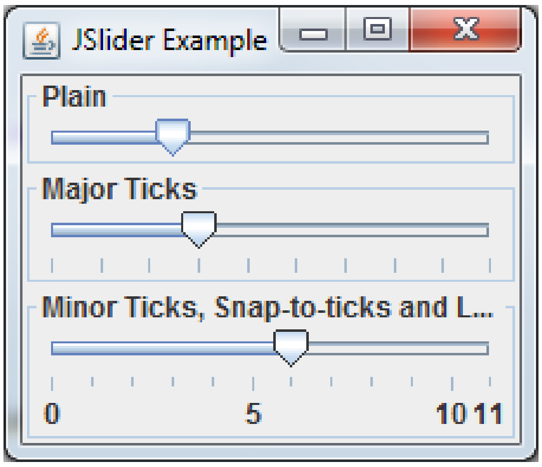
```
// Slider 2
p = new JPanel();
p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
p.setBorder(new TitledBorder("Major Ticks"));
s = new JSlider(100, 1000, 400);
s.setPaintTicks(true);
```

```
   s.setMajorTickSpacing(100);
   s.getAccessibleContext().setAccessibleName("Major Ticks");
   s.getAccessibleContext().setAccessibleDescription("A slider showing major tick
 marks");
   //s.addChangeListener(this);
   p.add(s);
   hp.add(p);
```

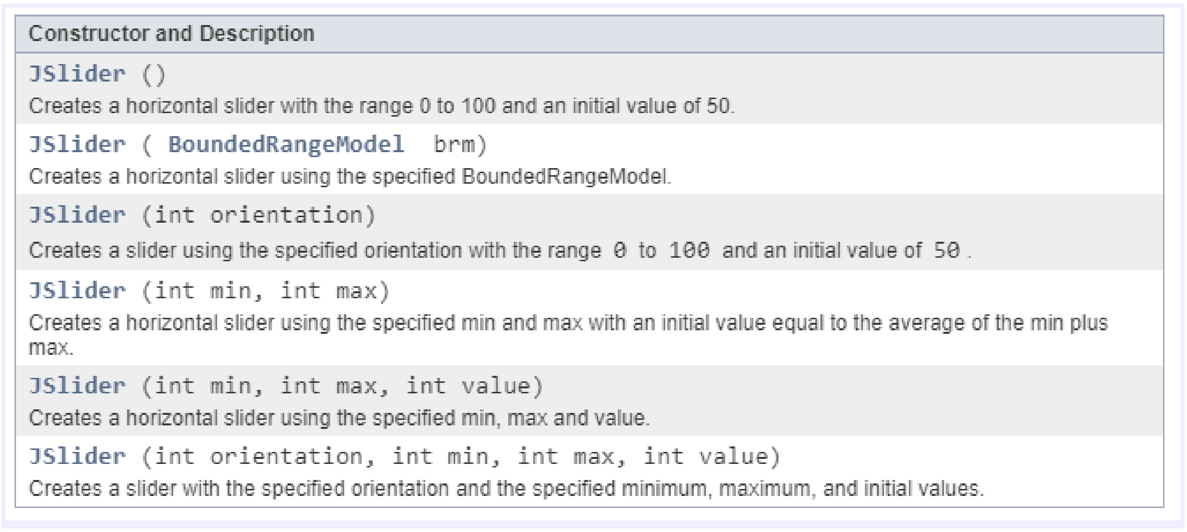
A JPanel object is created for each JSlider object. The layout of this JPanel object is set to BoxLayout, which allows multiple components to be laid out either vertically or horizontally, without wrapping. In this case the components are laid out in BoxLayout.Y_AXIS allowing the components to be laid out vertically, and these components will remain vertical, even when the JFrame is resized. For a BoxLayout you also have to pass the container that you wish to lay out - in this case p. TitleBorder is another border that you can use, that places a title around the border frame - in this case "Major Ticks". The JSlider object is then created with a minimum of 100, a maximum of 1000, and an initial value of 400. The setPaintTicks() enables/disables the tick lines below the JSlider component. The setMajorTickSpacing(100) method call sets spacing to 100, so we will have 10 ticks in this case.

The next two lines demonstrate the Accessibility API (part of the JFCs) that sets the Accessible name of this component to "Major Ticks" and the description to "A slider showing major tick marks", so that if the user has a visual disability that speech synthesis software would be capable of identifying the reason for the component to allow that person to use it.

Figure 9.10. A JSlider Example



Other constructors that can be used with JSlider are:



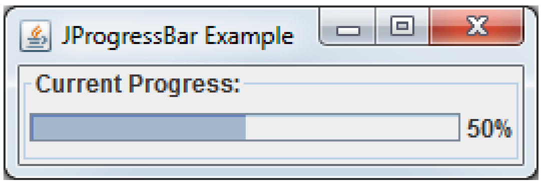
Constructor and Description
`JSlider ()` Creates a horizontal slider with the range 0 to 100 and an initial value of 50.
`JSlider ( BoundedRangeModel  brm)` Creates a horizontal slider using the specified BoundedRangeModel.
`JSlider (int orientation)` Creates a slider using the specified orientation with the range `0` to `100` and an initial value of `50` .
`JSlider (int min, int max)` Creates a horizontal slider using the specified min and max with an initial value equal to the average of the min plus max.
`JSlider (int min, int max, int value)` Creates a horizontal slider using the specified min, max and value.
`JSlider (int orientation, int min, int max, int value)` Creates a slider with the specified orientation and the specified minimum, maximum, and initial values.

The Full code for this example can be seen in JSliderExample.java

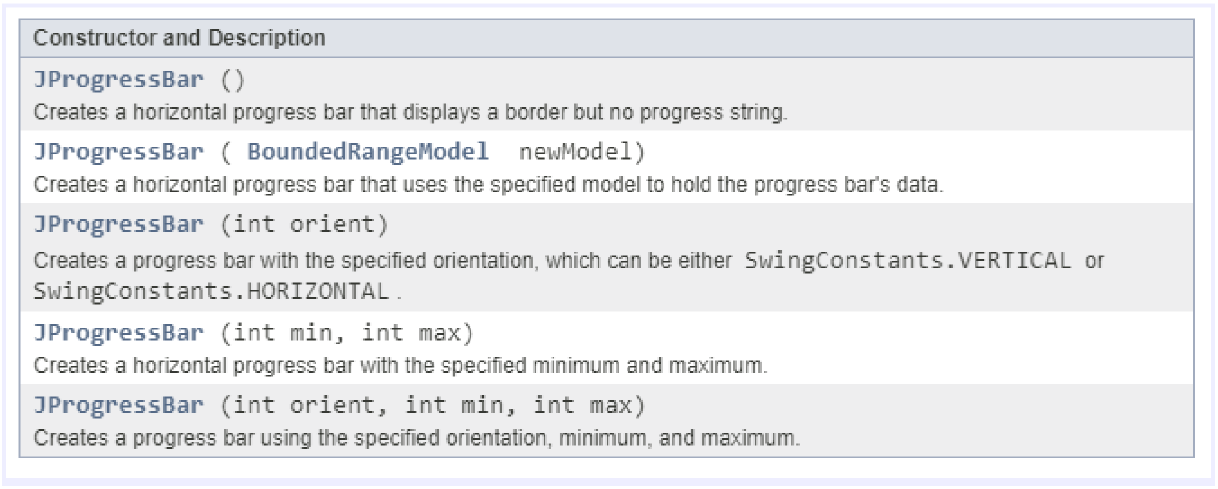
The JProgressBar Component

A JProgressBar component displays an Integer value within a bounded interval. A progress bar typically communicates the progress of an event by displaying its percentage of completion and possibly also provides a textual description. See Figure 9.11, “A JProgressBar Example” for a basic example.

Figure 9.11. A JProgressBar Example



The code for this is in JProgressBarExample.java There are several constructors for JProgressBar:



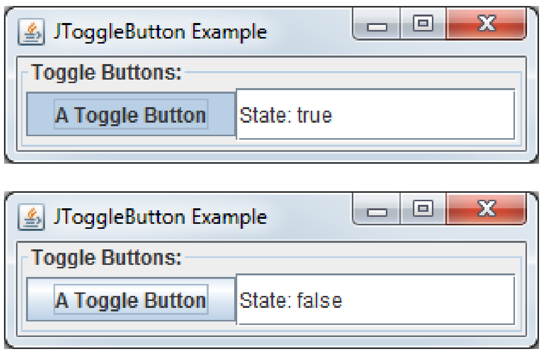
Constructor and Description
`JProgressBar ()` Creates a horizontal progress bar that displays a border but no progress string.
`JProgressBar ( BoundedRangeModel  newModel)` Creates a horizontal progress bar that uses the specified model to hold the progress bar's data.
`JProgressBar (int orient)` Creates a progress bar with the specified orientation, which can be either `SwingConstants.VERTICAL` or `SwingConstants.HORIZONTAL` .
`JProgressBar (int min, int max)` Creates a horizontal progress bar with the specified minimum and maximum.
`JProgressBar (int orient, int min, int max)` Creates a progress bar using the specified orientation, minimum, and maximum.

You can set the value of the progress bar by using the setValue(int) method and you can get the value by using the getValue() method.

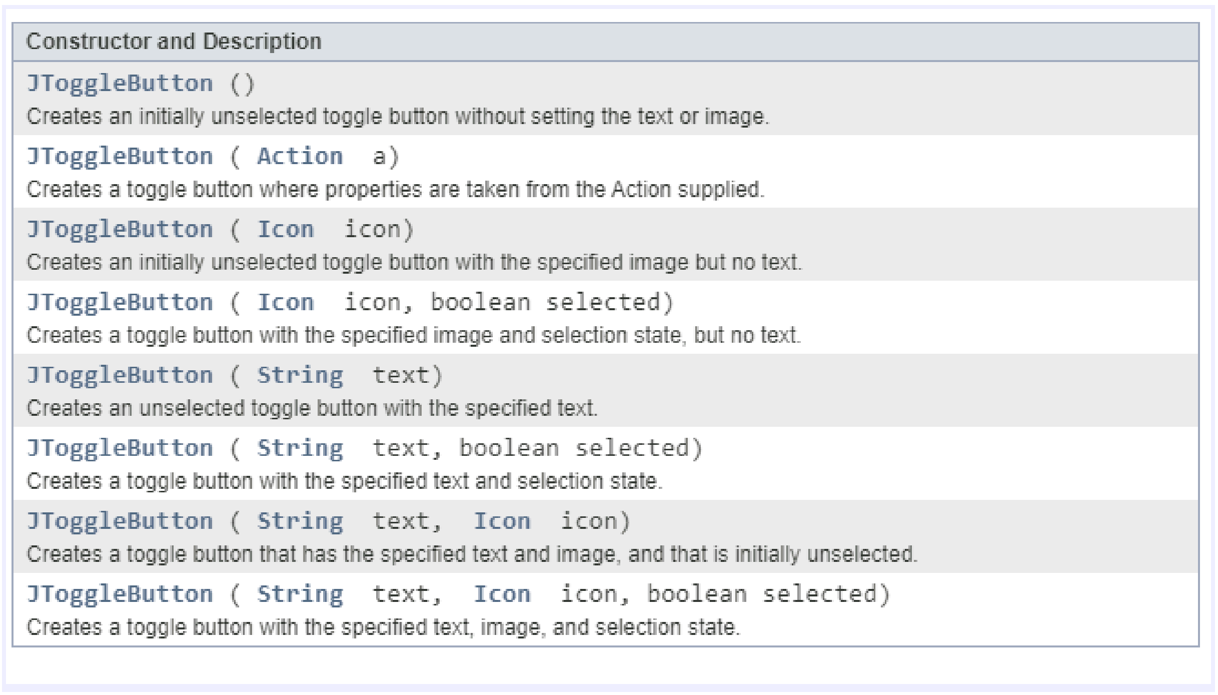
The JToggleButton Component

The JToggleButton component is very similar to a checkbox, in that it can be selected or not selected. Figure 9.12, "The JToggleButton component example (a) selected (b) not selected." shows a selected JToggleButton object and a non-selected component. It is very similar to the JButton class except that it can have two states.

Figure 9.12. The JToggleButton component example (a) selected (b) not selected.



The code for this example is in JToggleButtonExample.java There are several constructors for a JToggleButton:



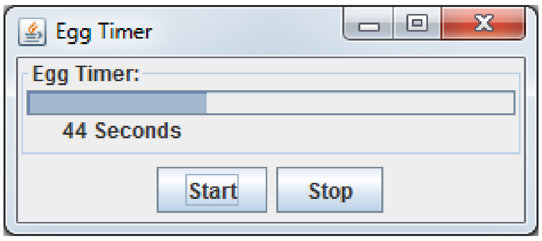
Constructor and Description
`JToggleButton ()` Creates an initially unselected toggle button without setting the text or image.
`JToggleButton ( Action a)` Creates a toggle button where properties are taken from the Action supplied.
`JToggleButton ( Icon icon)` Creates an initially unselected toggle button with the specified image but no text.
`JToggleButton ( Icon icon, boolean selected)` Creates a toggle button with the specified image and selection state, but no text.
`JToggleButton ( String text)` Creates an unselected toggle button with the specified text.
`JToggleButton ( String text, boolean selected)` Creates a toggle button with the specified text and selection state.
`JToggleButton ( String text, Icon icon)` Creates a toggle button that has the specified text and image, and that is initially unselected.
`JToggleButton ( String text, Icon icon, boolean selected)` Creates a toggle button with the specified text, image, and selection state.

If you wish to find the current state of the JToggleButton object using the isSelected() method that returns a boolean value.

Exercise. Write an Egg Timer

Task: Write an egg timer Swing application that counts to two minutes (well for runny eggs), updating the time as a progress bar and playing a sound when it is finished. You should be able to stop the timer at any stage using the "stop" button. My version is in Figure 9.13, "The Egg Timer Exercise". You can use this sound ready.wav if you wish.

Figure 9.13. The Egg Timer Exercise



Hints: Don't set the delay to 1 second until you have finished, otherwise you will have to wait a full two minutes every time you are debugging your application. To load a sound clip in an application you have to use something like:

```
URL clipLocation = this.getClass().getResource("ready.wav");
AudioClip theSound = Applet.newAudioClip(clipLocation);
```

and to play:

```
theSound.play();
```

And this involves importing the java.applet.* package. You may also (depending on the way you write your code) have to add a sleep call after you call the play() method of the AudioClip as the application/function may finish before the sound clip has played completely.

Solution: My solution is here - EggTimerApplication.java. Once again have a good attempt at the exercise before reading my solution (as yours may be better!).

Toolbars and Multiple Container Classes

Toolbars and Menus

Toolbars and Menus are containers for components, such as general actions related to an application. In most applications you will have a File menu that allows you to exit, or load a file or print your document etc.

The JToolBar Class

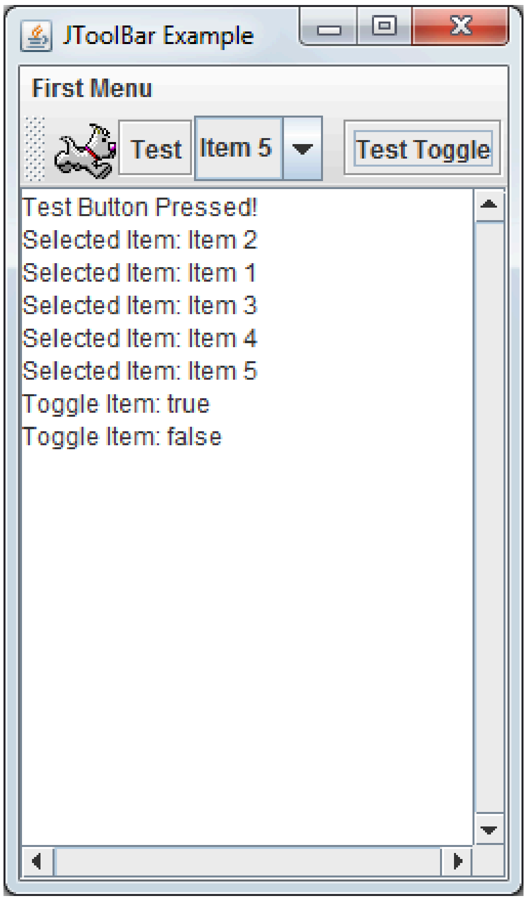
Figure 9.14, "The JToolBar Example Application. Toolbar (a) at top (b) at bottom." shows two views of the same application, where the toolbar is shown on top in (a) and at the bottom in (b). The toolbar has several components added to it. The first is a JLabel component with an ImageIcon, the second is a JButton component, the third is a JComboBox component with 5 items and finally a JToggleButton. The JToolBar object was created and the items added using the following code:

```
theToolbar = new JToolBar("My Toolbar");
theToolbar.setFloatable(true);
theTextArea = new JTextArea(20,20);JScrollPane p = new JScrollPane(this.theTextArea,
   JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
   JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
URL iconLocation = this.getClass().getResource("image.gif");
ImageIcon icon = new ImageIcon(iconLocation, "A running dog icon");
JLabel theLabel = new JLabel(icon
theToolbar.add(theLabel);
testButton = new JButton("Test");
testButton.addActionListener(this);
theToolbar.add(testButton);

String[] items = {"Item 1", "Item 2", "Item 3", "Item 4", "Item 5"};
comboBox = new JComboBox<String>(items);
comboBox.addActionListener(this);
theToolbar.add(comboBox);
theToolbar.addSeparator();

toggleButton = new JToggleButton("Test Toggle");
toggleButton.addActionListener(this);
theToolbar.add(toggleButton);
```

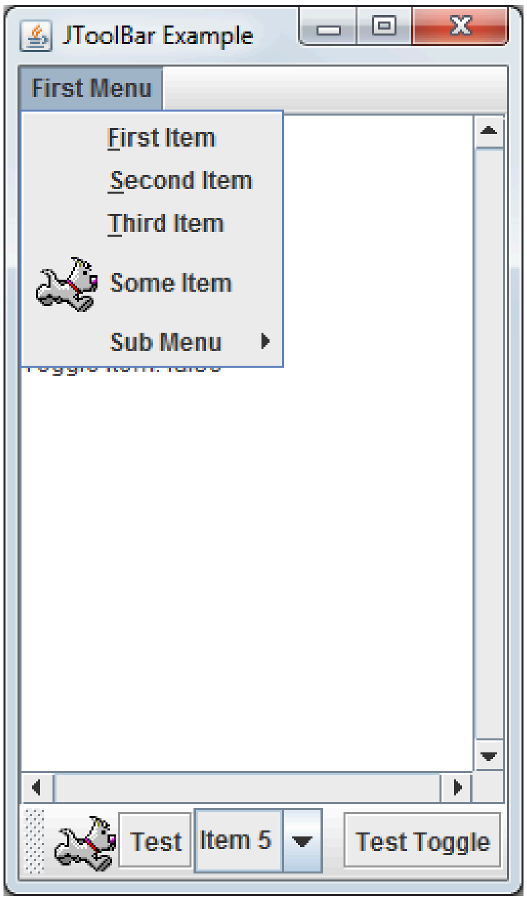
To allow the JToolBar object to be dragged from the top to the sides and to the bottom, the window container should have a BorderLayout and should not have other components added to the North, South, East or West.

In the code segment an ActionListener is added to the JButton object, to the JComboBox object and to the JToggleButton object. This class implements the ActionListener and so has an actionPerformed() method implemented. We will use the getSource() on the ActionEvent to identify the source object of the event. A space is added on the toolbar using the addSeperator() method. This method also allows you to specify the size of the gap you require. To add the toolbar to the window container simply use the add() method, in this case applied to the Container returned from the JFrame getContentPane() method.

Figure 9.14. The JToolBar Example Application. Toolbar (a) at top (b) at bottom.

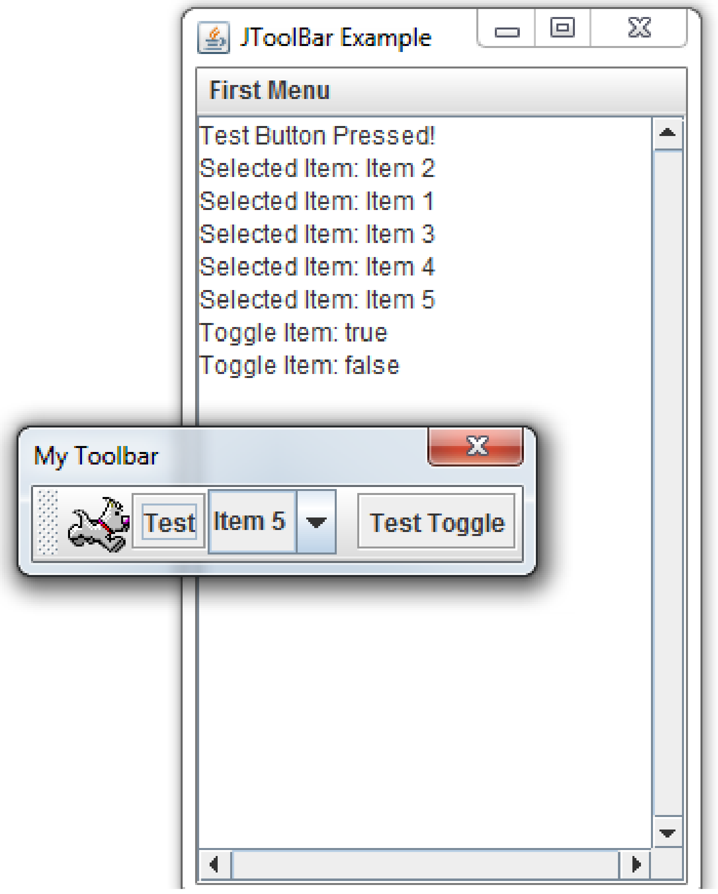


```
Test Button Pressed!
Selected Item: Item 2
Selected Item: Item 1
Selected Item: Item 3
Selected Item: Item 4
Selected Item: Item 5
Toggle Item: true
Toggle Item: false
```



If you allow the JToolBar to be floatable (using the JToolBar setFloatable(true) method then the menu bar can be dragged off the top of the application and can exist as its own window. See Figure 9.15, “The JToolBar Example with Floating Toolbar” to see this.

Figure 9.15. The JToolBar Example with Floating Toolbar



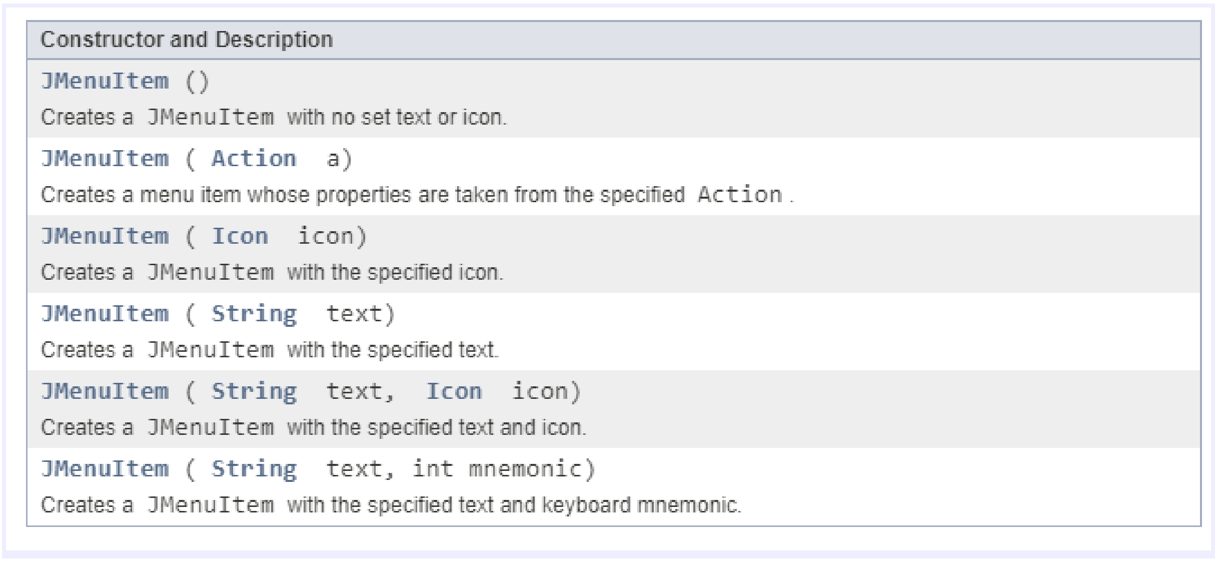
```
Test Button Pressed!
Selected Item: Item 2
Selected Item: Item 1
Selected Item: Item 3
Selected Item: Item 4
Selected Item: Item 5
Toggle Item: true
Toggle Item: false
```

The full code for this example is in JToolbarExample.java

The JMenuBar Class

The same application as shown above in the section called "The JToolBar Class" (and source code in JToolBarExample.java) also demonstrates the use of the JMenuBar class.

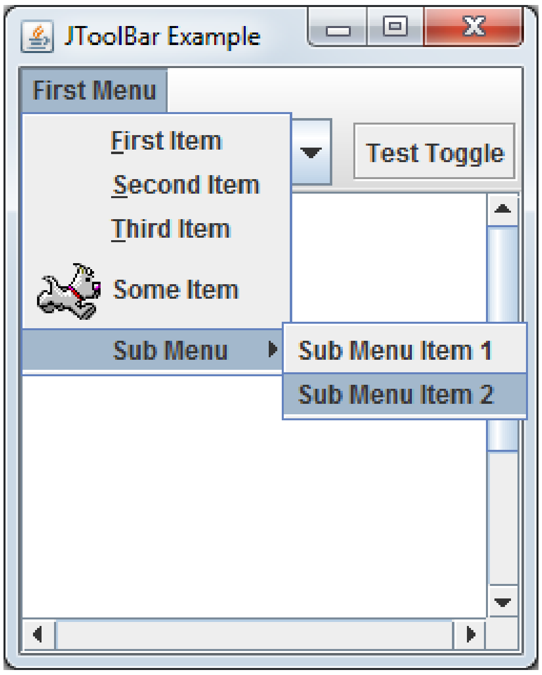
Figure 9.16, "The JMenuBar Example" shows the menu in action. There are four menu items and a sub-menu with two sub menu items. Notice that the first three items have short-cut keys (or accelerator keys). The constructors for JMenuItem are:



Constructor and Description
`JMenuItem ()` Creates a `JMenuItem` with no set text or icon.
`JMenuItem ( Action a)` Creates a menu item whose properties are taken from the specified `Action` .
`JMenuItem ( Icon icon)` Creates a `JMenuItem` with the specified icon.
`JMenuItem ( String text)` Creates a `JMenuItem` with the specified text.
`JMenuItem ( String text, Icon icon)` Creates a `JMenuItem` with the specified text and icon.
`JMenuItem ( String text, int mnemonic)` Creates a `JMenuItem` with the specified text and keyboard mnemonic.

Each of the JMenuItem objects has had an ActionListener applied, so when it is selected an ActionEvent object is generated and so our actionPerformed() method is called and in this case simply displays the menu item that was selected.

Figure 9.16. The JMenuBar Example



To add the MenuBar to a JFrame, use the JFrame class setJMenuBar() method (remember to put in the J as otherwise you would be using the AWT MenuBar and Frame methods). So you add the JMenuItem objects to the JMenu object and then you add the JMenu object to the JMenuBar object which is in turn added to the JFrame object. To create a sub-menu, as in Figure 9.16, "The JMenuBar Example" simply create a new JMenu object and add it to a JMenu object (i.e. there is no special sub menu class).

```
JMenuBar menuBar = new JMenuBar();
JMenu firstMenu = new JMenu("First Menu");
```

```
      JMenuItem item1 = new JMenuItem("First Item", KeyEvent.VK_F);
      JMenuItem item2 = new JMenuItem("Second Item", KeyEvent.VK_S);
      JMenuItem item3 = new JMenuItem("Third Item", KeyEvent.VK_T);
      JMenuItem item4 = new JMenuItem("Some Item", icon);
      JMenuItem sub1 = new JMenuItem("Sub Menu Item 1");
      JMenuItem sub2 = new JMenuItem("Sub Menu Item 2");
      JMenu aSubMenu = new JMenu("Sub Menu");
      aSubMenu.add(sub1);
      aSubMenu.add(sub2);
      item1.addActionListener(this);
      item2.addActionListener(this);
      item3.addActionListener(this);
      item4.addActionListener(this);
      sub1.addActionListener(this);
      sub2.addActionListener(this);
      menuBar.add(firstMenu);
      firstMenu.add(item1);
      firstMenu.add(item2);
      firstMenu.add(item3);
      firstMenu.add(item4);
      firstMenu.add(aSubMenu);

      this.setJMenuBar(menuBar);
```

Multi-Panel Containers

The JTabbedPane Class.

The JTabbedPane class allows the creation of a container that allows several panels to be added to the one space. Figure 9.17, “The JTabbedPane Example (a) first tab (b) second tab.” shows an example of a JTabbedPane with two tabs. You can select which container to view by pressing the tab at the top. To create a JTabbedPane simply use a default constructor:

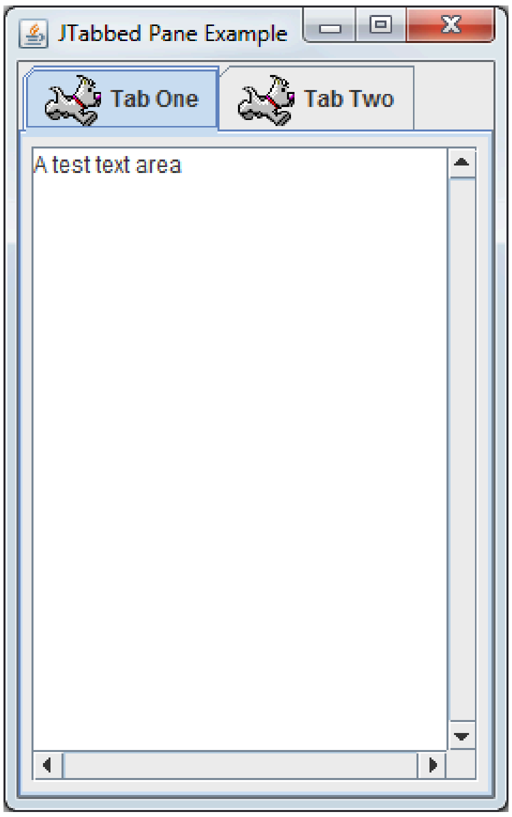
```
  JTabbedPane tabbedPane = new JTabbedPane();
```

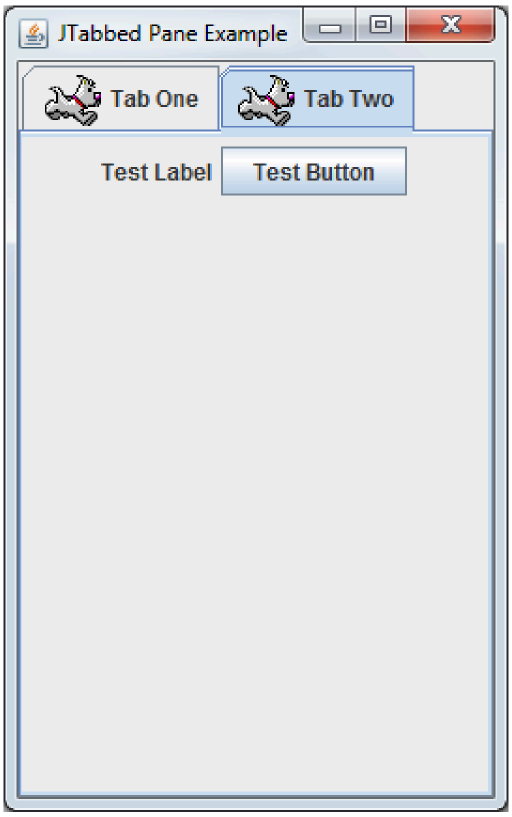
and then to add tabs to the JTabbedPane use:

```
  tabbedPane.addTab(String Title, ImageIcon icon, JPanel thePanel, String
 ToolTip);
```

For this example two JPanel objects are added to each tab. The first JPanel object contains a TextArea object in a JScrollPane container. The second JPanel container has a JLabel object and a JButton object.

Figure 9.17. The JTabbedPane Example (a) first tab (b) second tab.

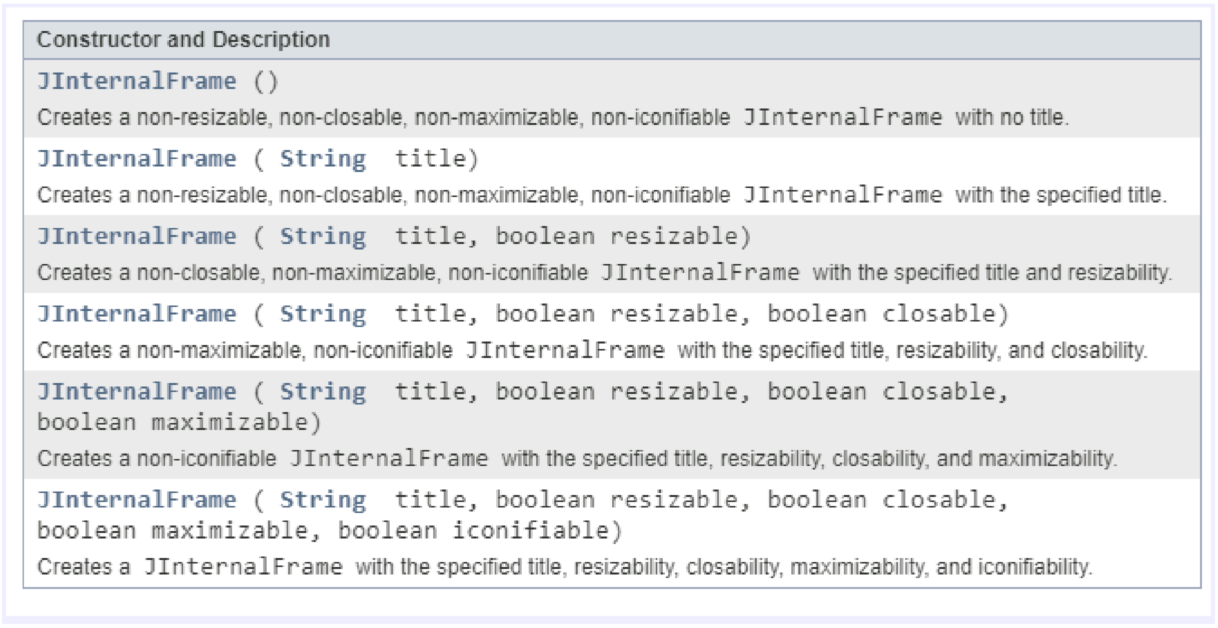




The full code for this example is here - JTabbedPaneExample.java

The JInternalFrame Class.

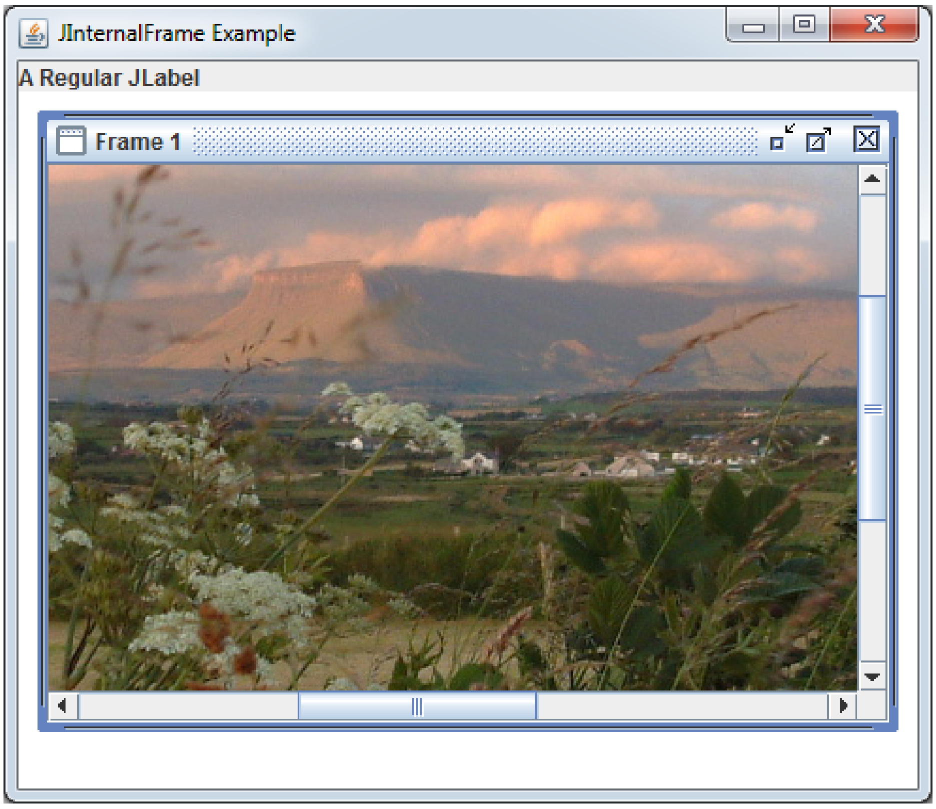
The JInternalFrame class allows you to create an application where you have windows within the main application. You may have seen this in applications like Adobe PhotoShop. To use JInternalFrame objects are added to a JDesktopPane, which is like a blank panel that contains the internal windows. Internal Frames are not windows, so they do not generate window events. They can be minimised to an icon within the desktop pane. The constructors for JInternalFrame are:



Constructor and Description
`JInternalFrame ()` Creates a non-resizable, non-closable, non-maximizable, non-iconifiable `JInternalFrame` with no title.
`JInternalFrame ( String title)` Creates a non-resizable, non-closable, non-maximizable, non-iconifiable `JInternalFrame` with the specified title.
`JInternalFrame ( String title, boolean resizable)` Creates a non-closable, non-maximizable, non-iconifiable `JInternalFrame` with the specified title and resizability.
`JInternalFrame ( String title, boolean resizable, boolean closable)` Creates a non-maximizable, non-iconifiable `JInternalFrame` with the specified title, resizability, and closability.
`JInternalFrame ( String title, boolean resizable, boolean closable, boolean maximizable)` Creates a non-iconifiable `JInternalFrame` with the specified title, resizability, closability, and maximizability.
`JInternalFrame ( String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)` Creates a `JInternalFrame` with the specified title, resizability, closability, maximizability, and iconifiability.

Figure 9.18, "The JInternalFrame Example" shows an example of the JInternalFrame in action. The full code for this example is below and in JInternalFrameExample.java.

Figure 9.18. The JInternalFrame Example



```java
package ee402;
import javax.swing.*;
import java.awt.*;
import java.net.URL;

@SuppressWarnings("serial")
public class JInternalFrameExample extends JFrame {

    public JInternalFrameExample() {
        super("JInternalFrame Example");
        JDesktopPane desktop = new JDesktopPane();
        this.getContentPane().add("North", new JLabel("A Regular JLabel"));
        this.getContentPane().add("Center", desktop);
        JInternalFrame internal1 = new JInternalFrame("Frame 1", true, true, true,
true);
        internal1.setSize(300,300);
        internal1.show();

        URL imageURL = this.getClass().getResource("test.jpg");
        Image testImage = this.getToolkit().getImage(imageURL);
        internal1.getContentPane().add(new JScrollPane(new JLabel(new
ImageIcon(testImage))));
        internal1.setLocation(50,50);
        desktop.add(internal1);
        this.setSize(400,400);
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new JInternalFrameExample();
    }
}
```

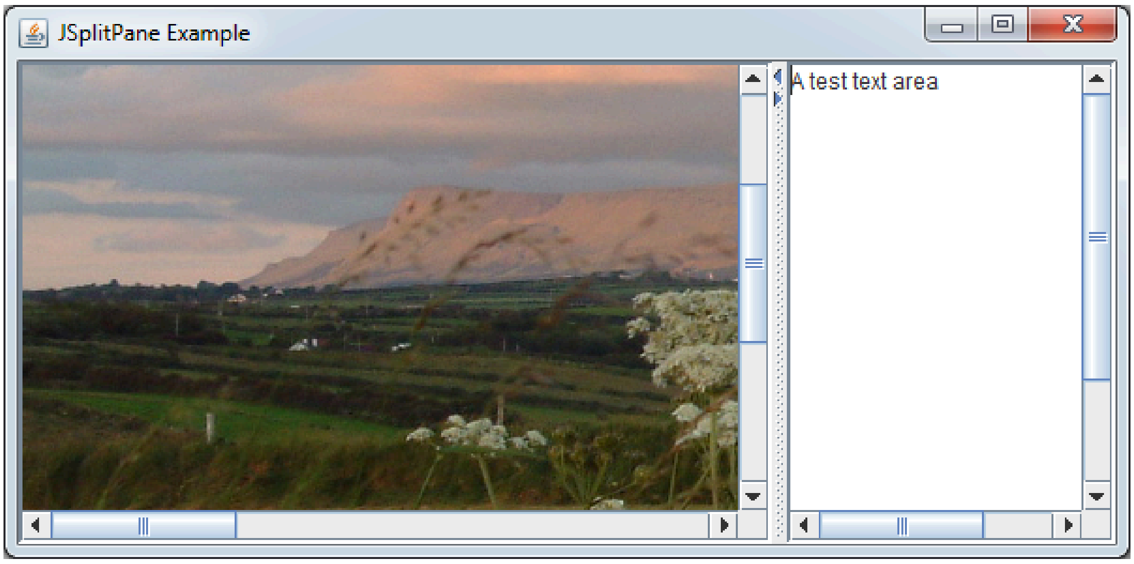
In this example the JFrame has a regular JLabel added to the "North" and the JDesktopPane added to the "Center". The JDesktopPane object is the container for the JInternalFrame objects. In this case there is only one JInternalFrame called internal1. In the creation of the JInternalFrame the title was set as "Frame 1" and it was allowed to be resizable, closable, maximizable and iconifiable (yes that is a real word!). The size was set at 300 x 300 using the setSize() method and then the show() method was called to make the internal frame visible. The show() method must be called to show the internal frames. An Image object is created and added to an ImageIcon object, which is in turn added to a JLabel and then this is added to a JScrollPane container. The location of the internal frame is set at (50,50) using the setLocation() method and finally the JInternalFrame object is added to the JDesktopPane container.

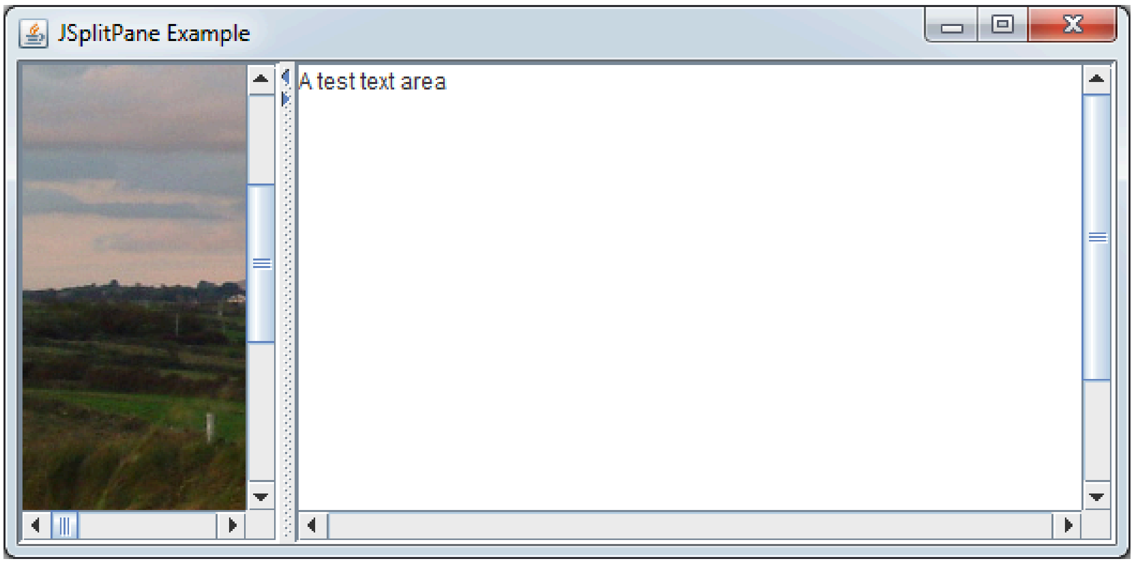
The full code for this example is here - JTabbedPaneExample.java. This application assumes an image in the same directory called test.jpg - You can use this one if you wish test.jpg (as with all files you can right-click and choose "Save Target As..")

The JSplitPane Class.

The JSplitPane class allows two components to be added to the split pane with a divider between them. The divider can be dragged left and right, if a horizontal split is used and top and bottom if a vertical split is used. You should really use JScrollPane containers around the JPanel or component that you use, as when you move the split in the split pane the container will not be completely visible. Figure 9.19, "The JSplitPane Example Application. Split (a) centre right (b) more right of centre." displays a JSplitPane container in action, where (a) and (b) show the split moved from left to right (note that the scrollbars move the container correctly). If you require more than two panes in the JSplitPane container, you can nest a JSplitPane container within another JSplitPane container.

Figure 9.19. The JSplitPane Example Application. Split (a) centre right (b) more right of centre.



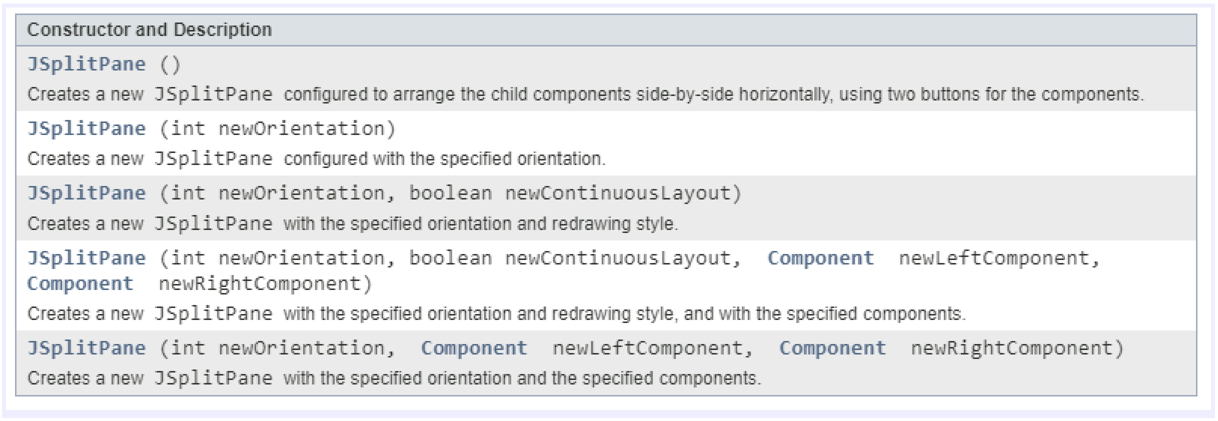


Here is the code for the example in Figure 9.19, "The JSplitPane Example Application. Split (a) centre right (b) more right of centre.", as below and as in JSplitPaneExample.java.

```java
package ee402;

import java.awt.Image;
import java.net.URL;
import javax.swing.*;

@SuppressWarnings("serial")
```

```java
public class JSplitPaneExample extends JFrame {

        public JSplitPaneExample() {
                super("JSplitPane Example");

                JScrollPane pane2 = new JScrollPane(
                        new JTextArea("A test text area",20,20),
                        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);

                URL imageURL = this.getClass().getResource("test.jpg");
                Image testImage = this.getToolkit().getImage(imageURL);

                JScrollPane pane1 = new JScrollPane(new JLabel(new ImageIcon(testImage)),
                        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);

                JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                           pane1, pane2);
                splitPane.setOneTouchExpandable(true);
                splitPane.setDividerLocation(400);

                this.getContentPane().add("Center", splitPane);
                this.setSize(600,400);       // set the size
                this.setVisible(true);
        }

        public static void main(String[] args) {
                new JSplitPaneExample();
        }
}
```

In this example two JScrollPane objects are added to the JSplitPane using the constructor of the object:



Constructor and Description
`JSplitPane ()` Creates a new `JSplitPane` configured to arrange the child components side-by-side horizontally, using two buttons for the components.
`JSplitPane (int newOrientation)` Creates a new `JSplitPane` configured with the specified orientation.
`JSplitPane (int newOrientation, boolean newContinuousLayout)` Creates a new `JSplitPane` with the specified orientation and redrawing style.
`JSplitPane (int newOrientation, boolean newContinuousLayout, Component newLeftComponent, Component newRightComponent)` Creates a new `JSplitPane` with the specified orientation and redrawing style, and with the specified components.
`JSplitPane (int newOrientation, Component newLeftComponent, Component newRightComponent)` Creates a new `JSplitPane` with the specified orientation and the specified components.

where JSplitPane.HORIZONTAL_SPLIT defines the orientation and the next two objects are the JScrollPane components. When the setOneTouchExpandable() is passed a value of true then the divider displays two little arrows, when pressed on fully expand the left of right pane (you can see these arrows in Figure 9.19, “The JSplitPane Example

Application. Split (a) centre right (b) more right of centre.” at the top of the divider). The divider location can be placed using the setDividerLocation().

There are other methods that are useful for the JSplitPane

- setDividerSize(int) - Sets the width of the divider.
- resetToPreferredSize() - Sets the size of the split pane based on the preferred size of the added components.
- setResizeWeight(double) - Sets how the scroll pane should react when it is resized. A value of 0 specifies the right component gets all the new space, 1 specifies the left component gets all the new space - 0.5 gives both sides equal new space.
- setRightComponent() - Allows you to specify a new component for the right hand side. setLeftComponent() allows you to specify a new component for the left hand side.

Dialogs and More Components

Dialogs

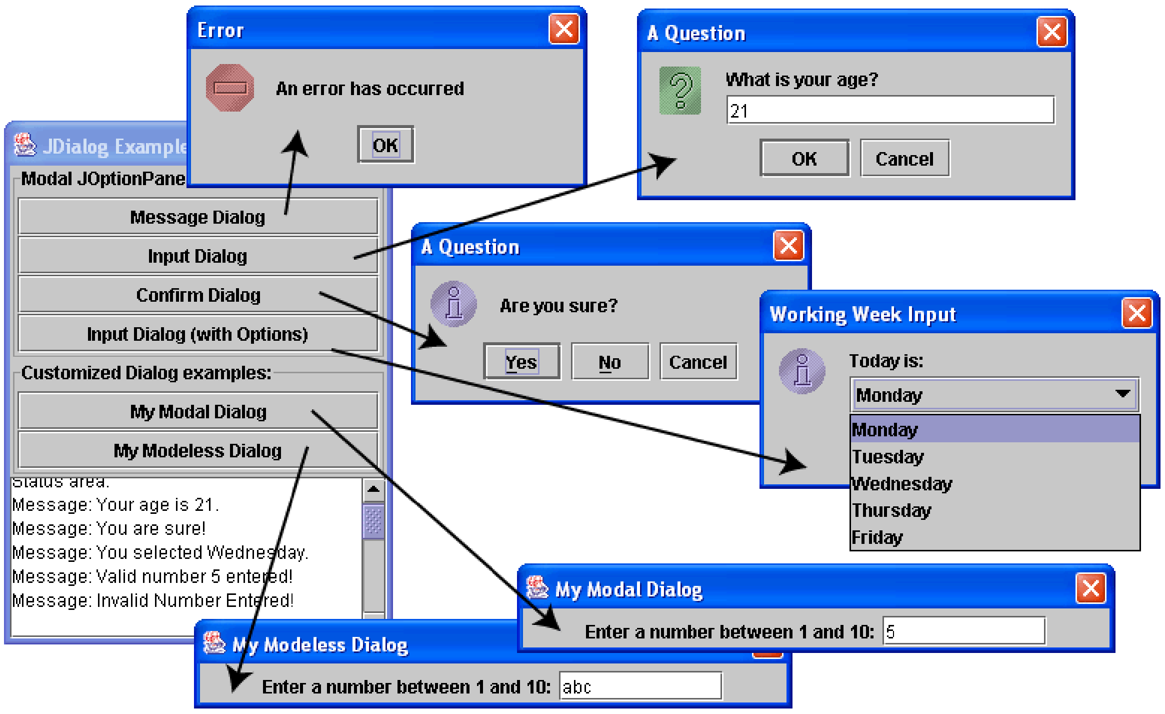
The JDialog class is the key class for creating dialog windows when using the Swing API. It is a child of the equivalent AWT Dialog class. A dialog is a pop-up window, usually reasonably small and non-resizable that displays or requests information to or from the user. You can create your own dialog windows or use some standard dialogs that are available.

A dialog is either modal or non-modal. A modal dialog, once visible, does not allow you to interact with the application in any other way, than through the dialog. A non-modal dialog gives you access to the dialog and to the application at the same time.

It is possible to create a very simple dialog using the JOptionPane class. This class allows you to display a message along with a predefined icon.

I have written an example to demonstrate most dialog types, as in Figure 9.20, “The Dialog Example Application” and with the full source code in - JDialogExample.java.

Figure 9.20. The Dialog Example Application

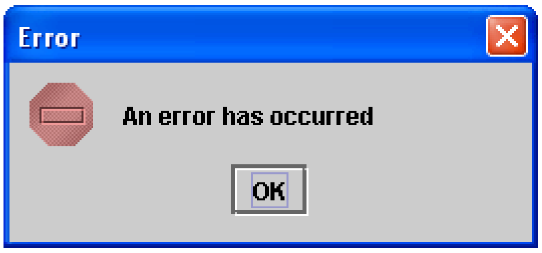


```
Status area:
Message: Your age is 21.
Message: You are sure!
Message: You selected Wednesday.
Message: Valid number 5 entered!
Message: Invalid Number Entered!
```

When the buttons are pressed in Figure 9.20, "The Dialog Example Application" the different dialogs open to demonstrate the different dialog types.

When the first button Message Dialog is pressed an ActionEvent is generated and a Message Dialog is created as in Figure 9.21, "The Message Dialog", where all you can do is read the message and press the "OK" or window "X" buttons.

Figure 9.21. The Message Dialog



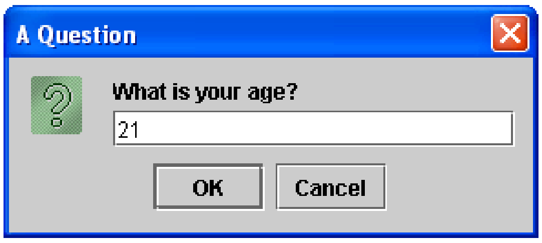
This is created using the code:

```
        JOptionPane.showMessageDialog(this, "An error has occurred", "Error",
 JOptionPane.ERROR_MESSAGE);
```

Where this refers to the parent Component, "An error has occurred" is the message, "Error" is the title, and JOptionPane.ERROR_MESSAGE is the pre-defined icon to use.

When the second button Input Dialog is pressed an ActionEvent is generated and an Input Dialog is created as in Figure 9.22, "The Input Dialog". The Input Dialog allows you to enter a value that is returned to the calling application as a String object. In this example the calling application takes the String object and displays it in the TextArea.

Figure 9.22. The Input Dialog



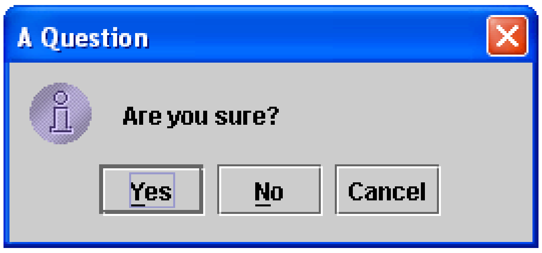
This is created using the code:

```
        String s = JOptionPane.showInputDialog(this, "What is your age?",
                    "A Question", JOptionPane.QUESTION_MESSAGE);
                this.sendMessage("Your age is " + s + ".\n");
```

Where this refers to the parent Component, "What is your age?" is the message, "A Question" is the title, and JOptionPane.QUESTION_MESSAGE is the predefined icon to use.

When the third button Confirm Dialog is pressed an ActionEvent is generated and a Confirm Dialog is created as in Figure 9.23, “The Confirm Dialog”. The Confirm Dialog allows you to have a selection of options, in this case "yes", "no" and "cancel", which when pressed causes an int value to be returned to the calling application. In this application the confirm details will be displayed in the TextArea component.

Figure 9.23. The Confirm Dialog



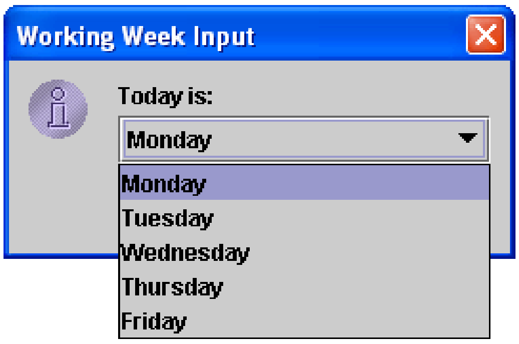
This is created using the code:

```


    int selected = JOptionPane.showConfirmDialog(this, "Are you sure?",
               "A Question", JOptionPane.YES_NO_CANCEL_OPTION,
               JOptionPane.INFORMATION_MESSAGE);
    if (selected == JOptionPane.NO_OPTION)
           this.sendMessage("You are not sure!\n");
    else if (selected == JOptionPane.YES_OPTION)
           this.sendMessage("You are sure!\n");
    else this.sendMessage("You cancelled!\n");


```

Where this refers to the parent Component, "Are you sure?" is the message, "A Question" is the title, YES_NO_CANCEL_OPTION is the pre-defined button arrangement to use and JOptionPane.INFORMATION_MESSAGE is the pre-defined icon to use. We can then compare the selected int value to the defined values of JOptionPane.NO_OPTION or JOptionPane.YES_OPTION

When the fourth button Input Dialog (with Options) is pressed an ActionEvent is generated and an Input Dialog is created as in Figure 9.24, "The Input Dialog with Options". The Input Dialog allows you to have a selection of options that you can choose from a pull down list. In this application the selected value will be displayed in the TextArea component.

Figure 9.24. The Input Dialog with Options



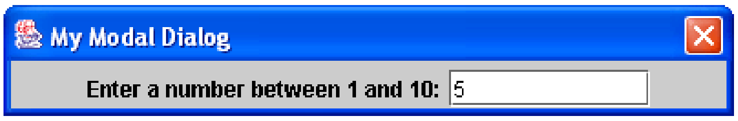
This is created using the code:

```


String[] theDays = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday" };
String selected = (String) JOptionPane.showInputDialog( this,
               "Today is: ", "Working Week Input",
               JOptionPane.INFORMATION_MESSAGE, null,
               theDays, theDays[0]);
               this.sendMessage("You selected " + selected + ".\n");


```

First off an array of String is created and this is passed as the options to the input dialog. The call to the input dialog will return an Object object that can be cast converted to a String objects, as the list we passed is an array of String objects. The method call showInputDialog() expects the parameters, Where this refers to the parent Component, "Today is: "is the message, "Working Week Input" is the title, JOptionPane.INFORMATION_MESSAGE is the pre-defined icon to use, null is a separate Icon object to use, theDays is the array of Object expected and theDays[0] is the inital selection.

When the fifth and sixth buttons My Modal Dialog and My Modeless Dialog are pressed an ActionEvent is generated and a special MyDialog is created as in Figure 9.25, "The MyDialog (modal and modeless look the same)". This dialog is our own implementation of JDialog where MyDialog is a child class of JDialog providing our own specific requirements. In this case it creates the GUI to request a number between 1 and 10 and validates to ensure that the number is indeed in this range. Invalid results and correct results are sent to the main application, as the MyDialog is passed a reference to the main application in its constructor.

Figure 9.25. The MyDialog (modal and modeless look the same)



Enter a number between 1 and 10: 5

This is created using the code (DialogExample.java):

```java


In the main application....

else if(e.getSource().equals(myDialogButton))
{
   myDialog = new MyDialog(this, "My Modal Dialog", this, true);
}
else if(e.getSource().equals(myModelessButton))
{
  myDialog = new MyDialog(this, "My Modeless Dialog", this, false);
}
....

// The MyDialog class implementation

class MyDialog extends JDialog implements ActionListener
{
  private JTextField entryField;
  private JDialogExample callingApp;

  public MyDialog(Frame frame, String title, JDialogExample callingApp,
                  boolean isModal)
  {
    super(frame, title, isModal);
    this.callingApp = callingApp;

    this.getContentPane().setLayout(new FlowLayout());
    this.getContentPane().add(new JLabel("Enter a number between 1 and 10:"));
    entryField = new JTextField(10);
    entryField.addActionListener(this);
    this.getContentPane().add(entryField);

    //prevent the dialog from closing without a value
    this.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);

    this.setSize(400,60);
    this.show();
  }

  public void actionPerformed(ActionEvent e)
  {
    String entry = entryField.getText();
    try{
      Integer i = new Integer(entry);
      if (i.intValue()<=10 && i.intValue()>=1)
      {
        callingApp.sendMessage("Valid number "+ i +" entered!\n");
        this.dispose();
```

```
            }
            else
            {
              callingApp.sendMessage("Invalid Number Entered " + i +"!\n");
            }
          }
          catch (NumberFormatException exept)
          {
            callingApp.sendMessage("Invalid Number Entered!\n");
          }
        }
      }


```

So at the top of this segment of code, if the ActionEvent was generated by either the fifth or sixth buttons then and instance of my Dialog class MyDialog is created and then runs. There are two forms here, modal and modeless, and all that differentiates between the calls is the true and false values that are passed to the parent class of MyDialog JDialog through the super() call. A modal dialog box takes all control and will not allow you to press any other place in the same program, whereas the modeless dialog box allows you to still use the calling application in full. There are many different reasons for using either

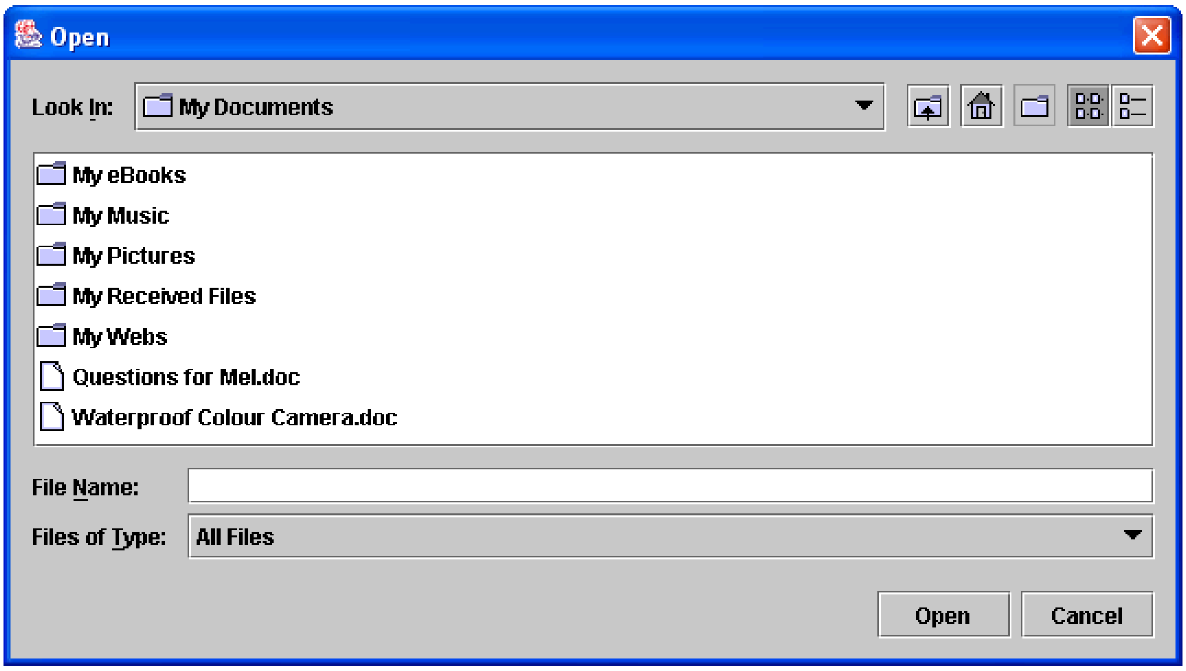
The MyDialog is passed a reference to the calling application to its constructor called callingApp. This is important as it allows us to communicate with the calling application to call its public sendMessage() method. The constructor then sets up the display by adding JLabel and JTextField components. One unusual call in the constructor is to the method setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE) which prevents the user from pressing the "X" button in the frame of the dialog to avoid entering a value - try it! The actionPerformed() method takes the input from the JTextField when enter is pressed in the field and validates the data in that field. It tries to convert the String into an Integer, which will throw an NumberFormatException if invalid text, such as "abc" is entered. If this exception does not occur then the Integer object will be converted to an int variable and checked if it is in the range 1 to 10. If it is then the MyDialog dialog will pass the value back to the calling application through the calling application's sendMessage() method. The dialog will then dispose of itself.

Remember that the code for all these examples is in file - JDialogExample.java. You should compile and run this example if you do not understand the difference between modal and modeless. Press the sixth button and then press another dialog button, try to do this with any other combination. The only modeless dialog is the last dialog in this example.

The File Chooser Dialog

The file chooser dialog is a useful dialog that allows the user of your application to browse their filesystem to find a file or folder. The JFileChooser is the Swing implementation of the file chooser and has several powerful features, such as allowing you to browse for certain types of file only (such as .gif images), or to even provide a preview of the files you are browsing. Figure 9.26, “The JFileChooser Dialog in Action.” shows an example of the JFileChooser in operation.

Figure 9.26. The JFileChooser Dialog in Action.



To create the dialog as shown in Figure 9.26, "The JFileChooser Dialog in Action." use something like the following code:

```


JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(this);  // where this is the parent
if(returnVal == JFileChooser.APPROVE_OPTION)
{
   File f = chooser.getSelectedFile();
   System.out.println("You have chosen " + f.getName());
}


```

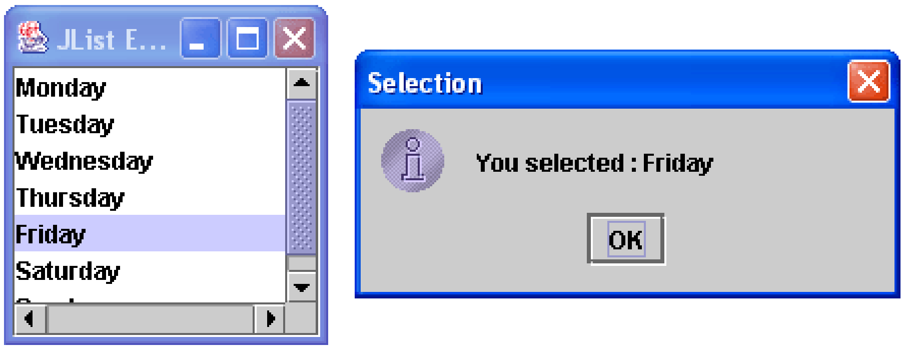
In this case a default JFileChooser was created, a current directory can be passed to the constructor as a File object. When the showOpenDialog() method is called, the return value defined if open or cancelled was pressed. If the open button was pressed then an int value of JFileChooser.APPROVE_OPTION will be returned. Once a file has been chosen you can obtain a reference to the file that was chosen, as an object of the file pointer java.io.File. This class allows you to perform operations on files, such as opening, closing, deleting. Other useful methods of this class are isDirectory() which return a boolean to define if the chosen file is a directory. length() returns a long that gives the size of the file in bytes. getPath() returns the path of the files as a String object.

Other Swing Components

The JList Class

The JList is the Swing replacement for the java.awt.List component. It differs from the AWT component in that it displays an array of Objects, not just an array of String. This allows you to create complex displays, such as displaying images in the JList component. It is common to use a Vector of objects to hold the data. Once again, this component does not support scrolling, you must pass this task onto a component such as JScrollPane.

Figure 9.27. The JList Example Application



You selected : Friday

The code for this example is given below and in JListExample.java

```
package ee402;
import javax.swing.*;
import javax.swing.event.*;

@SuppressWarnings("serial")
public class JListExample extends JFrame implements    ListSelectionListener {
        private JList<String> myJList;
        public JListExample() {
                super("JList Example");
                String[] data = {"Monday","Tuesday","Wednesday",
                    "Thursday","Friday","Saturday","Sunday"};

                myJList = new JList<String>(data);
                myJList.addListSelectionListener(this);
                JScrollPane p = new JScrollPane(this.myJList,
                        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
                this.getContentPane().add("Center", p);
                this.pack();
                this.setVisible(true);
        }
        public void valueChanged(ListSelectionEvent e) {
                JOptionPane.showMessageDialog(this, "You selected : " +
                     myJList.getSelectedValue(),
                     "Selection", JOptionPane.INFORMATION_MESSAGE);
        }
        public static void main(String[] args) {
                new JListExample();
        }
}
```

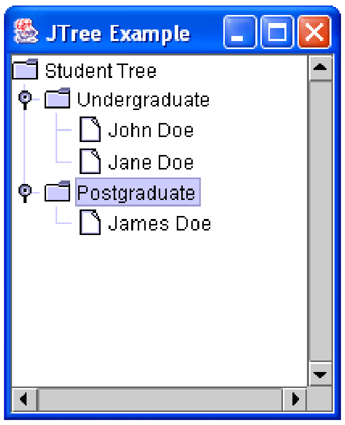
The list is populated by a String array, by passing the array to the constructor of the JList class. You can add a ListSelectionListener listener to the JList component. In this case the JListExample class is responsible for handling the ListSelectionEvent event object and so according to the ListSelectionListener interface the class must implement the valueChanged() method that receives a ListSelectionEvent event object. This application is very annoying to use, as it pops up a dialog every time you select any item from the list component. The dialog is created using the JOptionPane class as discussed in the section called "Dialogs".

The JList component allows for the multiple selection of items in the list. If you use the method myList.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION) you can have multiple selections (it is actually the default state). To get the multiple selected items you can call the getSelectedValues() that returns an array of Objects that can be appropriately cast converted to an array of String if required. Another useful method is the getSelectedIndex() that returns an int variable, defining the position of the item in the list that was selected. getSelectedIndices() returns an array of int if multiple selection is allowed.

The JTree Class

The JTree is very useful for displaying any type of hierarchical data. It displays data in the form of a real tree, with nodes, branches etc. Figure 9.28, "The JTree Example Application" shows a very basic tree in operation. You can set the JTree so that events are generated whenever the tree is expanded or collapsed, or when a particular cell is selected.

Figure 9.28. The JTree Example Application



The code for this example is given below and in - [JTreeExample.java]

```
package ee402;
import javax.swing.*;
import javax.swing.tree.*;
```

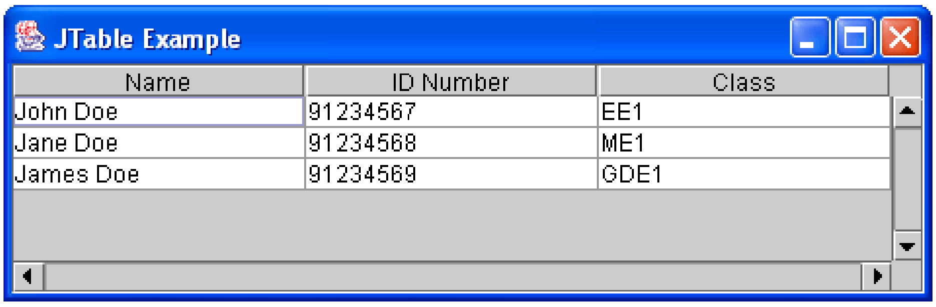
```
@SuppressWarnings("serial")
public class JTreeExample extends JFrame
{
	JTree myTree;

	public JTreeExample() {
		super("JTree Example");
		DefaultMutableTreeNode top = new DefaultMutableTreeNode("Student Tree");
		myTree = new JTree(top);
		DefaultMutableTreeNode ugNode = new
DefaultMutableTreeNode("Undergraduate");
		DefaultMutableTreeNode johnDoe = new DefaultMutableTreeNode(
			new Student("John Doe", "912345676", "EE1"));
		DefaultMutableTreeNode janeDoe = new DefaultMutableTreeNode(
			new Student("Jane Doe", "912345677", "ME1"));
		DefaultMutableTreeNode pgNode = new
DefaultMutableTreeNode("Postgraduate");
		DefaultMutableTreeNode jamesDoe = new DefaultMutableTreeNode(
			new Student("James Doe", "912345678", "GDE1"));
		top.add(ugNode);
		ugNode.add(johnDoe);
		ugNode.add(janeDoe);
		top.add(pgNode);
		pgNode.add(jamesDoe);
		JScrollPane p = new JScrollPane(this.myTree,
			JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
			JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
		this.getContentPane().add("Center", p);
		this.pack();
		this.setVisible(true);
	}

	public static void main(String[] args) {
		new JTreeExample();
	}
}
class Student
{
	private String name;
	private String id;
	private String classname;
	Student(String name, String id, String classname) {
		this.name = name;
		this.id = id;
		this.classname = classname;
	}

	public String toString() {
		return name;
	}
}
```

There are a few important points to note in this example. First off, the javax.swing.tree package must be imported if you wish to use the default tree nodes. In this application I have used the relatively straightforward DefaultMutableTreeNode for all my nodes, even the root node, by using the node object passed to the constructor of JTree. I have created a temporary Student class to use as the objects in the tree. Note that in this class the toString() method inherited from Object is over-ridden, to provide our own support for converting a Student object into a String object. This allows for the object to be displayed correctly on our tree. The nodes are added to other nodes and thus the tree builds up.

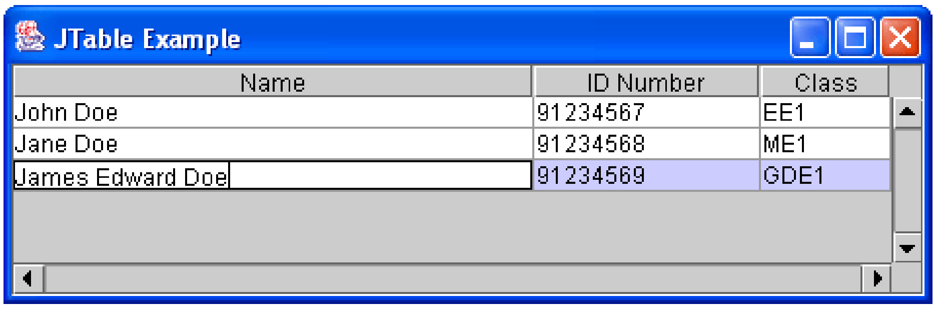
The JTable Class.

The JTable Class has many uses in displaying data in rows and columns, very useful in the display of a database lookup query. Figure 9.29, "The JTable Example Application. (a) As it appears on startup (b) moved and edited a bit" shows an example of a basic table in operation. In (a) you can see that the table is displaying a set of strings in a row/column layout. The titles are given on the top of the columns and the component allows you to move the dividers between cells, just like you can in Excel. As you can see in (b) the contents of the cells can be edited by default. There is no storage mechanism at this stage, you are simply editing the data on the table component.

Figure 9.29. The JTable Example Application. (a) As it appears on startup (b) moved and edited a bit



Name	ID Number	Class
John Doe	91234567	EE1
Jane Doe	91234568	ME1
James Doe	91234569	GDE1



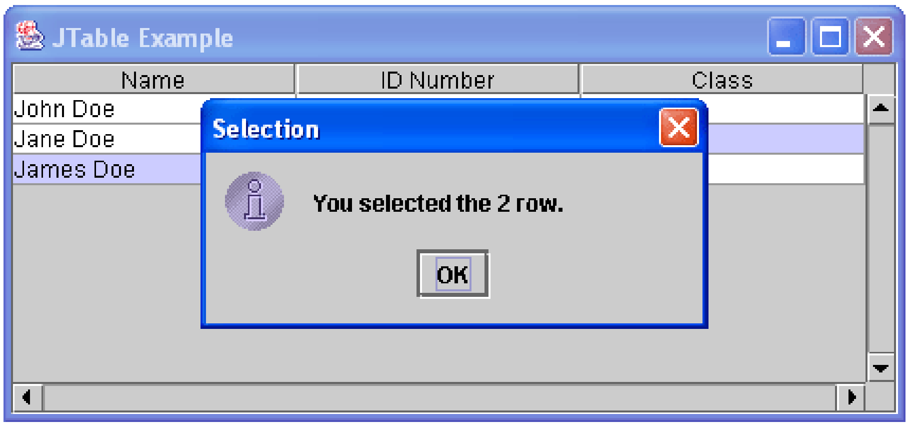
Name	ID Number	Class
John Doe	91234567	EE1
Jane Doe	91234568	ME1
James Edward Doe	91234569	GDE1

The code for this example is given in JTableExample.java

If you wish to allow selection of the table you can add a ListSelectionInterface interface to the JTable, just like for JLIst in the section called "The JList Class". Figure 9.30, "The JTable Example Application with ListSelection events." shows an example with

ListSelection events associated with the table. When you select a row the dialog will pop-up to say that you have selected that row.

Figure 9.30. The JTable Example Application with ListSelection events.



You selected the 2 row.

The code for this example is given below and in - [JTableExample2.java]

```


// Swing JTableExample2 - Derek Molloy
//       Table with events added in.

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

public class JTableExample2 extends JFrame implements ListSelectionListener
{
  JTable myTable;


  public JTableExample2()
  {
         super("JTable Example");

         String[] columnNames = {"Name", "ID Number", "Class"};
         String[][] data = {{"John Doe", "91234567", "EE1" },
                           {"Jane Doe", "91234568", "ME1" },
                           {"James Doe", "91234569", "GDE1" }};

         myTable = new JTable(data, columnNames);
         myTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

         ListSelectionModel selectModel = myTable.getSelectionModel();
         selectModel.addListSelectionListener(this);

```

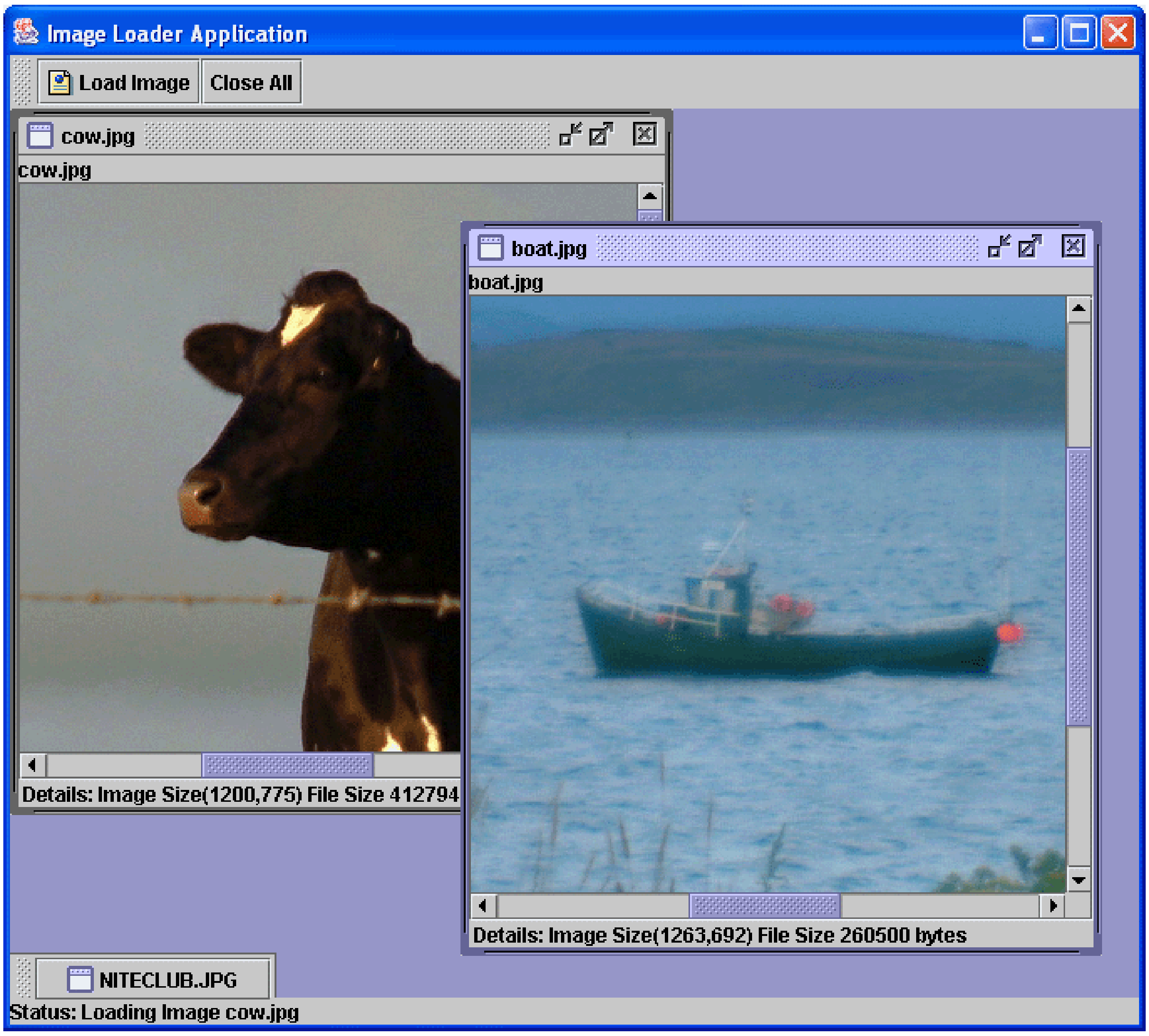
```java
        JScrollPane p = new JScrollPane(this.myTable,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);

        this.getContentPane().add("Center", p);

        this.pack();
        this.show();
    }

    public void valueChanged(ListSelectionEvent e)
    {
        ListSelectionModel lsm = (ListSelectionModel)e.getSource();
        int selectedRow = lsm.getMinSelectionIndex();
        JOptionPane.showMessageDialog(this, "You selected the " +
selectedRow + " row.",
                "Selection", JOptionPane.INFORMATION_MESSAGE);

    }

    public static void main(String[] args)
    {
        new JTableExample2();
    }
}
```

In this example the selection mode is set from multiple to single, so that only one row may be selected at a time. The ListSelectionModel interface object defines the selection model for the JTable, allowing us to associate the ListSelectionListener interface with the selection process. When a ListSelectionEvent object is generated this is handled by the valueChanged() method, where we can interrogate the event object e to get the selected index of the event, through the ListSelectionModel interface.

An Exercise for You. The Image Loader Application

Task: Write an Image Loader Application as shown in Figure 9.31, "The Image Loader Application". The Toolbar should have two functions, Load Image and Close all Images. The Load Image should call up a file chooser as shown in Figure 9.32, "The Image Loader Application (with File Chooser open)". The internal windows should display the name of the file, the image size and the size of the file from which it was loaded. The internal windows should be minimizable, maximizable, closable and iconifable. The main window should have a status label that displays details of the application as it runs. You can use my file icon if you wish - load.gif. Finally when the close all button is pressed the application should prompt the user to see if this is correct as in Figure 9.33, "The Image Loader Application (with confirm on close)".

Figure 9.31. The Image Loader Application

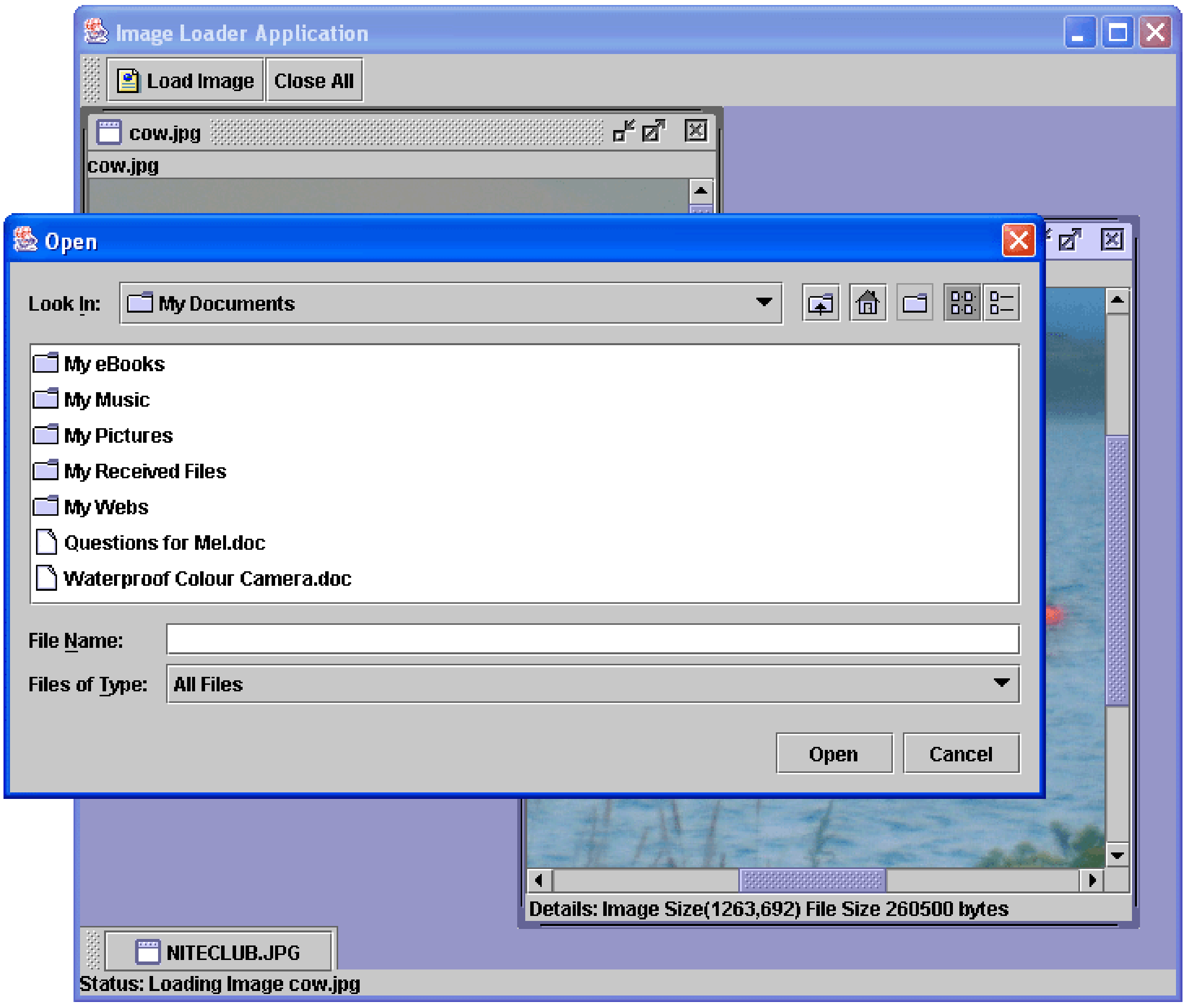


Details: Image Size(1200,775) File Size 412794

Details: Image Size(1263,692) File Size 260500 bytes

Status: Loading Image cow.jpg

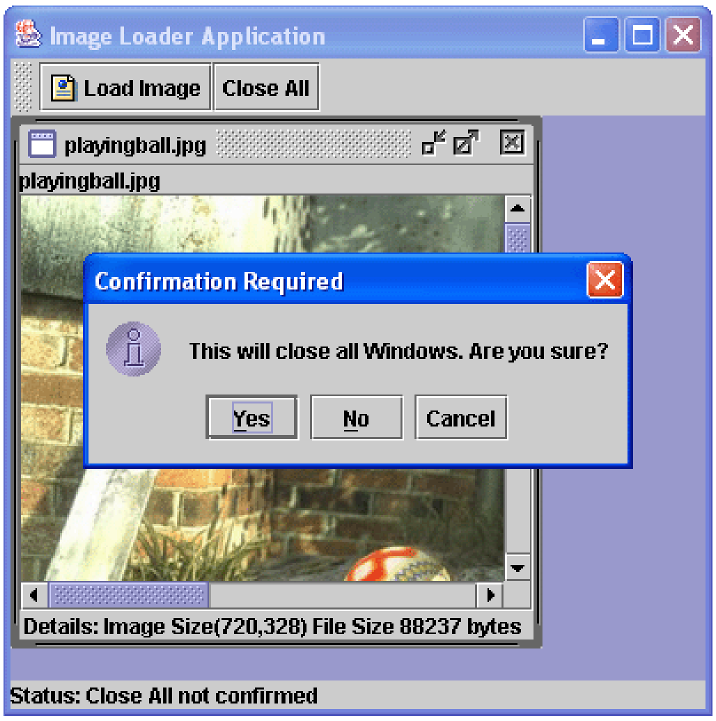
Figure 9.32. The Image Loader Application (with File Chooser open)



Details: Image Size(1263,692) File Size 260500 bytes

Status: Loading Image cow.jpg

Figure 9.33. The Image Loader Application (with confirm on close)



playingball.jpg

This will close all Windows. Are you sure?

Details: Image Size(720,328) File Size 88237 bytes

Status: Close All not confirmed