ITS Information Technology Service

Guide 108

Version 1.2

Advanced Java

The tutorial given in ITS *Guide 58: Getting started with Java* provides basic information on developing programs in the Java programming language. This Guide introduces some other topics on Java. In particular, it discusses:

- **the creation of Java programs that have graphical user interfaces (GUIs);**
- **the handling of collections of data using the List, Set and Map interfaces of Java's Collections API;**
- **the production of Java applets, code that gets executed when a person visits a WWW page.**

Durham University

£1

Document code: **Guide 108**
Title: **Advanced Java**
Version: **1.2**
Date: **June 2006**
Produced by: **University of Durham Information Technology Service**

Conventions:

In this document, the following conventions are used:

- A typewriter font is used for what you see on the screen.
- A **bold typewriter font** is used to represent the actual characters you type at the keyboard.
- A ***slanted typewriter font*** is used for items such as filenames which you should replace with particular instances.
- A **bold font** is used to indicate named keys on the keyboard, for example, **Esc** and **Enter**, represent the keys marked Esc and Enter, respectively.
- A **bold font** is also used where a technical term or command name is used in the text.
- Where two keys are separated by a forward slash (as in **Ctrl/B**, for example), press and hold down the first key (**Ctrl**), tap the second (**B**), and then release the first key.

Contents

1 Introduction 1

2 Providing a graphical user interface (GUI) 1

2.1 APIs for producing GUIs 1
2.2 What the Swing API includes and how it is organised 2
2.3 A simple example of a GUI 2
2.4 Stage A: obtaining the current date and time 2
2.5 Stage B: creating a window 3
2.6 Stage C: adding GUI components to the window 4
2.7 Stage D: responding to a click of the button 6
2.8 Stage E: altering the JTextField component 8
2.9 Stage F: closing the window 10
2.10 Conclusion 12

3 The Collections API 13

3.1 An introduction to the Collections API 13
3.2 The interface List and the classes ArrayList and LinkedList 14
3.3 Using the Iterator interface 18
3.4 The methods contains, indexOf, lastIndexof and remove 19
3.5 An example of a complete program that manipulates a list 22
3.6 Conclusion 23

4 Writing applets (for use with the WWW) 23

4.1 Using HTML to code WWW pages 23
4.2 Getting Java bytecodes executed when a WWW page is visited 24
4.3 Deriving from Applet instead of declaring a main method 25
4.4 Dealing with the different versions of the Java platform 27
4.5 Using appletviewer when developing Java applets 28
4.6 The lifecycle of a Java applet 30
4.7 Overriding the init method 30
4.8 Restrictions imposed on Java applets 31
4.9 Reworking an application as an applet: GetDateApplet 32
4.10 Producing code that can be used either as an application or an applet 33
4.11 Using the Java archive tool 33

5 Other information about Java 34

1 Introduction

The tutorial given in ITS *Guide 58: Getting started with Java* provides basic information on developing programs in the Java programming language. This Guide introduces some other topics on Java. In particular, it discusses:

- the creation of Java programs that have graphical user interfaces (GUIs);
- the handling of collections of data using the List, Set and Map interfaces of Java's Collections API;
- the production of Java applets, code that gets executed when a person visits a WWW page.

This Guide refers to the WWW pages documenting the Core APIs: *http://java.sun.com/j2se/1.4.2/docs/api*. These WWW pages can also be downloaded to filespace on your own computer. This Guide uses the notation *$API/java/lang/String.html* to refer to the WWW page *http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html*.

2 Providing a graphical user interface (GUI)

2.1 APIs for producing GUIs

One of the attractive features of Java is that it has APIs for producing GUIs. One of these APIs is called the *Abstract Windowing Toolkit* (or *AWT*), and it is provided in the package java.awt. Although the AWT has been present from the start, the facilities that the Java platform provides for producing GUIs have changed with each major release of Java.

In *JDK 1.0*, a reasonably comprehensive set of features were provided. However, events such as mouse movements, button clicks, and window closing had to be handled in a way which led to inefficient code, and code that was inappropriate in an object-oriented system.

In *JDK 1.1*, the *event-handling* mechanism was changed: instead, an object can register itself to handle any events on a particular GUI component (such as a mouse, a button or a window).

With the release of the Java 2 Platform in December 1998, a new set of classes for building GUIs was introduced. These classes form what is known as the *Swing API*. Unlike the AWT, the code of the classes that implement the Swing API is completely written in Java. Because of this, it is easy for a programmer to add new GUI components that can be used alongside the Swing components. However, when writing programs that use the Swing API, it is still necessary to use some of the basic classes of the java.awt package.

The Swing API also has a *pluggable look-and-feel*. The *look* of a window in a Windows environment is different from that in a Motif environment running on a UNIX workstation. With the Swing API, you can choose the *look-and-feel* to be that of a particular platform, to be a platform-independent look-and-feel, or to be a look-and-feel that depends on the platform on which the program is running.

Unfortunately, during the various beta releases of the Swing API, the position of the Swing API moved. This has been inconvenient for those people developing code (or looking at books) that use this API. Although it has previously resided at com.sun.java.swing and later at java.awt.swing, the Swing API is now in the javax.swing package.

2.2 What the Swing API includes and how it is organised

The package javax.swing consists of many classes. It provides *GUI component*s such as buttons, checkboxes, lists, menus, tables, text areas, and trees. It also includes GUI components that are *container*s (such as menu bars and windows), and higher-level components (such as dialog boxes, including dialog boxes for opening or saving files). And there are also classes for basic drawing operations, and for manipulating images, fonts and colours, and for handling events such as mouse clicks.

Many of these GUI components will have common features. For example, there is a method called setBackground that can be used to alter the background colour of a component. Although it would be possible to include a method declaration called setBackground in each of the classes, this is not sensible. Because Java has *inheritance*, it allows classes to be arranged in a *class hierarchy*: this means the Swing designers can declare the setBackground method in a class high up in the class hierarchy and it is automatically available in the classes that are lower down in the class hierarchy. So, an extensive class hierarchy is used to organise the classes of the Swing API (and the AWT).

2.3 A simple example of a GUI

Suppose we want a Java program that creates a window that has a button and a textfield (an area for storing a line of text), and each time the button is clicked the textfield is updated to show the current date and time.

Rather than just present the program that accomplishes this task, the program will be developed in stages, each stage conquering some of the problems that occur.

2.4 Stage A: obtaining the current date and time

To begin with, we need to know how to get the current date and time. The class Date from the java.util package can be used to do this. So the following program can be used to output the current date and time:

```
1: //                                          // GetDateProg.java
2: // Stage A: outputting the current date and time to the screen.
3: // Barry Cornelius, 22nd November 1999
4: import java.util. Date;
5: public class GetDateProg
6: {
7:    public static void main(final String[] pArgs)
8:    {
9:       final Date tDate = new Date();
10:      System.out.println(tDate);
11:   }
12: }
```

2.5 Stage B: creating a window

When producing a GUI, we will need to create windows on the screen. The Swing API has a number of classes that enable a program to create a new window on the screen or to make use of an existing window.

The classes are:

- JWindow — which allows a window without a border or a menu bar to be displayed;
- JFrame — which allows a window with a border and possibly a menu bar to be displayed;
- JDialog — which allows a dialog box to be displayed;
- JInternalFrame — which allows a frame to be created inside an existing frame;
- JApplet — which allows the frame of a WWW page to be accessed by a Java *applet*.

Here is a simple program that displays a new window on the screen:

```
13: // Stage B: creating a window.                    // GetDateProg.java
14: // Barry Cornelius, 22nd November 1999
15: import javax.swing. JFrame;
16: public class GetDateProg
17: {
18:    public static void main(final String[] pArgs)
19:    {
20:       final JFrame tJFrame = new JFrame("GetDateProg: Stage B");
21:       tJFrame.setLocation(50, 100);
22:       tJFrame.setSize(300, 200);
23:       tJFrame.setVisible(true);
24:    }
25: }
```

The program creates an object of the class JFrame. One of JFrame's constructors allows you to choose the string that is put into the title bar of the window:

```
JFrame tJFrame = new JFrame("GetDateProg: Stage B");
```

The use of this *class instance creation expression* just creates the JFrame object: it does not display the window on the screen. This is done by a call of the method setVisible:

```
tJFrame.setVisible(true);
```

Unless you specify otherwise, when the window is displayed, it will be positioned in the top left-hand corner of the screen. The call:

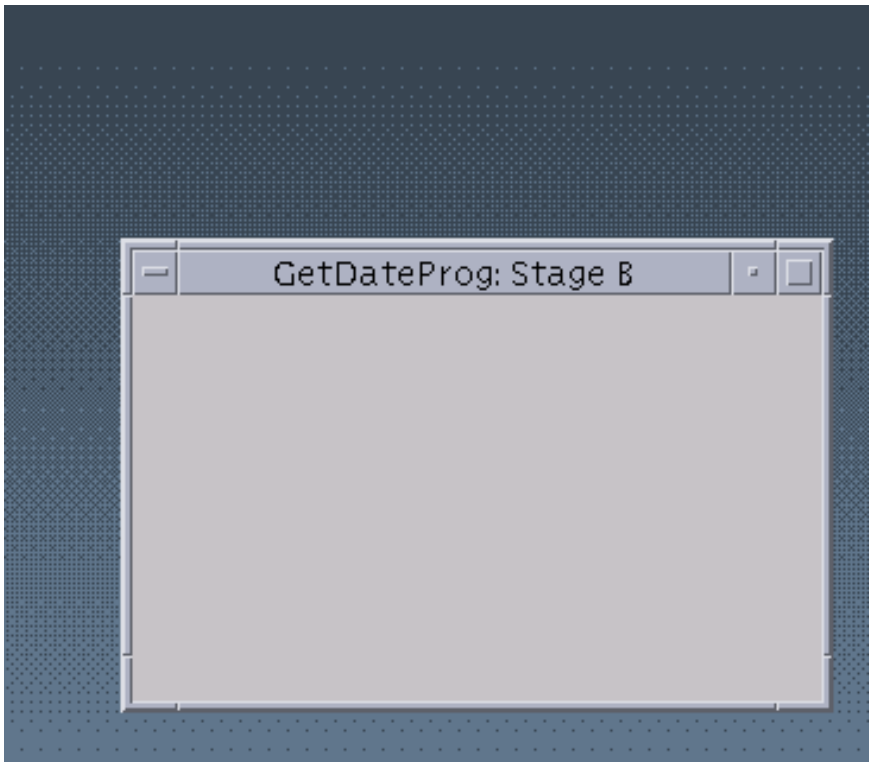
```
tJFrame.setLocation(50, 100);
```

says that you want the top left-hand corner of the window to be positioned 50 *pixel*s from the left-hand side of the screen and 100 pixels down from the top of the screen. And the call:

```
tJFrame.setSize(300, 200);
```

says that you want the window to be 300 pixels wide and 200 pixels high.

When this program is executed, it just displays a blank window on the screen. The result of executing this program is shown here:



The program has no code to understand the removal of the window: so if you want to stop the execution of this program, you will need to press **Ctrl/C** in the window in which you typed the command:

```
java GetDateProg
```

2.6 Stage C: adding GUI components to the window

Some GUI components will now be put into the window that is displayed by the program. As with the previous program, the first step is to create an object to represent that window:

```
JFrame tJFrame = new JFrame("GetDateProg: Stage C");
```

In order to get our program to display a textfield and a button, the program needs to create these GUI components and add them to the *content pane* of the frame.

The Swing API contains classes that enable us to represent textfields and buttons:

```
JTextField tJTextField = new JTextField("hello", 35);
JButton tJButton = new JButton("Get Date");
```

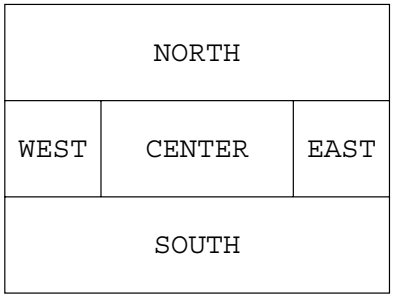
There are a number of constructors for these classes (as shown at *$API/javax/swing/JTextField.html* and *$API/javax/swing/JButton.html*). The ones used above create a textfield containing 35 columns which is initialized to the string "hello", and a button containing a label with the characters "Get Date". Once again, this just creates two objects within an executing Java program that represent a textfield and a button. It does not do anything with them, such as make them visible.

These GUI components need to be added to the content pane of the JFrame window. We can get a reference to the JFrame's content pane by executing the method getContentPane:

```
Container tContentPane = tJFrame.getContentPane();
```

The actual way in which GUI components are displayed within a container such as this content pane is controlled by a *layout manager*. The default layout manager for a content pane is a layout known as BorderLayout.

The BorderLayout layout manager allows you to use a method called add to place components in five divisions of the content page appropriately known as *NORTH*, *WEST*, *CENTER*, *EAST* and *SOUTH*. These divisions are illustrated by this diagram:



You do not have to put a component in each division: the layout manager will arrange the spacing of the components that you do provide:

```
tContentPane.add(tJTextField, BorderLayout.NORTH);
tContentPane.add(tJButton, BorderLayout.SOUTH);
```

The class java.awt.BorderLayout conveniently provides constants named NORTH, WEST, CENTER, EAST and SOUTH.

If you are unhappy with the layout, you can either use Container's setLayout method to choose another layout manager or you can use an object of class Box or JPanel to group items together. Both of these classes are in the javax.swing package: the Box class uses a layout called BoxLayout, and the JPanel class uses a layout called FlowLayout.

When you have added all of the components to the content pane, you should apply the method pack (from the class java.awt.Window) to the frame. This arranges for the size of the frame to be just big enough to accommodate the components. So this time there is no call of setSize: instead the call of pack determines an appropriate size for the window. A call of pack often appears just before a call of setVisible:

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

Here is the complete program:

```
26: // Stage C: adding GUI components to the window.        // GetDateProg.java
27: // Barry Cornelius, 22nd November 1999
28: import java.awt.    BorderLayout;
29: import java.awt.    Container;
30: import javax.swing. JButton;
31: import javax.swing. JFrame;
32: import javax.swing. JTextField;
33: public class GetDateProg
34: {
35:    public static void main(final String[] pArgs)
36:    {
37:       final JFrame tJFrame = new JFrame("GetDateProg: Stage C");
38:       final JTextField tJTextField = new JTextField("hello", 35);
39:       final JButton tJButton = new JButton("Get Date");
40:       final Container tContentPane = tJFrame.getContentPane();
41:       tContentPane.add(tJTextField, BorderLayout.NORTH);
42:       tContentPane.add(tJButton, BorderLayout.SOUTH);
43:       tJFrame.pack();
44:       tJFrame.setVisible(true);
45:    }
46: }
```

What gets displayed when this program is executed is shown below. As this time there is no call of setLocation, the window will appear in the top left-hand corner of the screen.

2.7 Stage D: responding to a click of the button

Having arranged for the textfield and the button to appear in the window, we need to be able to react to the user clicking the button. As was mentioned earlier, handling *event*s such as mouse clicks, mouse movements, key presses, window iconising, window removal, ... , is an area in which the Java Platform was improved between JDK 1.0 and JDK 1.1. Here we will look at how events are handled in versions of the Java platform from JDK 1.1 onwards.

In order to handle the event of a user clicking on the JButton component, you need to do two things:

- create an object that has an actionPerformed method containing the code that you want to be executed (when the user clicks on the JButton component);
- indicate that this object is responsible for handling any events associated with the JButton component.

To put this a little more formally:

1. the program needs to create an object that is of a class that implements the ActionListener *interface* (which is defined in the package java.awt.event);

2 the program needs to use the addActionListener method to register this object as the *listener* for events on the JButton component.

If you look at the WWW page *$API/java/awt/event/ActionListener.html*, you will see that in order to implement the java.awt.event.ActionListener interface you just need to have a class that declares one method, a method called actionPerformed that has the header:

```
public void actionPerformed(ActionEvent pActionEvent)
```

So, here is a class called JButtonListener that implements this interface:

```
47: // Stage D: a class whose actionPerformed method.    // JButtonListener.java
48: // writes to standard output.
49: // Barry Cornelius, 22nd November 1999
50: import java.awt.event. ActionEvent;
51: import java.awt.event. ActionListener;
52: import java.util.      Date;
53: public class JButtonListener implements ActionListener
54: {
55:    public JButtonListener()
56:    {
57:    }
58:    public void actionPerformed(final ActionEvent pActionEvent)
59:    {
60:       final Date tDate = new Date();
61:       System.out.println(tDate);
62:    }
63: }
64:
```

The GetDateProg program can create an object of this class in the usual way:

```
JButtonListener tJButtonListener = new JButtonListener();
```

That satisfies the first requirement given above.

The program also needs to say that this object is going to be responsible for handling the clicks on the button. What we are effectively wanting to do is to say: `please execute this object's actionPerformed method whenever there is a click on the JButton component`. In order to do this, we need to associate the object that has the actionPerformed method with the JButton object; or, in the jargon of Java, our JButtonListener object needs to be added as a *listener* for any events associated with the JButton object. This can be done using:

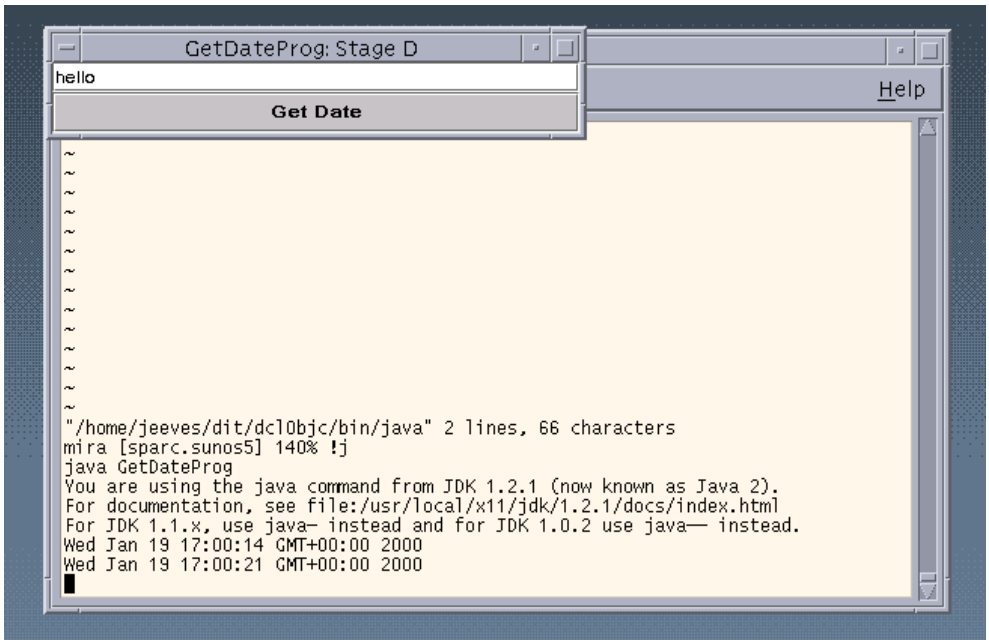
```
tJButton.addActionListener(tJButtonListener);
```

Because the addActionListener method has been applied to tJButton, the actionPerformed method of the object passed as an argument to addActionListener (i.e., tJButtonListener) will be executed at each click of this JButton component.

Here is the code of this version of the GetDateProg program:

```
65: // Stage D: responding to a click of a button.          // GetDateProg.java
66: // Barry Cornelius, 22nd November 1999
67: import java.awt.    BorderLayout;
68: import java.awt.    Container;
69: import javax.swing. JButton;
70: import javax.swing. JFrame;
71: import javax.swing. JTextField;
72: public class GetDateProg
73: {
74:    public static void main(final String[] pArgs)
75:    {
76:       final JFrame tJFrame = new JFrame("GetDateProg: Stage D");
77:       final JTextField tJTextField = new JTextField("hello", 35);
78:       final JButton tJButton = new JButton("Get Date");
79:       final JButtonListener tJButtonListener = new JButtonListener();
80:       tJButton.addActionListener(tJButtonListener);
81:       final Container tContentPane = tJFrame.getContentPane();
82:       tContentPane.add(tJTextField, BorderLayout.NORTH);
83:       tContentPane.add(tJButton, BorderLayout.SOUTH);
84:       tJFrame.pack();
85:       tJFrame.setVisible(true);
86:    }
87: }
```

An example of what happens when the JButton component is clicked is shown here:



Note that we do not have any precise control over when the actionPerformed method is called: this is at the whim of the person using the program. The act of registering code that will be executed later is sometimes referred to as creating a *callback*.

2.8 Stage E: altering the JTextField component

Although the program of Stage D outputs the current date and time whenever the JButton component is clicked, the program sends this output to the *standard output*, i.e., to the terminal window that runs the program. What we really want to do is to copy the date and time into the JTextField component. And it is the variable tJTextField of the main method of GetDateProg that points to the JTextField object that we want to be updated each time the user clicks on the button.

How can we refer to this JTextField object within the actionPerformed method? We cannot just use tJTextField as this variable is local to the main method, and, anyway, the main method is in a different class from the actionPerformed method.

The easiest way is to alter the *constructor* for the listener object so that tJTextField is passed as an argument:

```
JButtonListener tJButtonListener = new JButtonListener(tJTextField);
```

In this way, when the JButtonListener object is being created, the constructor knows which JTextField object we want to be altered: it is the one pointed to by tJTextField.

What can the constructor do with this information? Well, it can make its own copy of the pointer:

```
public JButtonListener(JTextField pJTextField)
{
  iJTextField = pJTextField;
}
```

where iJTextField is a private field of the JButtonListener object.

So when the JButtonListener object is created, it stores a pointer to the JTextField object in a field of the JButtonListener object. Whenever the actionPerformed method is executed, it just has to alter the contents of the object pointed to by iJTextField.

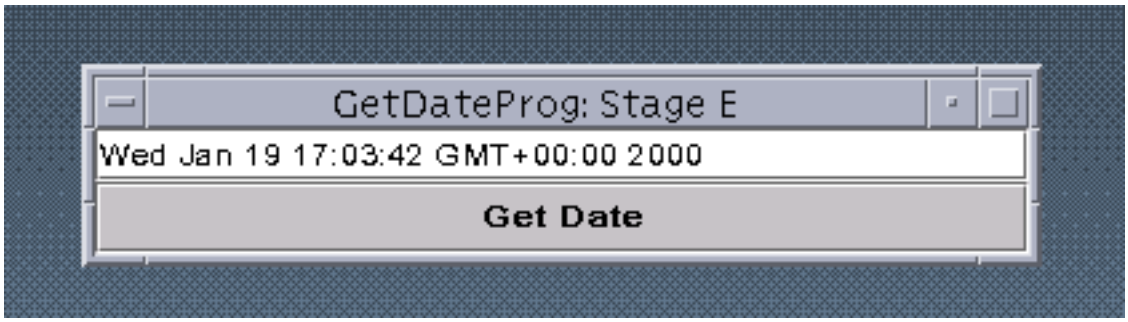
In order to change the value of a JTextField object, we need to apply a method called setText to the object, passing the appropriate string as an argument. Since we actually want to set the textfield to a string describing the current date and time, we need to do:

```
Date tDate = new Date();
iJTextField.setText("" + tDate);
```

Here is the complete text of this new version of the JButtonListener class:

```
 88: //                                          // JButtonListener.java
 89: // Stage E: implementing the ActionListener interface.
 90: // Barry Cornelius, 22nd November 1999
 91: import java.awt.event. ActionEvent;
 92: import java.awt.event. ActionListener;
 93: import java.util.      Date;
 94: import javax.swing.    JTextField;
 95: public class JButtonListener implements ActionListener
 96: {
 97:    private JTextField iJTextField;
 98:    public JButtonListener(final JTextField pJTextField)
 99:    {
100:       iJTextField = pJTextField;
101:    }
102:    public void actionPerformed(final ActionEvent pActionEvent)
103:    {
104:       final Date tDate = new Date();
105:       iJTextField.setText("" + tDate);
106:    }
107: }
108:
```

The GetDateProg program for this stage is the same as that used for Stage D. An example of what happens when the JButton component is clicked is shown here.:



2.9 Stage F: closing the window

Although this has achieved our goal of altering the JTextField component whenever the JButton component is clicked, there is one other thing that we ought to do. Up until now, the only way in which we have been able to terminate the execution of the program has been to press **Ctrl/C**. With this example, it may be useful to terminate the execution when the user closes the window.

In the same way that an ActionListener object is created to handle clicks on the JButton, we can establish an object which is responsible for handling events on a window. Unlike the ActionListener interface where we only had to provide one method, the WindowListener interface requires us to provide seven methods to provide for seven events concerning the manipulation of windows. The details are given on java.awt.event.WindowListener's WWW page which is at *$API/java/awt/event/WindowListener.html*.

Here is a class that implements the WindowListener interface:

```
109: //                                      // ExitOnWindowClosing.java
110: // Stage F: implementing the WindowListener interface.
111: // Barry Cornelius, 22nd November 1999
112: import java.awt.       Window;
113: import java.awt.event. WindowEvent;
114: import java.awt.event. WindowListener;
115: public class ExitOnWindowClosing implements WindowListener
116: {
117:    public void windowActivated(final WindowEvent pWindowEvent)
118:    {
119:    }
120:    public void windowClosed(final WindowEvent pWindowEvent)
121:    {
122:    }
123:    public void windowClosing(final WindowEvent pWindowEvent)
124:    {
125:       final Window tWindow = pWindowEvent.getWindow();
126:       tWindow.setVisible(false);
127:       tWindow.dispose();
128:       System.exit(0);
129:    }
130:    public void windowDeactivated(final WindowEvent pWindowEvent)
131:    {
132:    }
133:    public void windowDeiconified(final WindowEvent pWindowEvent)
134:    {
135:    }
136:    public void windowIconified(final WindowEvent pWindowEvent)
137:    {
138:    }
139:    public void windowOpened(final WindowEvent pWindowEvent)
140:    {
141:    }
142: }
```

Note that this class has the *implements clause* implements WindowListener and has method declarations for each of the seven methods.

Because we only want to do something special when a window is about to close, some code has been provided for the windowClosing method whereas the other six method declarations have empty blocks.

When using the ActionListener interface, we had to:

- provide an object of a class that implements the ActionListener interface;
- register this object as a *listener* for clicks on the JButton component.

We have to do similar things when using the WindowListener interface.

Consider this version of the GetDateProg program:

```
143: //                                          // GetDateProg.java
144: // Stage F: using a WindowListener to handle a window-closing event.
145: // Barry Cornelius, 22nd November 1999
146: import java.awt.    BorderLayout;
147: import java.awt.    Container;
148: import javax.swing. JButton;
149: import javax.swing. JFrame;
150: import javax.swing. JTextField;
151: public class GetDateProg
152: {
153:    public static void main(final String[] pArgs)
154:    {
155:       final JFrame tJFrame = new JFrame("GetDateProg: Stage F");
156:       final JTextField tJTextField = new JTextField("hello", 35);
157:       final JButton tJButton = new JButton("Get Date");
158:       final JButtonListener tJButtonListener = new JButtonListener(tJTextField);
159:       tJButton.addActionListener(tJButtonListener);
160:       final Container tContentPane = tJFrame.getContentPane();
161:       tContentPane.add(tJTextField, BorderLayout.NORTH);
162:       tContentPane.add(tJButton, BorderLayout.SOUTH);
163:       final ExitOnWindowClosing tExitOnWindowClosing =
164:                                          new ExitOnWindowClosing();
165:       tJFrame.addWindowListener(tExitOnWindowClosing);
166:       tJFrame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
167:       tJFrame.pack();
168:       tJFrame.setVisible(true);
169:    }
170: }
```

This program includes code that creates an ExitOnWindowClosing object:

```
ExitOnWindowClosing tExitOnWindowClosing = new ExitOnWindowClosing();
```

and registers this object as a listener for window events on the window associated with the tJFrame object:

```
tJFrame.addWindowListener(tExitOnWindowClosing);
```

The following statement (from the GetDateProg program) ensures that, when the user clicks on the window's close button, the window is not removed from the screen:

```
tJFrame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

This is to ensure that the program retains control: the window will only be removed when the program executes setVisible with an argument of false.

Now when the user manipulates the window, one of the methods of the ExitOnWindowClosing class will get executed. Most of these do not do anything. However, when the user clicks on the button to close the window, the code of the following method declaration gets executed:

```
public void windowClosing(final WindowEvent pWindowEvent)
{
  final Window tWindow = pWindowEvent.getWindow();
  tWindow.setVisible(false);
  tWindow.dispose();
  System.exit(0);
}
```

You can see that when the windowClosing method is called, some object (that is of the class WindowEvent) is passed as an argument to windowClosing. This object contains details of what caused the window event to take place. The WindowEvent class has various methods that can be applied to this WindowEvent object: one of these is called getWindow. So, the first statement of the windowClosing method makes tWindow point to the Window object associated with the window being closed. For GetDateProg, this is the JFrame object that was created by its main method.

A reference variable (that is of a class type) can point to an object of its class or an object of any subclass. Here, tWindow, a reference variable of the class type java.awt.Window is pointing to an object of the class javax.swing.JFrame, one of the subclasses of java.awt.Window.

The next statement of the windowClosing method calls the setVisible method. The call of setVisible with argument false ensures that the window is no longer displayed on the screen. When a Window object (such as a JFrame object) is created, some other objects are created. The call of dispose indicates that the space for these objects is no longer required. Although in general it is useful to do this, it is unnecessary in this program as the call of dispose is followed by the call of System.exit that terminates the program.

2.10 Conclusion

In this section, we have only seen a glimpse of what is available in the Swing API. You may also want to look at:

- JTextArea and JScrollPane which enable you to set up a scrollable multi-line area of text;
- Box which enables you to group GUI components together;
- JDialog which enables you to display a dialog box forcing the user to respond to a question;
- JDesktopPane and JInternalFrame which enable you to create windows inside a parent window;
- JMenuBar, JMenu and JMenuItem which enable you to add a menu system to a window.

3 The Collections API

3.1 An introduction to the Collections API

Besides having APIs for GUIs, another attraction of Java is that it has an API for representing collections of values. This API, which was new with the Java 2 Platform, is known as the *Collections API*.

The designers of the Collections API have decided that there are three main ways in which we will want to represent a collection of values:

- as a *list* — an ordered collection or sequence of values: there may be duplicates;
- as a *set* — a collection where each value appears only once: there are no duplicates;
- as a *map* — a collection where there is a mapping from keys to values: the keys are unique.

They have provided *interface*s called List, Set and Map that define the methods that can be applied to objects that are lists, sets and maps.

One of the benefits of using a List is that it allows duplicates, i.e., it allows the same value to appear more than once in the collection. This may be important. For example, if you are representing a collection of CDs, it may be that you have the same CD more than once. Or it may be that you are some kind of collector; perhaps you collect beer mats. In this case, you will often have duplicates because these allow you the possibility of swopping one of your duplicates with another collector.

The other benefit of using a List is that it allows values to be ordered in any way you like. For example, suppose you want to represent a mailbox as a collection of messages. The user might want to add a message to this mailbox at some particular position in the mailbox, or they might want to delete a particular message from the mailbox. Or, if you have a queue of people, you will want insertions to be made at the tail of the queue whereas deletions are to be made at the head of the queue. For both of these examples, a List could be used.

For the List interface, the API provides two classes that implement the interface. They are called ArrayList and LinkedList. The class ArrayList should be used if you want to make random accesses to the values of a collection, e.g., for a collection of messages in a mailbox, you will want to access each individual message: you might want to access the 5th message, then the 2nd, then the 7th, and so on. This is a typical situation in which ArrayList would be used.

The other class that implements the List interface is called LinkedList. This should be considered if you want to make frequent insertions and deletions from a list. If insertions/deletions dominate the activities that are performed on the List, then a LinkedList should be considered (instead of an ArrayList). In practice, it seems that the implementation of ArrayList is very good as it is often as fast or better than LinkedList in situations where. intuitively, LinkedList should be faster.

However, if your collection has no duplicates or you do not want such flexibility about ordering, you may want to consider representing the

collection using a Set. For the Set interface, a class called HashSet is provided. This gives a fast implementation of:

- adding new elements to a set;
- removing elements from a set;
- seeing whether a set contains a particular value.

Unlike the List, it is not possible to control the order in which values are stored in a Set. However, it may be that the values being added to a Set have a *natural order*. This is an ordering that is based on comparing the values of the collection. For example, for a set of strings, the collection may be ordered in alphabetical order; for a set of people, the collection may be ordered by alphabetical order of the name field of each person; and so on.

For this kind of collection, the designers of the Collections API have provided a *subinterface* of Set called SortedSet, and a class called TreeSet that implements the SortedSet interface. So, for an object that is of the TreeSet class, the method that iterates through the elements of the set produces the values in this natural order.

Finally, for some collections, a particular part of each value in the collection in some way identifies the value: it is called the *key*. The distinguishing feature of the Map interface is that it permits us to represent a mapping from keys to values. It could be used to represent a dictionary, a mapping from words to meanings. Or a database that, given a person's name, delivers the personal details of that person.

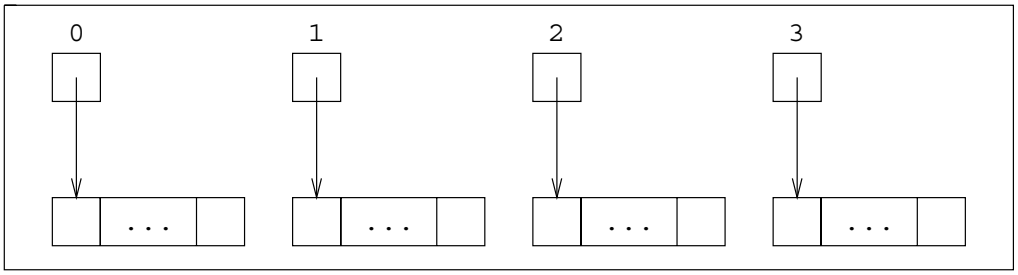
With the database, it may not be important for the values to be ordered: we may have no requirement to go through the thousands of people in the database in some order. Instead, we just want the values of the collection to be stored as efficiently as possible. For such a collection, the Collections API provides a class called HashMap (that implements the Map interface).

However, in the case of the dictionary, we may want the values of the collection to be sorted by the order of the words, as this will allow us easily to output the dictionary. There is a subinterface of the Map interface called SortedMap, and a class called TreeMap that implements this interface.

The preceding paragraphs summarize the overall design of a large part of the Collections API, and also briefly indicate the situations in which you might use the various interfaces and classes. In this Guide, we will just be looking at Lists.

3.2 The interface List and the classes ArrayList and LinkedList

A *list* is an ordered collection of *element*s, where each element contains a pointer to a value. You could visualize an object that is a list as:



A list of the methods that can be applied to an object that is a list is given at *$API/java/util/List.html*.

To begin with, we will just consider the following methods of the List interface:

- add — which adds a new element to a list;
- remove — which removes an element from a list;
- get — which returns a pointer to the value that is at a particular position in the list;
- set — which replaces the value of the element that is at a particular position in the list;
- size — which returns how many elements there are in the list.

We will now look at an example that shows how these methods can be used.

In the example, we will use the class Person that was produced in ITS *Guide 58: Getting started with Java*. Suppose we have three people:

```
Person tTom   = new Person("Tom%1.6%1981-12-25");
Person tDick  = new Person("Dick%1.7%1980-3-18");
Person tHarry = new Person("Harry%1.8%1979-8-4");
```

Suppose we want to create a list containing these three people. List is an interface, and (as was mentioned earlier) the Collections API provides two classes that implement this interface: they are ArrayList and LinkedList. So a list can be created using either:

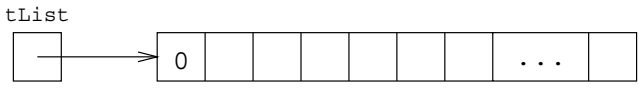
```
List tList = new ArrayList();
```

or:

```
List tList = new LinkedList();
```

Both of these statements create an *empty list*.

Suppose we use an ArrayList. We can visualize the ArrayList's empty list as follows:



Having created the empty list, the statements:

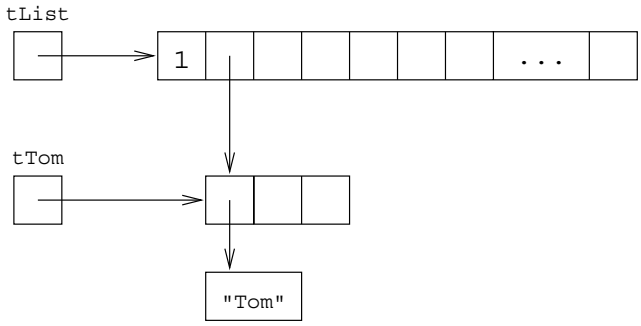
```
int tSize = tList.size();
boolean tIsEmpty = tList.isEmpty();
```

assign 0 to the variable tSize and the value true to tIsEmpty.

Suppose tTom points to a Person object. We can add the tTom object to the list using:

```
tList.add(tTom);
```

The result can be visualized as:



where only the first field of the Person object has been shown in detail.

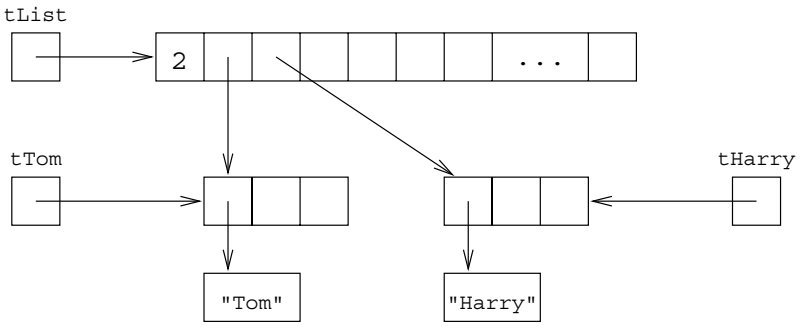
The designers of the Collections API have chosen to *share* the objects of a collection with *client*s of the collection. This means that the add method does not make its own copy of the object pointed to by tTom: it just establishes a new element of the list that points to the object that tTom is pointing to. So, the value of any element of the list that points to the tTom object will be affected if we later choose to change the value of the object pointed to by tTom.

If we now do:

```
tList.add(tHarry);    // TH
```

the list will contain two elements, the first one describing Tom, the second one describing Harry. The comment after the call of add, i.e., // TH, gives a cryptic indication of the state of the list after the method call has been executed.

So we now have the following situation:



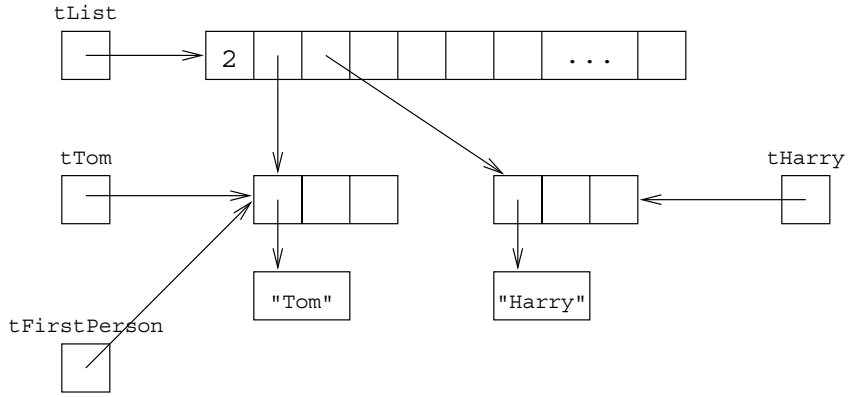
The add method has a parameter that is of the class Object. This means that the method can be used with an object of any class. It also means that the argument of the call does not have to be of the same class each time we call the method, and so we could build lists where the elements of the list do not have the same class.

The get method can be used to obtain the value of any element of the list. This method has one parameter: it indicates the position of the element in the list. The numbering of the elements starts from 0 (rather than from 1). So to get a pointer to the object at the first element of the list use:

```
Person tFirstPerson = (Person)tList.get(0);
```

Because a List can be used to store objects of any class, the result type of get is Object. So tList.get(0) returns a value of class Object and we have to *cast* this in order to treat the object as a Person object. If you cast to the wrong type, then (at execution time) the program will crash with a ClassCastException.

The above statement results in a situation that can be visualized as:



Note again that, because the Collections API adopts the *share approach*, tFirstPerson points to the same object as that of one of the elements of the list.

If we also execute the statements:

```
System.out.println(tFirstPerson);
System.out.println((Person)tList.get(1));
System.out.println(tList.size());
```

the following would be output:

```
Tom%1.6%1981-12-25
Harry%1.8%1979-08-04
2
```

When add is used with one argument as in the calls given earlier, the new element is added at the end of the list. Instead, we can use add with an additional argument that indicates a position:

```
tList.add(1, tDick);     // TDH
```

This means that tDick is to be inserted at position 1 and the element previously at position 1 is now at position 2.

Here are some other examples of calls of methods from the List interface:

```
tList.add(3, tHarry);    // TDHH
tList.add(0, tDick);     // DTDHH
tList.remove(tHarry);    // DTDH
tList.remove(1);         // DDH
tList.set(0, tTom);      // TDH
```

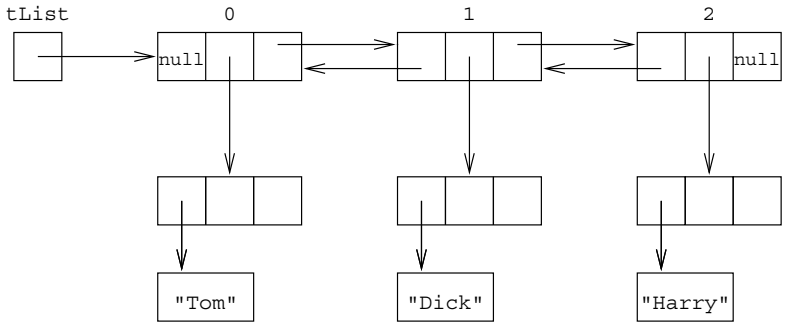
where the comments give a cryptic description of the state of the list after each statement has been executed.

3.3 Using the Iterator interface

We often want to do some task to each element of a list. This is known as *iterating* through the elements of the list. With a List, it is possible to do this using the following code:

```
for (int tPersonNumber = 0; tPersonNumber<tList.size(); tPersonNumber++)
{
  final Person tPerson = (Person)tList.get(tPersonNumber);
  iProcessPerson(tPerson);
}
```

where iProcessPerson is a method that contains the code that we want to execute on each element of the list. Although this would be reasonably efficient for a list that is implemented as an ArrayList, for a LinkedList it is very inefficient. This is because a LinkedList is implemented as a *doubly-linked list*:

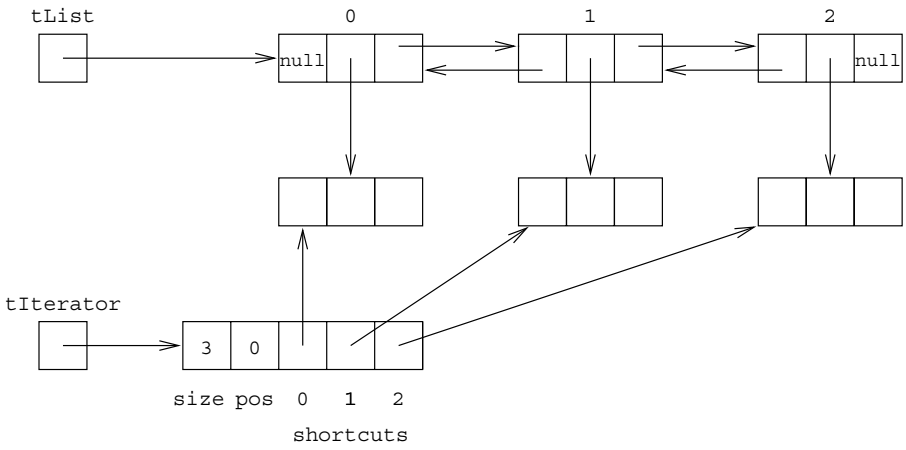


Consider what happens when the above code is used when tList points to a LinkedList object. When get is called, the code of get has to work its way down the list starting from the first element, and this has to be done on each of the calls of get.

So, we will avoid calling get in a loop. A different approach uses the iterator method that is defined in the List interface:

```
Iterator tIterator = tList.iterator();
```

No matter whether tList is pointing to an ArrayList object or a LinkedList object, the call of iterator will create information that enables the list to be iterated efficiently. Assuming tList is pointing to a LinkedList object, then, after the above statement has been executed, some sort of structure like the following will have been set up:



The call of iterator returns a pointer to an object which supports the Iterator interface. This interface has the methods documented at *$API/java/util/Iterator.html*.

The methods iterator, hasNext and next can be used as follows:

```
Iterator tIterator = tList.iterator();
while (tIterator.hasNext())
{
  final Person tPerson = (Person)tIterator.next();
  iProcessPerson(tPerson);
}
```

The methods hasNext and next can be efficiently implemented: a call of hasNext just returns the value of pos<size, and a call of next just returns the value of the posth element of the shortcuts and also increases the value of pos by 1.

3.4 The methods contains, indexOf, lastIndexof and remove

Given the method called iterator, it is very easy to find out whether a collection contains a particular value. Suppose we have a List which contains a collection of values all of which are Person objects. Suppose we now want to write a method called iIsInList that returns true if and only if the List object pList has an element which points to an object representing a person with the name pName:

```
private static boolean iIsInList(final List pList, final String pName)
{
  final Iterator tIterator = pList.iterator();
  while (tIterator.hasNext())
  {
    final Person tPerson = (Person)tIterator.next();
    if (pName.equals(tPerson.getName()))
    {
      return true;
    }
  }
  return false;
}
```

Here is an example of a call of this method:

```
boolean tFound = iIsInList(tList, "Dick");
```

However, it is a waste of time declaring this method as the List interface has a method called contains that does this job for us. So instead we can use:

```
Person tTargetPerson  = new Person("Dick%%");
boolean tFound = tList.contains(tTargetPerson);
```

Each of the methods:

```
public boolean    contains(Object pValue);
public int        indexOf(Object pValue);
public int     lastIndexOf(Object pValue);

public boolean      remove(Object pValue);
```

(of the List interface) requires the target list to be searched from the head (or the tail for lastIndexOf) of the list to find an element of the list that has the same value as the object pointed to by pValue. The WWW pages that document this interface (*$API/java/util/List.html*) state that each of these

methods looks for an element e such that pValue.equals(e). Because the parameter e is of type Object, the method being used here has the header:

```
public boolean equals(Object pObject);
```

Because pValue, the target of the equals, is actually of the class Person, and because the class declaration for Person declares a method with the above header then that method will be used when any of these four methods is executed.

So, when executing:

```
Person tTargetPerson  = new Person("Dick%%");
boolean tFound = tList.contains(tTargetPerson);
```

the contains method searches to see if it can find an element which equals that of tTargetPerson. It will use the method called equals declared in the class Person. This method says that two Person objects are equal if and only if the names are the same. For this to work, the class Person must provide a method with the header:

```
public boolean equals(Object pObject);
```

rather than (or in addition to):

```
public boolean equals(Person pPerson);
```

The method contains is not particularly useful if you want to do something to an element of the collection. Instead, it is better to use indexOf which will return the position of the element. Here is an example:

```
final Person tTargetPerson  = new Person("Dick%%");
final int tPosition = tList.indexOf(tTargetPerson);
if (tPosition=0)
{
  final Person tPerson = (Person)tList.get(tPosition);
  tPerson.setName("Richard");
  ...
}
```

Because the Collections API uses the *share approach*, tPerson is pointing to the same object that an element of tList is pointing to. So, the statement:

```
tPerson.setName("Richard");
```

also changes one of the elements of tList (which may or may not be what you want). If you prefer not to alter the element of the list, the result of the call of get should be cloned:

```
final Person tPerson = new Person((Person)tList.get(tPosition));
tPerson.setName("Richard");
...
```

It is also possible to use indexOf when you want to remove an element from a list: first find the appropriate position in the list and then remove the element at that position:

```
Person tTargetPerson  = new Person("Dick%%");
int tPosition = tList.indexOf(tTargetPerson);
if (tPosition=0)
{
  tList.remove(tPosition);
}
```

However, the List interface has another remove method which is more suitable (as it eliminates the need to call indexOf). So, the above is better coded as:

```
Person tTargetPerson  = new Person("Dick%%");
tList.remove(tTargetPerson);
```

3.5 An example of a complete program that manipulates a list

Here is a complete program that manipulates a list.

```
171: //                                          // ExamineList.java
172: // Read a list of people from a file, output the list, and then examine it.
173: // Barry Cornelius, 6th February 2000
174: import java.util. ArrayList;
175: import java.io.   BufferedReader;
176: import java.io.   FileReader;
177: import java.io.   InputStreamReader;
178: import java.io.   IOException;
179: import java.util. Iterator;
180: import java.util. List;
181: public class ExamineList
182: {
183:    public static void main(final String[] pArgs) throws IOException
184:    {
185:       if (pArgs.length!=1)
186:       {
187:          System.out.println("Usage: java ExamineList datafile");
188:          System.exit(1);
189:       }
190:       final List tList = new ArrayList();
191:       // read a list of people from a file
192:       final BufferedReader tInputHandle =
193:                   new BufferedReader(new FileReader(pArgs[0]));
194:       while (true)
195:       {
196:          final String tFileLine = tInputHandle.readLine();
197:          if (tFileLine==null)
198:          {
199:             break;
200:          }
201:          final Person tFilePerson = new Person(tFileLine);
202:          tList.add(tFilePerson);
203:       }
204:       // output the list that has been read in
205:       final Iterator tIterator = tList.iterator();
206:       while (tIterator.hasNext())
207:       {
208:          final Person tIteratePerson = (Person)tIterator.next();
209:          System.out.println(tIteratePerson);
210:       }
211:       // ask the user to examine the list
212:       final BufferedReader tKeyboard =
213:                   new BufferedReader(new InputStreamReader(System.in));
214:       while (true)
215:       {
216:          System.out.print("Person? ");
217:          System.out.flush();
218:          final String tKeyboardLine = tKeyboard.readLine();
219:          if (tKeyboardLine.equals(""))
220:          {
221:             break;
222:          }
223:          final Person tTargetPerson = new Person(tKeyboardLine);
224:          System.out.print(tTargetPerson);
225:          final int tPosition = tList.indexOf(tTargetPerson);
226:          if (tPosition=0)
227:          {
228:             System.out.println(" is at position " + tPosition);
229:          }
230:          else
231:          {
232:             System.out.println(" is absent");
233:          }
234:       }
235:    }
236: }
```

The program begins by obtaining the name of a file from the command line. It then reads lines from this file, each line containing the details about one person. As it reads each line, it creates a Person object and adds this object to a list. Having read the file, the program uses an Iterator to output the contents of the list. Finally, the program keeps reading lines from the keyboard (each line containing the details of a person) and finding out whether the person is in the list. It keeps doing this until the user of the program types in an empty line.

3.6 Conclusion

As mentioned earlier, besides providing Lists, the Collections API also provides interfaces and classes for Sets and Maps. These are used in a similar way to those for Lists.

4 Writing applets (for use with the WWW)

4.1 Using HTML to code WWW pages

When you use a *WWW browser* (such as Netscape's *Navigator* or Microsoft's *Internet Explorer*) to display a WWW page, you will see a combination of paragraphs of text, bulletted lists of information, tables of information, images, links to other pages, and so on. The people that have prepared WWW pages have coded them (or have arranged for them to be coded) using *HTML* (*HyperText Markup Language*).

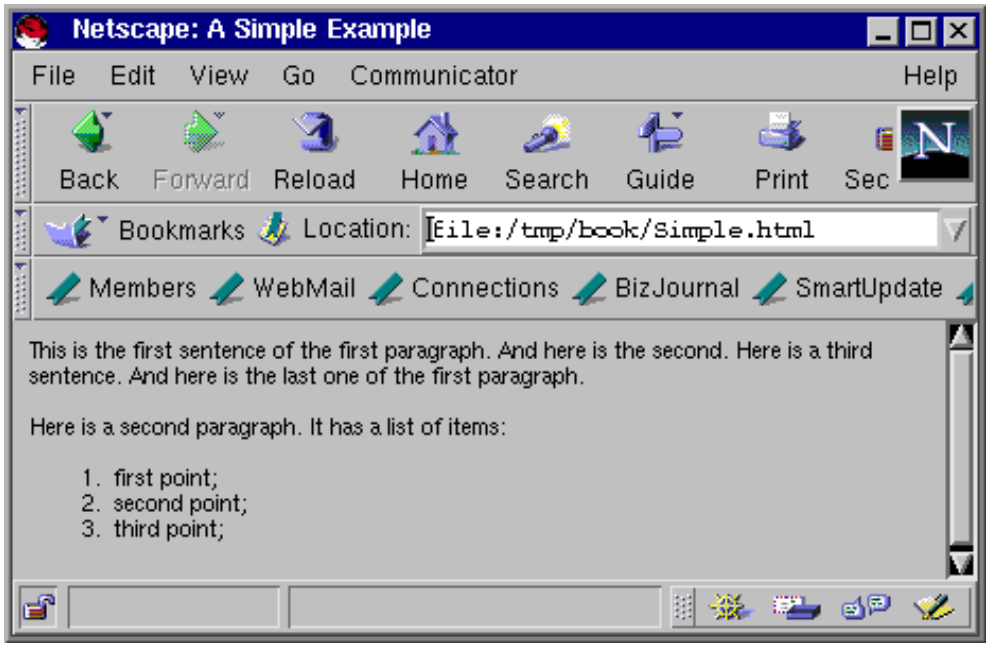
Here is an example of some HTML:

```
237: <HTML>
238: <HEAD>
239: <TITLE>A Simple Example</TITLE>
240: </HEAD>
241: <BODY>
242: <P>
243: This is the first sentence of the first paragraph.  And here is the
244: second.
245: Here is a third sentence.  And
246: here is the last one of the first paragraph.
247: </P>
248: <P>
249: Here is a second paragraph.
250: It has a list of items:
251: <OL>
252: <LI>first point;</LI>
253: <LI>second point;</LI>
254: <LI>third point;</LI>
255: </OL>
256: </P>
257: </BODY>
258: </HTML>
```

We will suppose that this text has been stored in a file called Simple.html. The HTML language involves the use of *tag*s which usually occur in pairs. An example is <P> and </P> which are used to indicate that the embedded text should be displayed by the WWW browser as a paragraph.

When someone (perhaps on the other side of the world) uses a browser to visit a WWW page, the HTML instructions are transferred across the Internet to the browser; the browser interprets these instructions and then displays something within the browser's window. The HTML given in the file

Simple.html would cause a browser to display something like that shown here.



4.2 Getting Java bytecodes executed when a WWW page is visited

Since the inception of the WWW in the early 1990s, people have been finding different ways of making a WWW page more appealing to the visitor to the page. When Sun first produced Java in 1995, they thought it would be useful if Java code could be executed as part of browsing a WWW page. The Java code could do some processing and display its output within the pane of the WWW browser. They showed that this was possible by producing a WWW browser that had this capability — it was first called *WebRunner* and later called *HotJava*. They then persuaded Netscape whose browser (Navigator) was the most popular at that time to include a Java interpreter as part of the code of Navigator. Support for Java within Microsoft's Internet Explorer came later.

In order that the author of a WWW page could indicate which Java .class file was to be executed when the WWW page was loaded, HTML was altered to include an APPLET tag. Here is an example of some HTML that includes an APPLET tag. Suppose that this text is stored in the file HelloApplet.html.

```
259: <HTML>
260: <HEAD>
261: <TITLE>The HelloApplet Example</TITLE>
262: </HEAD>
263: <BODY>
264: <P>
265: Start.
266: </P>
267: <APPLET CODE="HelloApplet.class" WIDTH="150" HEIGHT="25">
268: <P>Java does not seem to be supported by your WWW browser</P>
269: </APPLET>
270: <P>
271: Finish.
272: </P>
273: </BODY>
274: </HTML>
```

WWW browsers ignore tags that they do not understand. So if a WWW browser is given this HTML and it does not understand the APPLET tag, it will display the message Java does not seem to be supported by your WWW browser. However, if a WWW browser is capable of running Java, then, when the HTML interpreter of the browser sees this APPLET tag, it will start to obtain the *bytecode*s from the file mentioned in the CODE attribute of the APPLET tag. So, with the HTML given in the file HelloApplet.html, it would download the bytecodes that are in the file HelloApplet.class. Unless you also include a CODEBASE attribute, the browser will assume that this file is in the same directory from which it is obtaining the file containing the HTML instructions.

These bytecodes will be transferred from the .class file into a storage area known to the Java interpreter of the WWW browser. Often the bytecodes of a .class file will take some time to be transferred and so the rest of the WWW page is likely to be displayed before they arrive. When the bytecodes have finally arrived, the browser's Java interpreter will execute them.

So, although the author of the WWW page compiled the Java source code on his/her computer, the .class file(s) that were produced by the compiler will be executed by the Java interpreter contained in a WWW browser that is running on the computer of the person visiting the WWW page.

4.3 Deriving from Applet instead of declaring a main method

So far the Java source code that we have produced has been for programs that we have run on our own computer. Such programs are called *Java application*s. We are now about to produce Java source code that is to be run by the Java interpreter of a WWW browser. This kind of source code is called a *Java applet*.

The source code for an application is different from that for an applet. For an application, we provide a class that has a method called main, and this is the method that is executed first when we run the java command, the command that executes the Java interpreter. For an applet, we do not

provide a main method: instead, we use *inheritance* to derive a class from java.applet.Applet and *override* methods like paint, init, start, stop and destroy. We do this because this is what the Java interpreter contained in the WWW browser expects.

Here is an example of some Java source code that is a Java applet:

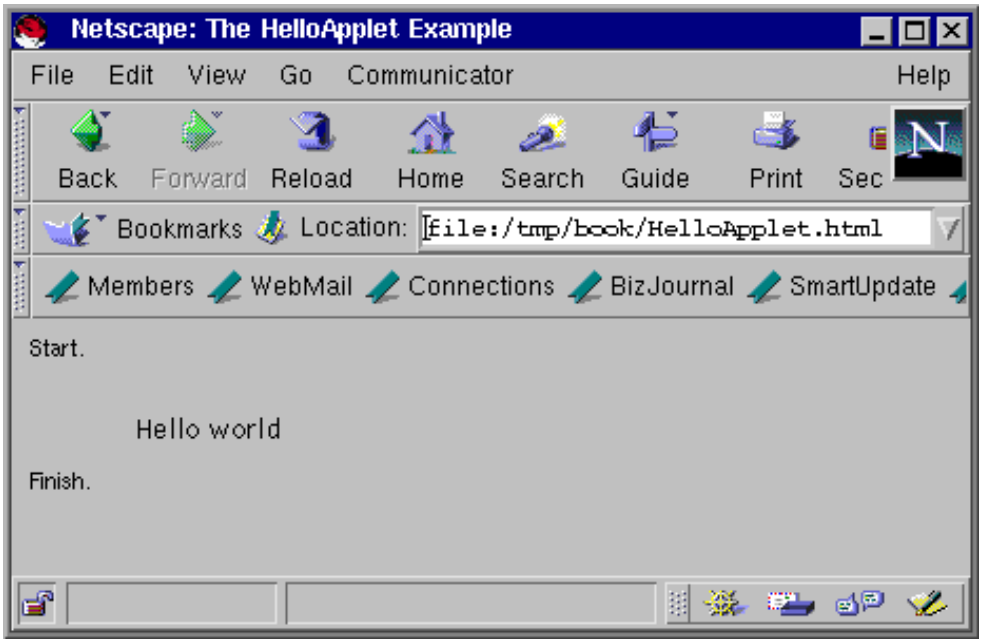
```
275: // The code of the HelloApplet applet paints a string.   // HelloApplet.java
276: // Barry Cornelius, 24th April 1999
277: import java.applet.Applet;
278: import java.awt.Graphics;
279: public class HelloApplet extends Applet
280: {
281:    public void paint(Graphics pGraphics)
282:    {
283:       pGraphics.drawString("Hello world", 50, 25);
284:    }
285: }
```

This can be compiled in the usual way:

```
javac HelloApplet.java
```

in order to produce the file HelloApplet.class.

Suppose we tell a WWW browser to read the WWW page that is in the file HelloApplet.html. When it reaches the APPLET tag, it knows it has to obtain the bytecodes contained in the file HelloApplet.class. When these bytecodes have arrived, the Java interpreter contained in the WWW browser will create an object of the HelloApplet class. And, because we have overridden the paint method, the code of this method will be executed. The result is displayed within the window of the browser (as shown here).



The code of HelloApplet is simple, and so the only classes that it depends on are classes from Java's Core APIs. However, normally the code for an applet will be dependent on other classes that the author has written. If this is the case, then, as the Java interpreter executes the bytecodes, it will detect that the bytecodes of other classes that need to be downloaded, and

so it will return to the author's WWW site to download the bytecodes from the appropriate .class files.

4.4 Dealing with the different versions of the Java platform

Early versions of WWW browsers contain a Java interpreter that understands *JDK 1.0.2*, the version of Java that was prevalent at the time they were released. Each time a new version of a WWW browser was released, the latest version of the Java interpreter was included in the browser. So the Java interpreter of some WWW browsers understand JDK 1.1.x (although, unfortunately, with many versions of WWW browsers, some parts of JDK 1.1.x are missing).

During the years 1996-1998, this led to a chaotic state of affairs: some browsers would only execute applets coded with JDK 1.0.2, and other browsers only understood parts of JDK 1.1. The best advice during this time was to write the Java source code for Java applets in terms of the language and the APIs of JDK 1.0.2, and to compile the source code with the JDK 1.0.2 compiler.

Of course, such an approach does not mean that you can reap the benefits of later versions of the Java Platform. For example, the way of handling events (such as the event of a user clicking a button) was improved between JDK 1.0 and JDK 1.1. And the Java 2 Platform brought the release of the Swing and Collection APIs.

The Java 2 Platform equivalent of the HelloApplet applet is the HelloJApplet applet that is given here:

```
286: // A JDK 1.2 version of the HelloApplet applet.        // HelloJApplet.java
287: // Barry Cornelius, 24th April 1999
288: import java.awt.Graphics;
289: import javax.swing.JApplet;
290: public class HelloJApplet extends JApplet
291: {
292:    public void paint(final Graphics pGraphics)
293:    {
294:       pGraphics.drawString("Hello world", 50, 25);
295:    }
296: }
```

This class is derived from the JApplet class (from javax.swing), a class that is itself derived from java.applet.Applet. But, because it uses the features that were new with the Java 2 Platform, how can we run this applet?

Recognizing that browsers supporting different versions of Java interpreters was a major problem, Sun looked at how this problem might be overcome. They decided that it would be more flexible to provide a *plug-in* containing a Java interpreter. (A *plug-in* is an additional piece of software that a browser can be configured to use. The advantage of using a plug-in is that a user can update it without having to update the browser.)

This plug-in is known as the *Java Plug-in*. (Note: it was previously called the *Java Activator*.)

The idea is that the applet is executed using the Java interpreter contained in this plug-in, and any Java interpreter contained in the WWW browser will be ignored. But, how can you arrange for your browser to use the plug-in's Java interpreter rather than the browser's Java interpreter?

With early versions of the Java Plug-in, Sun suggested that developers of WWW pages that use Java applets code their HTML in such a way that the visitor to the WWW page is asked to download the plug-in to their computer if the plug-in appropriate to the version of Java required by the applet is not present on the computer.

Unfortunately, the way in which the HTML is written in order for this to happen depends on what browser is being used. For example, the HTML that is required for Netscape's Navigator is different from that that is needed for Microsoft's Internet Explorer. Although you could provide HTML that only works for one of these browsers, it is better for your HTML to allow any browser. So the reasonably simple APPLET tag of the file HelloApplet.html needs to be replaced by the HTML of the file HelloJApplet.html which is shown on the next page.

This file contains a large number of difficult lines of HTML in order for it to work with both Navigator and Internet Explorer. Essentially, the lines immediately following the OBJECT tag are used by Internet Explorer, whereas those immediately following the EMBED tag are used by Navigator. There are details about what all this means at *http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/using_tags.html*

On that WWW page, Sun give an even more complicated version that can deal with situations not catered for by the HTML given on the next page. Even though it is complicated, once you have got it right, the only parts that need to be changed are the two sets of references to the name of the .class file (HelloJApplet.class), the width (150) and the height (25).

With later versions of the Java Plug-in, when the user is installing the plug-in, they can arrange for the plug-in's interpreter to be used when a WWW page is coded using an APPLET tag.

Obviously, you have no control over the users visiting your WWW pages: you do not know whether their browser has the Java Plug-in installed, nor whether it is a recent version of the Java Plug-in, nor whether they have configured the latter to understand the APPLET tag.

At the current time, probably the best advice is as follows:

- If you are writing the applet for your own use, download the latest version of the Java Plug-in; configure it to be used when an APPLET tag is used; and code your WWW pages in terms of the APPLET tag.
- If you are providing your applet for others to use, do not use the APPLET tag: instead use the complicated HTML given overleaf.

```
297: <HTML>
298: <HEAD>
299: <TITLE>The HelloJApplet Example</TITLE>
300: </HEAD>
301: <BODY>
302: <P>
303: Start.
304: </P>
305: <OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
306:   width="150" height="25"
307:   codebase="http://java.sun.com/products/plugin/1.2/jinstall-12-win32.cab#Version=1,2,0,0">
308:   <PARAM NAME="code" VALUE="HelloJApplet.class">
309:   <PARAM NAME="type" VALUE="application/x-java-applet;version=1.2">
310:   <COMMENT>
311:   <EMBED type="application/x-java-applet;version=1.2"
312:     width="150" height="25"
313:     code="HelloJApplet.class"
314:     pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
315:    <NOEMBED>
316:      </COMMENT>
317:        No JDK 1.2 support which is what is needed for this applet
318:    </NOEMBED>
319:   </EMBED>
320: </OBJECT>
321: <P>
322: Finish.
323: </P>
324: </BODY>
325: </HTML>
```

4.5 Using appletviewer when developing Java applets

When developing a Java applet, the .class file will often be changed before the final version is produced. One problem that authors of Java applets often face is the difficulty in persuading a WWW browser to load a new version of a .class file if the .class file has changed on the author's WWW site. With some releases of some WWW browsers, pressing the *Shift* key at the same time as clicking the browser's *Reload* button may cause it to reload everything. If this does not work, then the only sure way to get round this problem is to exit from the WWW browser and to start it up again.

If you are developing a WWW applet, it will be very tedious if you have to restart the WWW browser frequently.

However, the SDK/JDK comes with a tool called *appletviewer* that can be used to view the output of an applet whose .class file is mentioned in a WWW page. So, having compiled some Java source code, e.g.:

```
javac HelloJApplet.java
```

the HelloJApplet.class file can be executed and its output can be displayed by running the following UNIX/MS-DOS command:

```
appletviewer HelloJApplet.html
```

You can keep this appletviewer program running. If you subsequently make a change to the HelloJApplet.java file and then recompile it, you can get appletviewer to load the new version of the HelloJApplet.class file by clicking on the *Reload* option of the appletviewer's menu. So this appletviewer program provides a useful tool for testing Java applets.

4.6 The lifecycle of a Java applet

The HelloJApplet applet just overrides the paint method. Most applets do something more involved than just painting. In order to write any code for an applet you need to be aware of the *lifecycle* of an applet, i.e., the various stages that an applet goes through from birth to death.

When the bytecodes of an applet are loaded (or reloaded), the init method of the applet is executed. Then the applet's start method is executed. This method is also re-executed when the user comes back to the WWW page associated with the applet after having visited another page. Whenever the user leaves this page, the stop method is executed. Finally, the destroy method is executed if the WWW browser has to unload the applet.

By default, the class java.applet.Applet defines methods for init, start, stop and destroy that do nothing, i.e., they have empty bodies. So, if you want to define some actions to take place at the various points in the lifecycle of an applet, you just need to override the appropriate methods.

Overriding the start and stop methods is important for applets which start a new *thread*.

4.7 Overriding the init method

All of the classes that create a window on the screen (i.e., JWindow, JFrame, JDialog, JInternalFrame and JApplet) have a *content pane*. This is the main area of the window, and, we saw earlier that a program can access the content pane of an object of one of these classes by executing its getContentPane method.

So, instead of overriding the paint method to output the string "Hello world" as is done by the HelloJApplet applet, we could instead add a JLabel containing this string to the applet's content pane. As we only want to execute the code to add the JLabel object to the content pane once, it is appropriate to put the call of add in an init method of an applet. Here is an applet that does this:

```
326: // An applet that adds a JLabel to its content pane.   // JLabelJApplet.java
327: // Barry Cornelius, 3rd May 1999
328: import java.awt.BorderLayout;
329: import java.awt.Container;
330: import javax.swing.JApplet;
331: import javax.swing.JLabel;
332: public class JLabelJApplet extends JApplet
333: {
334:    public void init()
335:    {
336:       final JLabel tJLabel = new JLabel("Hello world");
337:       final Container tContentPane = getContentPane();
338:       tContentPane.add(tJLabel, BorderLayout.CENTER);
339:    }
340: }
```

4.8 Restrictions imposed on Java applets

So far, programs have been able to read from files, to write to files, and to call methods (such as System.exit) that behind the scenes make *system call*s, i.e., calls to routines of the underlying operating system. Using some of the APIs that have not been considered, it is also possible to write Java source code that communicates with other computers. Although it is reasonable for these sort of activities to be performed by Java source code that is a program, i.e., a *Java application*, is it appropriate for these activities to be performed by *Java applet*s?

To be more specific: if you visit a WWW page, and the author of that WWW page causes your WWW browser to execute some bytecodes produced by the author, are you happy for these bytecodes to write to files on your computer, or to read any of your files?

The designers of Java took the view that it is not necessarily appropriate for these activities to be performed by Java applets that have been downloaded from the Internet. So, the environment of an applet is controlled by the user of the WWW browser. For example, there is no access to local files from Netscape's Navigator, whereas HotJava users can configure which files can be read from and which can be written to. More details about these restrictions are given at *http://java.sun.com/sfaq/*

This approach is often called the *Sandbox* approach. This was Sun's first attempt at controlling what an applet can do. With later revisions of the Java Platform, Sun have been providing ways in which an applet can be allowed to perform these activities. It is now possible to add to an applet a *digital signature* authorized by a *certificate* obtained from a *certificate authority*. If you download this *signed applet* and you allow your WWW browser to accept its certificate, the applet is said to be a *trusted applet*. There are more details about how to execute signed applets at *http://java.sun.com/security/signExample12/*

Sun's main WWW page on security restrictions is *http://java.sun.com/security/*

4.9 Reworking an application as an applet: GetDateApplet

Many of the programs that you have already produced can easily be rewritten as Java applets. Often this can be done by putting the code of the main method into an applet's init method. For example, we could take the statements of the main method of the GetDateProg program (given earlier in *Stage F*) and put them into an init method of a GetDateApplet class. The resulting code is shown here:

```
341: //                                          // GetDateApplet.java
342: // An applet containing the button to get the date and time.
343: // Barry Cornelius, 22nd November 1999
344: import java.awt.    BorderLayout;
345: import java.awt.    Container;
346: import javax.swing. JApplet;
347: import javax.swing. JButton;
348: import javax.swing. JFrame;
349: import javax.swing. JTextField;
350: public class GetDateApplet extends JApplet
351: {
352:    public void init()
353:    {
354:       final JFrame tJFrame = new JFrame("GetDateApplet: Stage F");
355:       final JTextField tJTextField = new JTextField("hello", 35);
356:       final JButton tJButton = new JButton("Get Date");
357:       final JButtonListener tJButtonListener =
358:                                  new JButtonListener(tJTextField);
359:       tJButton.addActionListener(tJButtonListener);
360:       final Container tContentPane = tJFrame.getContentPane();
361:       tContentPane.add(tJTextField, BorderLayout.NORTH);
362:       tContentPane.add(tJButton, BorderLayout.SOUTH);
363:       tJFrame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
364:       tJFrame.pack();
365:       tJFrame.setVisible(true);
366:    }
367: }
```

In producing this class, the two statements of GetDateProg that establish a window listener:

```
final ExitOnWindowClosing tExitOnWindowClosing = new ExitOnWindowClosing();
tJFrame.addWindowListener(tExitOnWindowClosing);
```

have been omitted. This is because the ExitOnWindowClosing class has a windowClosing method that calls System.exit. For the reasons explained in the previous section, it is not appropriate for an applet to have this code.

When an applet is executed, if it creates any new windows then these will appear with a *yellow warning banner* displaying the text Warning: Applet Window. This text warns the user of the WWW browser that the window being displayed ha s not been produced by a *trusted applet*. The WWW page at *http://java.sun.com/products/plugin/plugin.faq.html* says that the `yellow warning banner is an important security feature. It cannot be disabled by untrusted applets. If you use a signed applet, where the signing key is trusted by the end user, then the warning banner will not be shown.`

4.10 Producing code that can be used either as an application or an applet

If you want to use some Java source code sometimes as an application and sometimes as an applet, it would be better not to have duplicate copies of the code as is the case with the main method of the GetDateProg class and the init method of the GetDateApplet class. It is usually easy to rewrite the code of the program and applet classes so as to avoid the duplication.

4.11 Using the Java archive tool

As was mentioned earlier, the CODE attribute in the HTML that is used to run an applet identifies the name of the file containing the bytecodes of the applet's class. However, normally, the author of an applet will provide a number of supporting classes as well as the class of the applet. It was pointed out earlier that the bytecodes of each of the .class files will be downloaded from the author's WWW site as the Java interpreter being used by the WWW browser detects that it requires them.

For example, suppose a directory contains the files for the GetDateApplet applet. The following files would have to be downloaded in order to execute this applet: GetDateApplet.class and JButtonListener.class. Obviously, programs are usually a lot more complicated than this: such programs may have a large number of .class files.

The Java 2 SDK (or the JDK) contains a tool that enables the author of an applet to combine a number of files into a single file. The resulting file is called a *Java Archive*. The tool is called jar, and, like the other commands of the SDK/JDK, it can be run from a UNIX/MS-DOS command line. The documentation for the jar command says `When the components of an applet or application (.class files, images and sounds) are combined into a single archive, they may be downloaded by a Java agent (like a browser) in a single HTTP transaction, rather than requiring a new connection for each piece. This dramatically improves download times. jar also compresses files and so further improves download time.`

Assuming that the directory containing the files for the GetDateApplet applet only contains .class files that are associated with this applet, then a Java Archive can be produced from these .class files by the UNIX/MS-DOS command line:

```
jar cvf GetDateApplet.jar *.class
```

The first argument to the jar command, which in this example is cvf, indicates the options that you want to be passed to the jar command. There are three main ways in which the jar command is used. If the options contain a c, then this means that you want to create an archive; if they contain a t, you want the jar command just to list the contents of an archive (that already exists); and if they contain an x, you want the command to extract some .class files from an archive.

If the options include the letter v, then the jar command will produce some output to tell you what it is doing — v means *verbose*. Finally, the f means that the name of an archive is given as the next argument. When a c option is present, the remaining arguments give the names of the .class files that you want to be put into the archive. In UNIX/MS-DOS, the notation *.class refers to all of the files in the current directory that have a .class extension.

So the above jar command produces a file called GetDateApplet.jar that contains a compressed archive of the two .class files that constitute the GetDateApplet applet.

Suppose a WWW page contains a CODE attribute to say that the bytecodes of an applet's class are stored in the file GetDateApplet.class. If you want the WWW browser to download the bytecodes in the Java Archive GetDateApplet.jar instead of downloading each .class file, you will need to include an ARCHIVE attribute as well as the CODE attribute.

If your HTML uses an APPLET tag or an EMBED tag, the syntax of the ARCHIVE attribute is:

```
archive="GetDateApplet.jar"
```

and, if your HTML uses an OBJECT tag, you need to include:

```
<PARAM NAME="archive" VALUE="GetDateApplet.jar">
```

Note you need a CODE attribute as well as the ARCHIVE attribute: the latter gives the name of the file containing the Java Archive (in which all the .class files are stored) and the CODE attribute gives the name of the class file that contains the class of the applet, i.e., effectively it identifies the bytecodes that are executed first.

As mentioned earlier, there are two advantages in using a Java Archive:

- One HTTP connection from the computer running the WWW browser to the computer of the author's WWW site is made to obtain the bytecodes in the Java Archive instead of making lots of HTTP connections, one for each of the .class files.
- Because the information in the Java Archive are stored in a compressed format, there is less bytes to be downloaded.

5 Other information about Java

ITS *Guide 58: Getting started with Java* can be used to find out about the basics of writing programs in Java. *Guide 58* also provides:

- the URLs of some WWW pages that form the primary resources for information about Java;
- details about some books that can be used to find out more information about the Java programming language.