palgrave

A Java Foundation Classes Primer

Fintan Culwin

A Java Foundation Classes Primer

A Java Foundation Classes Primer

FINTAN CULWIN

South Bank University

First published 2000 by
MACMILLAN PRESS LTD
Houndmills, Basingstoke, Hampshire RG21 6XS
and London
Companies and representatives
throughout the world

ISBN 0–333–77339–X paperback

A catalogue record for this book is available
from the British Library.

This book is printed on paper suitable for recycling and
made from fully managed and sustained forest sources.

10 9 8 7 6 5 4 3 2 1

09 08 07 06 05 04 03 02 01 00

Typeset by Ian Kingston Editorial Services, Nottingham, UK

Printed in Great Britain by Antony Rowe Ltd, Chippenham,
Wiltshire

Contents

Preface

The Java Foundation Classes (JFC), also known as the Swing classes (apparently because one of the engineers developing them liked swing music), provide a collection of components, also known as widgets, that can be used to construct Graphical User Interfaces (GUIs) using the Java Programming Language. The initial Java 1.0 release contained a collection of user interface components known as the Abstract Windowing Toolkit (AWT). The AWT was only ever intended to be an interim facility, as its components had a very plain and primitive appearance, many standard widgets were not supplied and, most importantly, it had a flawed event-handling mechanism. The flaws in the event-handling mechanism were corrected in the Java 1.1 release, but the other failings of the AWT persisted.

The Java 2.0 release (known during its development as the 1.2 release) includes the JFC, which is built on top of, and inherits many of the capabilities of, the AWT. It uses the same event-handling model as the 1.1 AWT release and contains a comprehensive collection of components which have a rich and acceptable appearance.

One of the most frightening books that I received recently was a 2500 page tome on the JFC which weighed in at approximately two and a half kilos. Its preface claimed that it contained a comprehensive introduction to all of the significant JFC classes. From the inspection that I made of the contents this is very probably correct, and the book has joined a number of similar volumes on my bookshelf. The JFC package, javax.swing, contains approximately 90 classes and 15 subsidiary packages; the entire collection of packages contains approximately 450 classes. The swing documentation, supplied with Sun's Java Development Kit (JDK), contains 1500 HTML pages and is approximately 27 megabytes in size. Given the size of the JFC it is obviously not possible for the 2500 page tome to claim to cover all of its classes and their facilities comprehensively. Likewise, a book of this size can only claim to cover a small fraction of the JFC.

When my daughters were younger they came home from school telling me that in science class they learned that an atom was like a small solar system with the nucleus in the middle, like the Sun, and the electrons orbiting it, like the planets. My response was not to give them a crash course in quantum mechanics leading to the Schrödinger wave equation and draft a letter to the school requesting that the science teacher be sacked for incompetence. The solar system model of an atom is not totally incorrect: it is within the limits of human comprehension (which quantum theory may not be) and has many properties which are in accord with reality.

In the same way, the model of the JFC which is presented in this book is not as sophisticated as the model used by its designers. Like the solar system model of an atom it is sufficiently correct for a more sophisticated model to be built from it, should it ever be necessary. However, to emphasize the internal model of the JFC, as so many other books have done, is to miss the point! The internal workings of the JFC are far less important than the techniques that are required to use them for the design and construction of interfaces that are truly usable. The most important aspect of a Graphical User Interface is that it will be used by humans in the performance of some task, and it is a legal requirement, in the EU and the USA, that, if it is to be used commercially, it must be effective, efficient and enjoyable.

The proportion of software project development costs associated with the user interface, as opposed to the core functionality, is estimated as between 20 and 80%; and the proportion seems to be growing. This emphasis upon the user interface has added an additional item to the litany of software engineering complaints. In addition to being late, over-budget and under-functional, most software artifacts are now known to have severe usability faults. In an attempt to address this issue directly I have, for a number of years and in a number of books, argued that usability must be placed at the center of the entire software development process. In order to achieve this the notation that is used to specify the interactive behavior of the software and the user, must also inform the software design and implementation.

For a long time I felt like a prophet in the wilderness in advocating the use of State Transition Diagrams (STDs) as a notation and technique for the engineering of software artifacts that have a GUI. Recently, a book entitled *Constructing the User Interface with Statecharts*, by Ian Horrocks, was published which also espouses and explains these ideas. Initially I was a little disappointed to receive the book as I had been toying with the idea of writing a similar book myself. However, upon reading the book I was pleased that I had not, as I could not have made as good a job of it as Ian. I recommend this book as a companion text to this one, the key difference being that, unlike this one, it does not focus upon implementation. Further details of the book are given in the Appendix.

In the light of the explanation above, the intention of this book is to provide a sufficiently detailed introduction to a selection of the more important JFC classes within the context of the systematic development of artifacts that have usability considerations as a central theme of their design. This process commences in Chapter 1, where the key ideas that permeate the rest of the book are explored in the context of the development of a simple *Stopwatch* artifact.

Almost all commercial quality software is now multi-threaded. The word processor that I am using to write this preface is responding quickly to each keypress, placing the associated character on the screen. Apparently at the same time a second thread is marking up the words that might be misspelled, a third is attempting to correct my grammar, a fourth is autosaving on a regular basis and others, for example one to print the document, could also be active. However, this consideration seems to have eluded almost all other authors of GUI books, and in order to start to rectify this the first, *Stopwatch*, example requires the use of two threads: one to service the time-keeping aspects and one to respond to the user's actions.

Chapter 2 presents a second example of the systematic production of artifacts that have a GUI by the development of a utility called *Fontviewer*, which allows a user to explore the fonts which are available on a particular workstation. This example is also noteworthy as it introduces the first specialized component. Although the JFC contains a large number of widgets, it cannot supply one that is totally suitable for every possible requirement, and it is often either desirable or necessary for an existing component to be extended to satisfy a particular need. The component used to display a font in the *Fontviewer* artifact starts to introduce this process, using a display-only example that does not have any interaction with the user.

After the introduction in depth in Chapters 1 and 2, Chapter 3 takes a broader view by examining in detail the attributes and other resources of one of the simplest JFC classes, the JLabel class. In order to accomplish this it develops an artifact, called *TabbedLabelDemo*, which allows the user to explore the classes' resources interactively. This requires a number of different panels to be presented, each of which contains a different interactive interface and so allows a large number of the simpler JFC classes to be introduced alongside the JLabel class.

Chapter 4 returns to the theme of producing specialized components by the development of a number of highly interactive *NumericInput* widgets. This theme is continued in Chapters 5 and 6 with the development of an artifact called the *MemoryGame*, played by turning over pairs of cards and removing them from the playing area if they match. This is the first example in the book which has a minimally realistic degree of complexity. It allows application-level main menu systems, pre-supplied and specialized dialogs and drag and drop capability to be introduced, while illustrating the scale of the task that would be involved in the production of a commercial quality application.

Chapter 7 further consolidates the production of highly interactive specialized components with the development of a component called *TimeInput*, affording the user the capability of entering a time of day by dragging the hands of a clock around a clock face. Although this component is valuable in its own right it is introduced so that the considerations of Usability Engineering, particularly the processes involved in usability investigations, can be described.

Chapters 8 and 9 continue the introduction to the JFC classes with the development of a primitive Web browser, known as a *Brewser*. This allows additional features of application-level main menus to be introduced, as well as the techniques required for text handling, multiple internal windows, loading, saving and printing to be demonstrated. Chapter 10 completes the theme of specialized components with the development of a *DynaLabel* component, which is produced with regard to JavaBean requirements. A component developed with regard to the JavaBean specification is known as a bean and is capable of easily interacting with other beans and also with bean-aware software development tools.

In Chapter 11 a *DynaLabel* is used as a constituent part of the *ColaMachine* artifact, a simulation of a soft drink vending machine. This artifact is introduced in order to illustrate internationalization and localization capability, with interfaces for different countries being automatically supplied. The final chapter, Chapter 12, concludes the tour of the JFC classes by introducing the components that supply tree and table views.

This was a difficult book to write, mostly because it was written while the JFC was being developed and each new beta release of the toolkit had a different selection of bugs (or were they features?). Peter Chalk at South Bank University again took it upon himself to comment upon each chapter as it was drafted, and in many cases redrafted and redrafted again. The 1999 cohort of students on a unit called UEEM suffered bravely and, sometimes unknowingly, supplied many very useful ideas. Jackie Harbor was more than willing to find space in her diary to come and eat a pizza at the Castello pizzeria restaurant and, incidentally, discuss the problems with the book. At home Maria, Leah and Seana again put up with a distracted maniac muttering under his breath about regression bugs and version control. Finally, this book might have been finished somewhat sooner, were it not for the quality of the Pride and the friendly insults offered by Joe and Colin at the George.

Fintan Culwin
`fintan@sbu.ac.uk`
`http://www.scism.sbu.ac.uk/fintan/`

1

An initial artifact – introducing event handling

1.1 Introduction

This book contains an introduction to the systematic development of Graphical User Interfaces (GUIs) using the Java Foundation Classes (JFC). (The Java foundation classes were developed in tandem with the Java 1.1.X releases and are an integral part of the Java 2.X releases.) The text does not attempt to introduce the Java language and it is assumed that the reader has some familiarity with Object-Oriented Design and Development (OODD), together with its practical expression in Java. Suitable resources to assist with obtaining this knowledge are suggested in Appendix A.

The development of GUIs which are *effective*, *enjoyable* and *efficient* is becoming increasingly important to the production of software artifacts, particularly those intended to be used within Web pages. They have to be *effective* in the sense that they allow the user to accomplish some task, *enjoyable* so that the user is likely to want to (or at least not be adverse to) using them in the future, and *efficient* in the sense that they should allow the task to be completed without undue stress or strain. The development of such interfaces can never be a trivial task, as it requires a high level of software engineering, ergonomic and graphical skills. A book of this size can only provide an initial introduction to the development of the required skills, and it makes no claim to be a complete description of Java, the JFC or usability engineering.

In this book, whenever a Java reserved word is used this **bold alternative** font will be used. The alternative font by itself will indicate the use of a Java Application Programming Interface (API) resource and an *italic alternative font* for developer-supplied Java resources. The *italicized normal font* will be used when a new technical term is introduced for the first time or when its meaning is significantly extended. This *italicized font* is also used when a design fragment is referred to which cannot be identified by a corresponding Java term. A similar convention is used in program listings, where **`this bold font`** indicates Java's reserved words, *`italic font`* indicates developer-supplied resources and `normal font` indicates Java API resources.

> Sidebar comments such as this are used to give technical information which reinforces or extends the information given in the main body of text. For a general reading the contents of these sidebars can be omitted, but if the book is being used to support development then the information contained may be vital.

Boxed comments such as this are used to provide design tips or other commentaries upon the topics currently being discussed in the text.

1.2 Components, containers and events

The essential philosophy of this book is to provide an exposition of practical design and development techniques using examples chosen so as to allow a selection of the most important and useful aspects of the JFC to be exposed. Accordingly, this initial chapter will commence with a description of the systematic design and development of a first, relatively trivial, example. However, before doing so some fundamental concepts concerned with the nature of, and the facilities supplied by, the Java Foundation Classes will have to be explained.

The Java Foundation Classes supply a large number of user interface *components*, instances of which can be assembled together to implement graphical user interfaces. All of the JFC classes are extended from a more primitive GUI library known as the Abstract Windowing Toolkit (AWT). Consequently, many attributes of JFC components are inherited from the AWT and hence some knowledge of the AWT is required to use the JFC. The JFC classes are known as the swing classes and are contained within a package called swing. The name *swing*, like the name *Java*, is not an acronym and has no other particular meaning.

One fundamental distinction established by the AWT is the distinction between a simple Component and components that are also Containers, i.e. descended from the AWT Container class. The AWT Container class introduces the facilities for a component to contain, and manage the positioning of, a number of other components which can themselves also be containers. All of the JFC classes are descended from the AWT Container class and hence inherit this capability, although some make no apparent use of the facility.

One other fundamental facility is introduced by the AWT event package and is essential to the development of GUIs. When a user interacts with a component on an interface, for example clicking upon a button, an Event is constructed and dispatched to any listeners which have been registered with the. This is the fundamental message-passing mechanism used within graphical user interfaces and hence is of primary importance for the effective design and implementation of GUIs. It is also possible, as will be demonstrated shortly, for non-GUI objects to make use of this event-handling mechanism to implement message passing.

Source code that makes use of the AWT components can be easily adapted to make use of the JFC components. For every AWT class, for example Button*, there is a corresponding class, for example* JButton*. By systematically preceding the name of every class, and likewise calls of the class constructors, with a ‘*J*’, the AWT components will be replaced with JFC components. There is no need to make any changes to the component’s method calls, as every AWT method is either inherited or overridden by the corresponding JFC component.*

The only other change required is where components are added to Applets *or* Frames*. Whereas with the AWT the component was added directly to the* Applet*, as in:*

```
applet.add( instanceChild)
```

> *In the JFC these components have a number of panes (as in window panes) and the child has to be added to its* contentPane*, as follows, to achieve the same effect.*
>
> ```
> applet.getContentPane().add( instanceChild)
> ```
>
> *The Java compiler will issue an error if an attempt is made to add a child component directly to a container when it should be added to a pane.*

1.3 The *Timer* class

To introduce the techniques that can be used in the construction of artifacts that have a graphical user interface, the rest of this chapter will describe the design and implementation of a simple (and not very accurate) stopwatch. As will be described shortly, a rigorous division between interface and application functionality is essential for effective user interface engineering, and accordingly the first class that will be implemented is one that supplies the timing capability.

Instances of this class will generate *events* on a regular basis and dispatch them to a *listener* object that will react to them in some way. In this example it will update the time displayed on the stopwatch, but in other applications the receipt of an event may be a signal for some other occurrence. For example, it may cause the value of a temperature sensor to be obtained and logged to a file. The generation and receipt of events is central to the production of GUIs and requires techniques appropriate to *event-driven programming* to be used. Event-driven programming is a style of programming where *flow of control* is determined by the pattern of events that occurs. The class diagram for the *Timer* class is given in Figure 1.1.

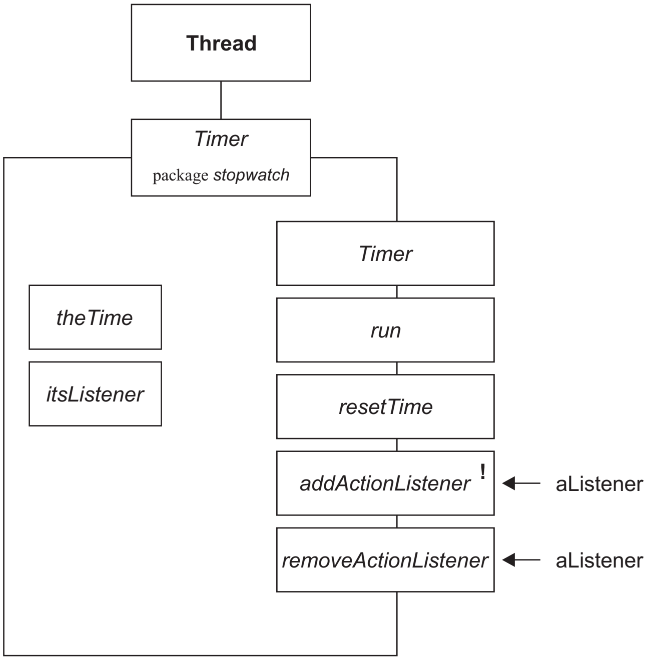


Figure 1.1 Timer class diagram.

The diagram indicates that the *Timer* class is a member of the *stopwatch* package of classes and extends the Thread class, so inheriting the capability to execute independently in a concurrent manner. This facility is required by instances of this class as they have to execute continually in order to attempt to record the passage of time accurately. Of the methods inherited from the Thread class only the *run()* method needs to be overridden in the *Timer* class. When the inherited start() method is called it will arrange for the instance's *run()* method to be executed on a separate flow of control (thread) from that which called the start() method. Execution will continue until the run() method terminates or until the Thread is halted in some other way.

Object → java.lang.Thread

```
public Thread()
```

Default constructor.

```
public sychronized void start()
public final void stop()
public void run()
```

A call of start() will cause a new thread of control to be established and arrange for run() to be executed upon it. Execution will continue until run() terminates, until stop() is called or until the thread is destroyed in some other manner (e.g. the application terminates).

```
public final  void suspend()
public final  void resume()
public static void sleep( int milliseconds)
                          throws InterruptedException
```

A call of suspend() will cause the thread not to execute until resume() is called. A call of sleep() will suspend the thread for (approximately) the number of milliseconds specified.

In this example the *run()* method executes indefinitely within a non-terminating loop. Each pass of the loop arranges for the Thread to sleep() for a period of time, fixed at one tenth of a second in this example, and then to generate and dispatch an event. The instance attribute *theTime* records the number of periods of sleep which have elapsed since the *Timer* was started or last reset.

The other instance attribute, *itsListener*, contains the identity of the object to which the events generated by the *run()* method are dispatched. The two remaining methods, *addActionListener()* and *removeActionListener()*, are supplied in order to support this attribute.

The protocols for the use of events will be one of the major themes of this book and a full explanation will be given in due course. One aspect of the protocols is the distinction between *unicaster* and *multicaster* event sources. Unicasters can only maintain knowledge of, and hence dispatch events to, a single listener at a time. Multicasters, in comparison, maintain a list of listeners, and each event they generate is dispatched to all listeners in the list. In the *Timer* class, the exclamation mark in the top right of the method which registers listeners, *addActionListener()*, indicates that this class is a unicaster, and an exception will be thrown if an attempt is made to register a second listener[1].

[1] The swing package contains a multicaster Timer class that is not suitable for use with this design. Details are given in Exercise 1.5.

Unicaster event sources allow only a single listener to be registered with them; multicaster event sources allow a number of listeners to be registered. When a multicaster event source dispatches an event a *copy* of the event will be sent to each registered listener, but the sequence in which they will be called is not guaranteed. A unicaster can be distinguished from a multicaster in designs and implementations by the TooManyListenersException that it might throw.

One of the simplest Event classes that can be used to convey messages between objects is the ActionEvent class. This class has three attributes: the identity of the instance which was the source of the event, a manifest reason for the generation of the event, known as its ID, and an actionCommand string which can contain information specific to the particular event.

A design decision has been taken to make use of the ActionEvent class to communicate the passage of time between instances of the Timer class and their listeners. The actionCommand of each ActionEvent generated will contain the value, as a String, of the *theTime* attribute when the event was generated.

The implementation of this class, as far as the end of its constructor, is as follows.

```
0001  // Timer.java
0002  // Extended Thread class which implements an ActionEvent
0003  // unicaster protocol and generates an event every 1/10
0004  // of a second.
0005  //
0006  // Written for the JFC book, ch 1, see text.
0007  // Fintan Culwin, v0.1, Dec 1999.
0008
0009  package stopwatch;
0010
0011  import java.awt.event.*;
0012
0013  public class Timer extends Thread {
0014
0015  private int            theTime     = 0;
0016  private ActionListener itsListener = null;
0017
0018     public Timer() {
0019        super();
0020        theTime     = 0;
0021        itsListener = null;
0022     } // End Timer constructor.
```

Following the header comments, line 0009 states that this class is a member of the *stopwatch* **package** of classes and line 0011 imports the entire java.awt.event package in order that this class can have more convenient visibility of the event handling resources. Line 0013 then declares the *Timer* class as an extension of the Thread class, exactly as stated on the class diagram. The instance attribute, *theTime*, is declared on line 0015 as a primitive **int**eger variable with its value confirmed as zero. The other encapsulated attribute, *itsListener*, is declared on line 0016 as an ActionListener instance: the nature of an ActionListener will be explained shortly. The constructor is declared on lines 0018 to 0021 and is implemented as a call of the **super**, Thread, default constructor followed by a confirmation of the initial values of the two attributes.

The *run()* method will be described after the other three public methods, whose implementation follows, have been presented.

```
0045     public synchronized void resetTime() {
0046        theTime = 0;
0047     } // End resetTime;
0048
0049
0050     public void addActionListener( ActionListener listener)
0051                    throws java.util.TooManyListenersException {
0052        if ( itsListener == null) {
0053           itsListener = listener;
0054        } else {
0055           throw new java.util.TooManyListenersException();
0056        } // End if.
0057     } // End addActionListener.
0058
0059
0060     public void removeActionListener( ActionListener listener) {
0061        if ( itsListener == listener) {
0062           itsListener = null;
0063        } // End if.
0064     } // End removeActionListener.
0065
0066  } // End Timer.
```

These methods commence with *resetTime()*, whose requirement is to reset *theTime* to zero, so starting timing again. The only complication with the implementation of this method is that it must be *thread safe*. That is, as a call of this method may originate from another thread in the artifact it is possible that two threads may attempt to reset the timer at the same stage. Alternatively the timer might be in the process of resetting when an event is dispatched, causing it to report the wrong time.

To avoid these consequences the *resetTime()* method is declared with the modifier **synchronized**, which ensures that only one thread can use this method at a time and that while it is being used no other methods of the instance can operate. In general, all classes should be constructed with the possibility of them being used in threaded circumstances, and any methods which change the state of an attribute should be declared with the **synchronized** modifier.

The *addActionListener()* method is noted on the class diagram as possibly throwing an exception. This identifies the class as an event unicaster, with the understanding that the exception will be thrown if an attempt is made to add a second listener. The method prototype, on lines 0050 and 0051, indicates the possibility that an instance of the java.util.TooManyListenersException class may be thrown, which is in accord with the required protocol.

The implementation of the *addActionListener()* method, on lines 0052 to 0056, is a two-way selection which stores the identity of the *listener* passed as an argument in the *itsListener* attribute, so long as it is not already holding the identity of a listener. Or, if it already has a non-**null** value, it constructs and throws an exception.

The *removeActionListener()* method, on lines 0060 to 0064, will only restore the *itsListener* attribute to a **null** value if the *listener* argument identifies the already registered listener; otherwise the call will have no effect.

An ActionListener is an instance of a class that implements the ActionListener **interface**. This is another part of the fundamental Java listener protocols, designed to ensure that it is possible to guarantee at the time of compilation that, when an event source dispatches an event to its registered listener, the listener will have a method which is able to accept the event. When a class declares that it implements a particular **interface** it is promising that that it will supply a certain set of methods and Java will ensure that it does so by reporting a compilation error if it does not. In return for this, Java will allow an instance of the class to be passed as an actual argument when the formal argument is of the **interface** class. Hence, when a listener instance is registered it can be guaranteed that it will have the set of methods specified in the listener **interface** specification available. Only the actions that are mandated by the interface specification can be called via the registered listener and not any other methods which the instance may also have.

> *Listener* ***interface*** *specifications declare a set of methods which a class stating that it implements the* ***interface*** *must provide. Methods to add and remove listeners are used to identify event sources and these methods require as a formal argument an instance of a class that implements the appropriate listener* ***interface****. This guarantees, to the event source, that the methods in the* ***interface*** *specification will be provided by the registered listener.*
>
> *There is an unfortunate overload of the word 'interface' in Java. In some situations it means the specification of a required set of methods (****interface****). In other situations it refers to the way in which application functionality is presented to the user (graphical user interface). The intended use should always be clear from the context.*

In the case of the ActionListener **interface** specification, the only method which is mandated is actionPerformed(), whose prototype indicates that it takes a single argument of the ActionEvent class. Hence any instance which needs to be registered with a *Timer* instance as *itsListener* must be of a class which implements the ActionListener **interface**. This guarantees, to the *Timer* instance, that the registered listener will have an actionPerformed() method, so this method can be called, with a suitable actual argument, when an event needs to be dispatched.

java.awt.event.ActionListener interface

```
public void actionPerformed( ActionEvent event)
```

Method to be called by the event source when the listener needs to be notified that some event has occurred.

This is illustrated within the *Timer run()* method, whose implementation is as follows.

```
   public void run() {
      while ( true) {
         try {
            this.sleep( 100);
         } catch (InterruptedException exception) {
            // Do nothing.
         } // End try/catch
         theTime++;
```

```
0032
0033           if ( itsListener != null) {
0034           String timeString = new String(
0035                           new Integer( theTime).toString());
0036           ActionEvent theEvent = new ActionEvent( this,
0037                                  ActionEvent.ACTION_PERFORMED,
0038                                  timeString);
0039              itsListener.actionPerformed( theEvent);
0040           } // End if.
0041        } // end while.
0042     } // End run.
```

The *Timer run()* method overrides the Thread run() method and will be called, on a separate thread of control, when the *Timer*'s inherited start() method is called. The new thread will continue executing, concurrently with the thread from which it was started, until the *run()* method terminates; or until the thread is stopped in some other way. As this thread is required to continue running indefinitely its implementation is bounded by a non-terminating **while** loop between lines 0025 and 0041.

Within the loop body there are two sections: the first suspends execution of the thread for (approximately) one tenth of a second and then increments the *theTime* instance attribute. The second part of the loop body constructs and dispatches an ActionEvent, reporting the number of tenths of a second which have elapsed to any registered listener.

The first part of the loop body is implemented on lines 0026 to 0031 and involves a call of the inherited Thread sleep() method on line 0027. This method takes an integer as an argument and will suspend execution of the thread for the number of milliseconds specified by the argument. In this example the thread is suspended for 100 milliseconds, which is one tenth of a second. It is possible that the thread will be resurrected before the time requested in the sleep() call has elapsed, and in these circumstances an InterruptedException will be thrown. This exception cannot be ignored, but in this example nothing needs to be done; accordingly an empty exception handler is supplied on lines 0028 to 0030.

Once the **try/catch** structure has terminated, on line 0031, the *theTime* instance attribute is incremented to record the passage of another tenth of a second. The precise amount of time for which a thread is sent to sleep is not guaranteed and the remaining actions of the loop will take a small amount of time to execute before the thread is put to sleep again. Accordingly this implementation cannot be relied upon to give a totally accurate timing facility; one suggestion to make it more accurate will be given as an end of chapter exercise. However, the accuracy of this implementation is sufficient for its purpose, which is to illustrate the generation of events by an event source and the subsequent construction of a graphical user interface to allow users access to the functionality it supplies.

The second part of the loop body is implemented on lines 0033 to 0040 and is responsible for dispatching an event to *itsListener*. It is guarded by an **if** statement, on line 0033, which will omit this part of the loop if no listener is currently registered.

If a listener is registered then it is necessary to send it an ActionEvent whose actionCommand attribute contains a String representing the number of tenths of a second elapsed since the *Timer* was started or last reset. The first step of the **if** body, on lines 0034 to 0035, is to construct the *timeString* from the *theTime* attribute. Having prepared the *timeString*, on lines 0036 to 0038, an ActionEvent, called *theEvent*, is constructed containing it. The three arguments to the constructor are the identity of the object that generated the event (**this**), the manifest reason why the event was dispatched, ActionEvent.ACTION_PERFORMED, and the actionCommand, *timeString*.

Object → EventObject → AWTEvent → java.awt.event.ActionEvent

```
public static final int ACTION_PERFORMED
```

The only manifest reason for dispatching an event.

```
public static final int ALT_MASK
public static final int CTRL_MASK
public static final int META_MASK
public static final int SHIFT_MASK
```

Bit masks for the state of the keys on the keyboard.

```
public ActionEvent( Object source,  int id, String command)
public ActionEvent( Object source,  int id,
                    String command, int modifiers)
```

Constructors specifying the Object that generated the event, the manifest reason for dispatching the event (id) and the actionCommand attribute of the event. The *modifiers* argument indicates the state of the SHIFT, CONTROL, ALT and META keys on the keyboard.

```
public Object getSource()
public int    getID()
public String getActionCommand()
public int    getModifiers()
```

Inquiry methods corresponding to the four possible constructor arguments; getSource() is inherited from EventObject and getID() from AWTEvent.

Once theEvent has been constructed it is used as the actual argument of a call of the *itsListener* actionPerformed() method, on lines 0036 to 0038. As was explained in detail above it can be guaranteed that, whatever the real class of the listener instance, it will have an actionPerformed() method, as it must implement the ActionListener interface in order to be accepted by the addActionListener() method. Within the actionPerformed() method of the listener, as will be described below, the actionCommand attribute can be obtained as a String and the **int**eger number of tenths of a second which it represents retrieved from it. Effectively, every one tenth of a second an instance of the *Timer* class will send a message, using Java's event listener protocols, to its listener informing it of the amount of time which has elapsed. This situation is illustrated in the instance diagram in Figure 1.2. It shows that *aTimer* is an event source and *aListener* is an instance of any class which implements the ActionListener interface and has been registered with *aTimer* as its listener.

1.4 The *TextTimerDemonstration* class

It is essential that application functionality is at least demonstrated, or better tested, before the user interface for it is constructed. If this advice is followed then any faults that are discovered during the development of the interface can be isolated to the user interface code. In this part of the chapter a text-only non-interactive demonstration of the *Timer* class will be provided. This demonstration will be supplied by an instance of the *TextTimerDemonstration* class whose class diagram is given in Figure 1.3.

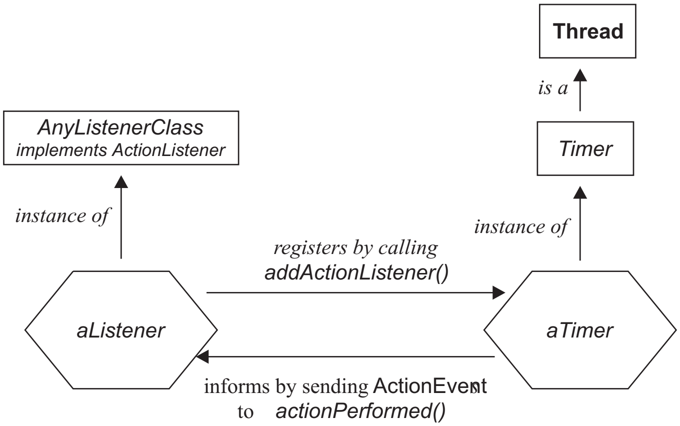


Figure 1.2 *Timer* instance diagram illustrating event protocols.

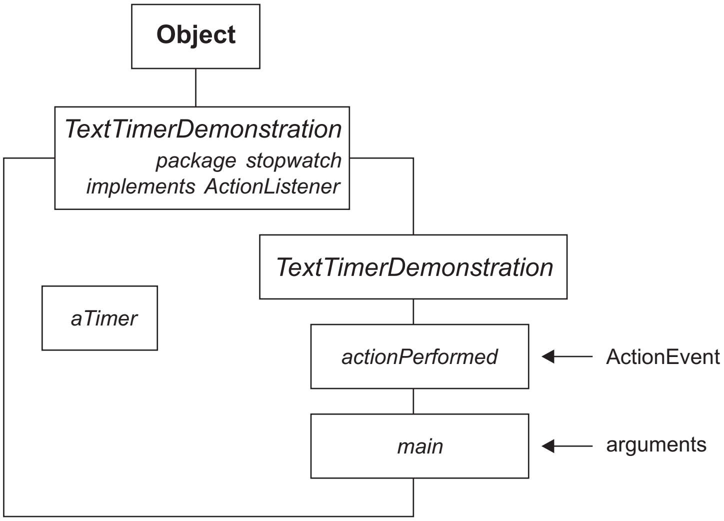


Figure 1.3 *TextTimerDemonstration* class diagram.

The diagram indicates that the *TextTimerDemonstration* class implements the ActionListener interface, so the class must provide an *actionPerformed()* method that takes a single argument of the ActionEvent class. In addition to a single default constructor, the only other public resource is a class-wide *main()* method. The class also has an encapsulated instance attribute of the Timer class called *aTimer*. The implementation of the class, as far as the end of its constructor, is as follows.

```
// TextTimerDemonstration.java
// Text only class which demonstrates receiving
// ActionEvents from a Timer unicaster.
//
// Written for JFC book, ch 1, see text.
// Fintan Culwin, v0.1, Dec. 1999.
```

```
0007
0008  package stopwatch;
0009
0010  import java.awt.event.*;
0011
0012  public class TextTimerDemonstration extends Object
0013                                   implements ActionListener {
0014  private Timer aTimer = null;
0015
0017     public TextTimerDemonstration() {
0018        super();
0019
0020        aTimer = new Timer();
0021        try {
0022           aTimer.addActionListener( this);
0023        } catch ( java.util.TooManyListenersException exception){
0024           // do nothing
0025        } // End try/catch.
0026
0027        aTimer.start();
0028     } // End TextTimerDemonstration constructor.
```

This class, another member of the *stopwatch* package of classes, also has to import the java.awt.event package and, on lines 0012 and 0013, the *TextTimerDemonstration* class is declared as extending the Object class and implementing the ActionListener interface. An encapsulated *Timer* instance, called *aTimer*, is declared on line 0014.

The constructor's declaration commences on line 0017 and its first step, on line 0018, is to call its **super**, Object, constructor. This is followed by the construction of the *Timer* instance, *aTimer*, on line 0020. On line 0022 the *aTimer addActionListener()* method is called to register a listener with the *Timer* instance. The identity of the listener to which the ActionEvents generated by *aTimer* are to be dispatched is this instance of the *TextTimerDemonstration* class currently being constructed. Accordingly **this** is specified as the actual argument of the *addActionListener()* call, which will be allowed as the formal argument is declared to be of the ActionListener interface class that is implemented by *TextTimerDemonstration*.

The consequence of this listener registration is that the ActionEvents generated by the *aTimer* instance will be passed as actual arguments of calls of the *TextTimerDemonstration actionPerformed()* method. The *Timer addActionListener()* method is declared as possibly throwing a TooManyListenersException. This exception cannot be ignored, so the registration of **this** as the listener, on line 0022, has to be contained within a **try/catch** structure. However, as it can be guaranteed that an exception will never be thrown in this demonstration the **catch** part of the structure can be implemented to do nothing.

Once the listener has been registered with the *aTimer* instance its *start()* method is called, on line 0027, as the last step of the constructor. This will cause the *aTimer*'s *run()* method to be called, on a separate thread of control, and will result in a series of ActionEvent instances being passed as arguments to the listener's *actionPerformed()* method. The implementation of this method is as follows.

```
0030     public void actionPerformed( ActionEvent event) {
0031
0032     String theCommand  = event.getActionCommand();
0033     int    timeElapsed = Integer.parseInt( theCommand, 10);
0034
0035        if ( timeElapsed < 31 ) {
0036           System.out.println( timeElapsed);
0037        } else {
0038           System.exit( 0);
0039        } // End if.
0040     } // End actionPerformed.
```

The *Timer run()* method passes the number of tenths of a second that have elapsed, as a String, in the actionCommand attribute of the ActionEvent *event* argument. The first steps of the *actionPerformed()* method, on lines 0032 and 0033, are to obtain the actionCommand attribute into the *theCommand* String and then to convert it into a primitive **int**, called *timeElapsed*. The remaining steps of the *actionPerformed()* method will output the value of *timeElapsed* until 30 tenths of a second have elapsed, whereupon the program will terminate, via a call of System.exit() on line 0038.

The *main()* method merely needs to construct an instance of the *TextTimerDemonstration* class which will automatically construct and start an instance of the *Timer* class. This class will dispatch events to its listener, which will output the time elapsed until the program terminates after 30 events have been handled. The implementation of *main()* is as follows.

```
0042     public static void main( String args[]) {
0043
0044     TextTimerDemonstration theDemo =
0045                                  new TextTimerDemonstration();
0046     } // End main.
0047
0048  } // End class TextTimerDemonstration.
```

The output of this program when it is executed will be all the integers from 1 to 30, following which it will finish. The instance diagram for this program is, apart from the appropriate names, as given in Figure 1.2.

1.5 The *TimerDemonstration* artifact

The second *Timer* demonstration class will also provide an introduction to the techniques that can be used to build an artifact which is able to appear either within a separate top-level application window or within an area in a Web page. This will require a more complex three-layer architecture, as illustrated in Figure 1.4.

The artifact contains three instances of three different classes. An instance of the *Timer* class, on the right, supplies the application functionality and generates events which are listened to by an instance of the *TimerDemonstration* class, in the middle. This class, in turn, displays the elapsed time in an instance of the *TimerLabel* class, shown on the left of the diagram. Three-layer models divide artifacts that have a graphical user interface into three different areas of concern: application, behavioral and presentational, as illustrated above.

The appearance of this artifact, as an application and as an area within a Web page, is illustrated in Figure 1.5. One immediately visible difference in the output is that the elapsed time is shown in a formatted manner indicating the number of hours, minutes, seconds and tenths of a second that have elapsed.

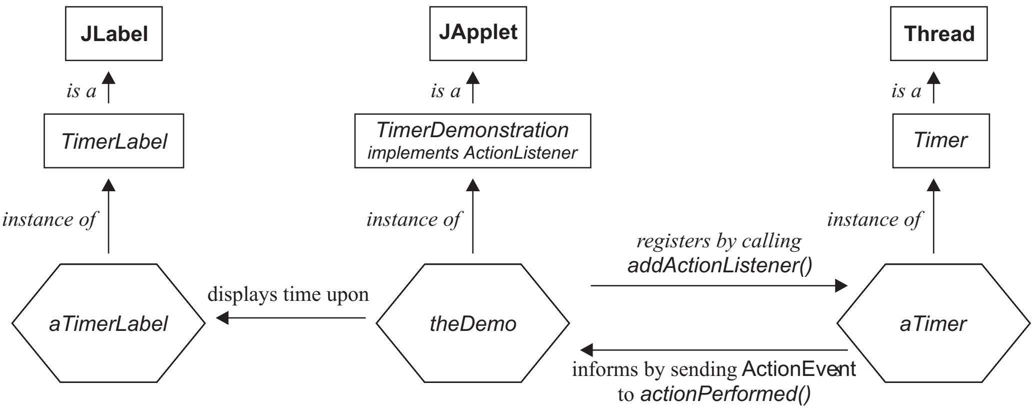
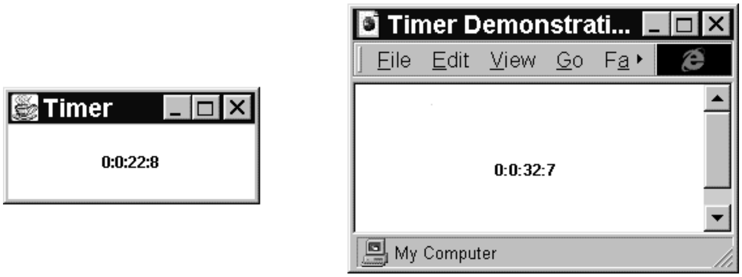


Figure 1.4 *TimerDemonstration* three-layer instance diagram.



0:0:22:8

0:0:32:7

Figure 1.5 *TimerDemonstration* as an application and as an applet.

The instance diagram also indicates that the *TimerLabel* class is an extension of the JLabel class. The JLabel class will be described in detail in Chapter 3: briefly it supplies the facility to display a text label, and/or an icon, in a window within an interface. In this example it will be restricted to showing only a text label, which is continually updated to show the elapsed time. The class diagram for the *TimerLabel* class is given in Figure 1.6.

As it is very unlikely that a *TimerLabel* instance will ever be required outside the context of the *stopwatch* package of classes the class diagram indicates that its constructor and its single method, does not have fully public visibility. The *TimerLabel* constructor overrides the JLabel default constructor and supplies an additional method called *setTime()* which takes an argument identified as *timeNow*. The *timeNow* argument is supplied to the *setTime()* method as an integer indicating the number of tenths of a second which are to be shown. The *TimerLabel* instance is responsible for formatting this value before it is displayed in the window. This is in accord with the division of responsibilities, as described above. In terms of the three-layer model the *TimerLabel* instance is the presentational component, so it should take responsibility for deciding how the output is to be shown to the user.

The implementation of the *TimerLabel* class as far as the end of its constructor, is as follows.

```
// TimerLabel.java
```

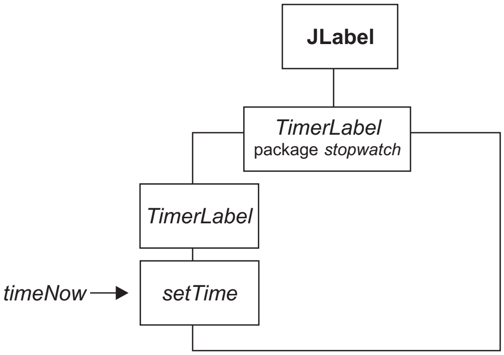


Figure 1.6 *TimerLabel* class diagram.

```
0002 // Supplies a window within which the
0003 // state of a Timer can be shown.
0004 //
0005 // Written for the JFC book, ch 1, see text.
0006 // Fintan Culwin, v0.1, Dec. 1999.
0007
0008 package stopwatch;
0009
0010 import java.awt.*;
0011 import javax.swing.*;
0012
0013 class TimerLabel extends JLabel {
0014
0015 private static final int TENTHS_PER_SECOND  = 10;
0016 private static final int SECONDS_PER_MINUTE = 60;
0017 private static final int MINUTES_PER_HOUR   = 60;
0018 private static final int SECONDS_PER_HOUR   = 3600;
0019
0020
0021    protected TimerLabel(){
0022       super();
0023       this.setOpaque( true);
0024       this.setBackground( Color.white);
0025       this.setForeground( Color.black);
0026       this.setHorizontalAlignment( SwingConstants.CENTER);
0027    } // End TimerLabel default constructor.
```

Following the header comments, line 0008 states that this class is a member of the stopwatch package of classes. The required importations are effected on lines 0010 and 0011. Some facilities of the AWT will be required by this class, so line 0010 imports the entire awt package. Likewise, some facilities of the JFC will be required, most obviously the JLabel class, so the entire swing (JFC) package is imported on line 0011.

Line 0013 is the declaration of the *TimerLabel* class as a non-public extension of the JLabel class. This allows instances of the *TimerLabel* class to inherit all the attributes and methods of the JLabel class and need only supply attributes and methods which are specific to its specialized nature, in this example a method to format and display a time. Lines 0015 to 0018 declare some constants which will be useful to the *setTime()* method.

The **protected** constructor commences on line 0021 and its first step is to call its **super**, JLabel, constructor. It then configures itself by establishing opaque visibility, a white background, a black foreground and setting its horizontalAlignment resource so that the label it displays is centered within the available width. These resources, and the reasons for establishing these values, will be explained in more detail in the following two chapters.

The implementation of the **protected** *setTime()* method is as follows.

```
0030     protected void setTime( int timeNow) {
0031
0032     int tenths  = timeNow % TENTHS_PER_SECOND;
0033     int seconds = (timeNow / TENTHS_PER_SECOND)  %
0034                                             SECONDS_PER_MINUTE;
0035     int minutes = (timeNow /
0036                     (TENTHS_PER_SECOND * SECONDS_PER_MINUTE)) %
0037                                                MINUTES_PER_HOUR;
0038     int hours   = timeNow /
0039                         (TENTHS_PER_SECOND * SECONDS_PER_HOUR);
0040
0041     String timeString = new String( hours   +":" +
0042                                     minutes +":" +
0043                                     seconds +":" +
0044                                     tenths);
0045        this.setText( timeString);
0046     } // End setTime.
0047
0048  } // End class TimerLabel.
```

This method commences, on lines 0032 to 0039, by computing the number of tenths of a second, seconds, minutes and hours which are represented by the *timeNow* argument, using the constants previously declared. Having done this, lines 0041 to 0044 then construct a String, called *timeString*, which contains these values catentated together and separated by colons. The final step of the *setTime()* method is to call the inherited JLabel setText() method passing the *timeString* as an argument. The consequence of this call is to cause the formatted time string to be displayed in the *TimerLabel*'s window, as can be seen in Figure 1.5.

An instance of the *TimerLabel* will be constructed by the *TimerDemonstration* class as it is initialized and its *setTime()* method will be called from its *actionPerformed()* method, every time it receives an ActionEvent from the Timer instance which it also constructs during initialization. This is in accord with the instance diagram in Figure 1.4 and again reinforces the division of the artifact according to the three-layer model.

The class diagram for the *TimerDemonstration* class is given in Figure 1.7. It shows that the *TimerDemonstration* class extends the JApplet. It also shows that it implements the ActionListener **interface** in order to be able to listen to ActionEvents generated by the encapsulated *aTimer* Timer instance. The class diagram also indicates that it contains an encapsulated instance of the *TimerLabel* class, called *aTimerLabel*.

The JApplet class supplies all the resources and functionality that allow an artifact to appear within a browser window, or in a separate top-level window on the desktop; to

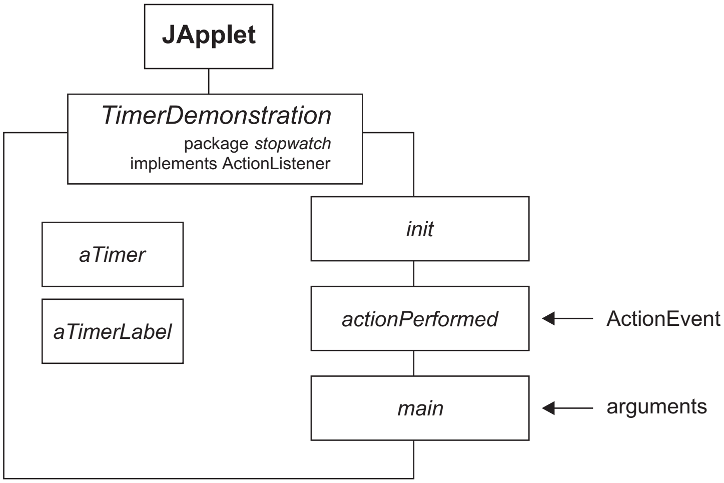


Figure 1.7 *TimerDemonstration* class diagram.

respond appropriately to the user's actions as they interact with the interface it presents; collect and dispatch events from the interface; and much more as well. As the construction of a JApplet instance is a complex task, fraught with potential problems, it is not a good idea to override it. Instead, a method called *init()* is supplied by JApplet which will be called during the applet's construction when it is safe to do so. Accordingly, any initialization steps which have to be taken by a particular artifact should be placed in an overridden *init()* method. (Alternatively, the *init()* method can be thought of, not strictly correctly, as the applet's constructor).

Accordingly the *TimerDemonstration* class does not supply a constructor, but does supply an *init()* method. It also supplies an *actionPerformed()* method in order to be able to listen to ActionEvents from the *Timer* and a *main()* method in order to be executed as an application.

The implementation of this class, as far as the start of its *init()* method, is as follows.

```
// TimerDemonstration.java
// Timer class which demonstrates receiving
// ActionEvents from a unicaster and showing
// them in a window.
//
// Written for JFC book, ch 1, see text.
// Fintan Culwin, v0.1, Dec 1999.

package stopwatch;

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


public class TimerDemonstration extends JApplet
                             implements ActionListener {

```

```
0018  private Timer      aTimer      = null;
0019  private TimerLabel aTimerLabel = null;
```

Following the header comments, the package declaration and the importation of the required facilities, the class is declared in accord with the class diagram on lines 0015 and 0016. Lines 0018 and 0019 then declare, but do not construct, the two private attributes. The *init()* method is implemented as follows.

```
0022     public void init() {
0023
0024        aTimerLabel = new TimerLabel();
0025        this.getContentPane().add( aTimerLabel);
0026
0027        aTimer = new Timer();
0028        try {
0029           aTimer.addActionListener( this);
0030        } catch ( java.util.TooManyListenersException exception){
0031           // do nothing
0032        } // End try/catch
0033        aTimer.start();
0034     } // End init.
```

This method is divided into two parts, the first concerned with the *TimerLabel* and the second with the *Timer*. The second part of the method, from line 0027 to 0033, is identical to the *TextTimerDemonstration* constructor. It constructs an instance of the *Timer* class, registers itself as its actionListener (within a **try/catch** structure) and finally starts the *Timer*.

Prior to this the *TimerLabel* is constructed, on line 0024, and added to **this** JApplet, so as to become visible when the applet becomes visible. The JApplet will, during its construction, request and obtain an area on the screen within which to display itself. Within this area any components that have been added, or added to containers that are added to it, will be able to show themselves.

This is slightly complicated, as the components cannot be added directly to the JApplet, but have to be added to one of a number of different *panes* (as in window*panes*) that it contains. The pane which has default visibility is the contentPane, so the *TimerLabel* is added to it. The identity of the contentPane is obtained using **this** JApplet's getContentPane() method. The consequence of line 0025 is that when the applet becomes visible its single child component, *aTimerLabel*, will become visible within it.

The implementation of the *TimerDemonstation actionPerformed()* method is as follows.

```
0037     public void actionPerformed( ActionEvent event) {
0038
0039     String theActionCommand = event.getActionCommand();
0040     int    tenthsElapsed    = Integer.parseInt(
0041                                  theActionCommand, 10);
0042        aTimerLabel.setTime( tenthsElapsed);
0043     } // End actionPerformed.
```

This method will be called by the *aTimer* instance approximately every tenth of a second with an ActionEvent argument whose actionCommand() String contains a representation of the number of tenths of a second which have elapsed. Its task is to extract the actionCommand attribute of the *event*, convert it to a primitive **int** and pass this **int** value as an argument to the *aTimerLabel*'s *setTime()* method.

Line 0039 retrieves the actionCommand from the *event* storing it in the local String *theActionCommand*. This string is then converted to an **int**, called *tenthsElapsed*, which is subsequently used, on line 0042, as the actual argument of a call of *aTimerLabel*'s *setTime()* method. The consequence of these steps is that approximately every tenth of a second the time displayed within the applet is updated.

As it stands at the moment the class is capable of being displayed within a Web page in a suitably enabled browser, as a main() method is not required for this capability. In order to accomplish this the HTML <APPLET> tag must be used to instruct the browser which applet to obtain and execute. A minimal HTML file, which was used to obtain the right-hand illustration in Figure 1.5, is as follows.

```
<HTML>
<HEAD>
<TITLE>Timer Demonstration</TITLE>
</HEAD>
<BODY>
<CENTER>
<APPLET CODE="stopwatch.TimerDemonstration"
        WIDTH=150 HEIGHT=100>
</APPLET>
</CENTER>
</BODY>
</HTML>
```

The <APPLET> tag, in the HTML <BODY> section, indicates the name and location of the class file to be loaded and executed; it must also indicate the HEIGHT and WIDTH of the area required within the browser.

In order for the artifact to be used as an application with its own top-level window, outside any browser, a *main()* method must be provided. The *TimeDemonstration main()* method is as follows.

```
0046     public static void main( String args[]) {
0047
0048     JFrame             frame   = new JFrame("Timer");
0049     TimerDemonstration theDemo
0050                            = new TimerDemonstration();
0051
0052        theDemo.init();
0053        frame.getContentPane().add(
0054                       theDemo, BorderLayout.CENTER);
0055
0056        frame.pack();
0057        frame.setVisible( true);
0058     } // End main.
0059
0060  } // End class TimerDemonstration.
```

The top-level window is provided by an instance of the JFrame class, called *frame*, declared and constructed on line 0048. The string argument to the constructor is the title that will be used as part of the window border's decoration, as can be seen in the left-hand image in Figure 1.5. Lines 0049 and 0050 construct an instance of the *TimerDemonstration* class called *theDemo*.

Initialization of *TimerDemonstration* takes place automatically when it is used as an applet within a Web page. However, when it is used as an application initialization must be explicit. Accordingly, the first step of the *main()* method, on line 0052, is to call *theDemo*'s *init()* method. Having initialized *theDemo* it can be added to the *frame*'s contentPane. JFrame instances' contentPanes have a default layout *management policy* known as BorderLayout. A layout management policy determines where the children of a container are positioned relative to each other and how much space each is accorded.

Layout management will be discussed in detail in various parts of the early chapters of this book. All that needs to be known for the moment is that the most suitable location within the *frame* to position the *theDemo* applet is in its center location. On line 0054, when *theDemo* is added to the *frame*'s contentPane the second argument, BorderLayout.CENTER, indicates which part of the BorderLayout is to be used.

Having added *theDemo* to the *frame* it is posted to the desktop by first, on line 0056, packing the components contained within the frame. This will cause layout negotiations to take place, the result of which is that the size and position of the interface is determined. Once this has happened, on line 0057, it is made visible and hence appears. The consequence of running this artifact as an application from the command line is that its *main()* method will be executed. This, in turn, will cause an instance of the *TimerDemonstration* class to be shown within a Frame when it becomes visible on the desktop, in its own top-level window.

> *For simple artifacts the provision of a main() method need not be any more complicated that the main() method used here. All that needs to be changed is the name of the class that supplies the applet on lines 0049 and 0050. The development of an artifact can be much more convenient for the developer if it can be executed from the command line rather than being loaded, and re-loaded, into a Web browser.*

The meaning of the three-layer architecture, as illustrated in Figure 1.4, should now be clearer. The major part of this artifact is the *TimerDemonstration* instance, *theDemo*. It constructs an instance of the *Timer* class, called *aTimer*, and registers itself with it as its listener. It also constructs an instance of the *TimerLabel* class, called *aTimerLabel*, and adds it to its own *contentPane* in order to make it visible to the user. When the *aTimer* instance implements the application's timing functionality by generating an ActionEvent containing the elapsed time, *theDemo* both listens to this event and also calls a method of the *aTimerLabel* which formats the value as appropriate and presents it to the user. Hence the *theDemo* component supplies the behavior for this artifact a role which will become clearer when the next artifact, the *Stopwatch*, is described.

1.6 The *Stopwatch* artifact

The *Stopwatch* artifact uses the functionality of a *Timer* instance to provide the user with a simple stopwatch which can be used to time the duration of episodes. The appearance of the artifact is shown in Figure 1.8, which shows the stopwatch in three different states. On the left is the *reset state*, where the time displayed at the top of the interface is unchanging and set to zero. Of the three buttons below it only the *start* button is capable of being pressed, and this is indicated by the other two buttons' labels being shown in a grayed-out manner. When the interface is first presented to the user it is in this state.

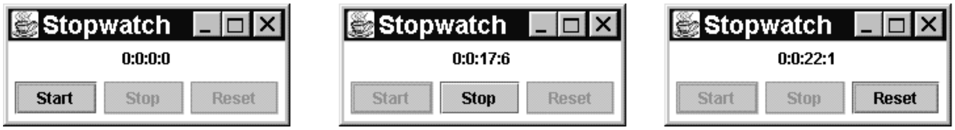


Figure 1.8 *Stopwatch* artifact showing the three states.

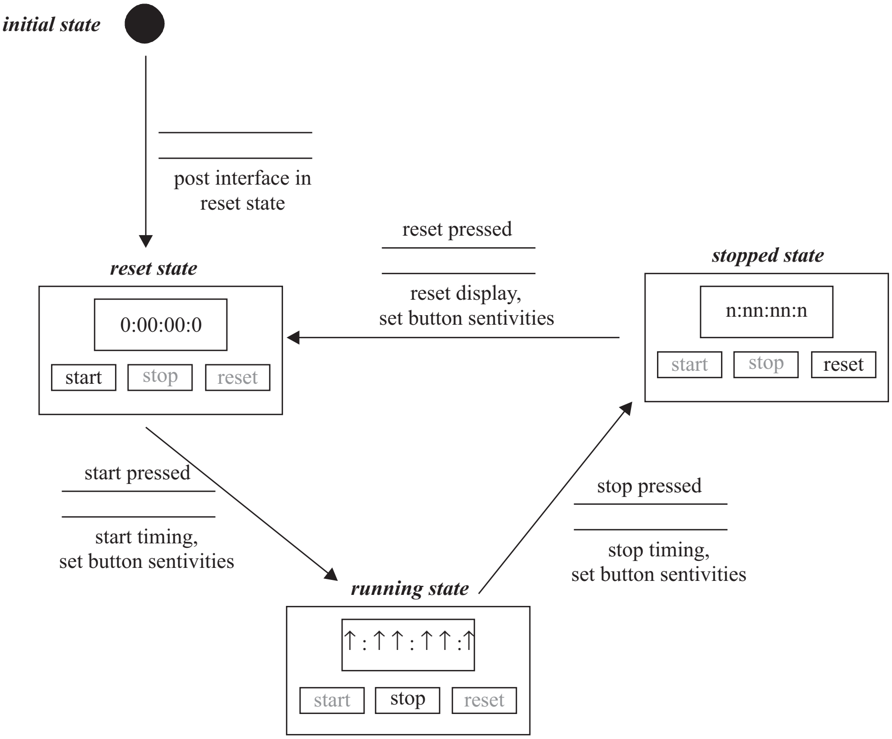


Figure 1.9 *Stopwatch* STD.

In the middle is the *running state*, obtained by pressing the *start* button in the reset state, where the display is continually updating and only the *stop* button is capable of being pressed. On the right is the *stopped state*, obtained by pressing the *stop* button in the running state; here the display is not updated and continues to show the time when the *stop* button was pressed. In this state only the *reset* button is capable of being pressed, and pressing it causes the interface to revert to the reset state.

A description of the required behavior of an interface can be more conveniently and precisely expressed in a *S*tate *T*ransition *D*iagram (STD). The STD for the *Stopwatch* artifact is given in Figure 1.9.

A state transition diagram consists of a number of *states*, generally shown as rectangular boxes, which might illustrate salient parts of the appearance of the interface while it is in that state, connected by *transitions*, shown as arrows with three labels. An STD will have an *initial state*, shown by a solid circle and may have a *terminal state*, shown by a bullseye circle.

The three labels on the transition arrows indicate the *event* that must occur for the transition to be considered, the *pre-condition* that must be true for the transition to be followed and the *consequences* of following that transition. Any or all of the labels may be empty (although if all three were empty the meaning of the transition would be most unclear). If the *event* is empty then the *pre-condition* will be tested as soon as the interface arrives in that state and the transition will be taken only if it is true. An empty *pre-condition* is taken as true, so the transition will be followed as soon as the *event* occurs. Empty consequences indicate that nothing happens, apart from the state change, during the transition. In general, empty consequences should be treated with suspicion, as there will be no visible indication to the user that a transition has occurred and so no way by which they can become aware of it.

The *Stopwatch* STD starts at the top with the transition from the *initial* to the *reset state*. As both the event and the pre-condition label are empty this transition will always be followed and results in the interface becoming visible to the user. From this state only the *start* button can be pressed, which always results in a transition to the running state, starting the timer as it does so. The illustration of this state is using upward facing arrows to indicate that the display is continually updated. The transition from this state to the stopped state always takes place when the stop button is pressed and stops timing as it does so. The illustration of this state uses *n*s in the display to indicate that it is frozen with a non-zero value. In this state, only the reset button can be pressed, which always causes the transition back to the reset state to be taken, causing the display to be reset to zero.

An STD, like all formal notations, should provide a precise, communicable and unambiguous description of the behavior of the artifact that is to be constructed. It can be used at the design stage to assess the usability of the interface. For example, the small number of states and total absence of pre-conditions indicates that this interface should be simple and intuitive for the user.

The architecture of the implementation of this interface will again use a three-layer model; its instance diagram is given in Figure 1.10. Again, on the right, the essential functionality of this artifact is supplied by an instance of the *Timer* class which is both constructed and listened to by the main component; an instance of the *Stopwatch* class which extends the JApplet class. The main component makes use of an instance of the *StopwatchUI* class, shown on the left of the diagram.

This presentation component is more complex than the presentation component used in the *TimerDemonstration* applet. It extends the JPanel class in order to be able to lay out the components which it accommodates, and contains a single instance of the *TimerLabel* class together with three instances of the JButton class. The JButton instances will generate ActionEvents when the user presses them, and these are also listened to by the *stopwatch* instance.

Consequently the *actionPerformed()* method of instances of the *Stopwatch* class will be listening to ActionEvents from the buttons on the interface and also to those generated by the *Timer*. Its responsibility will be to implement the behavior defined by the STD so that the STD can be used after construction to validate that it is correctly doing so.

Before commencing the construction of the *StopwatchUI* class, the geometrical relationships between its components, and the layout management policies needed to achieve them, need to be considered. This can be accomplished by means of a *layout management diagram*; for the *StopwatchUI* this is given in Figure 1.11.

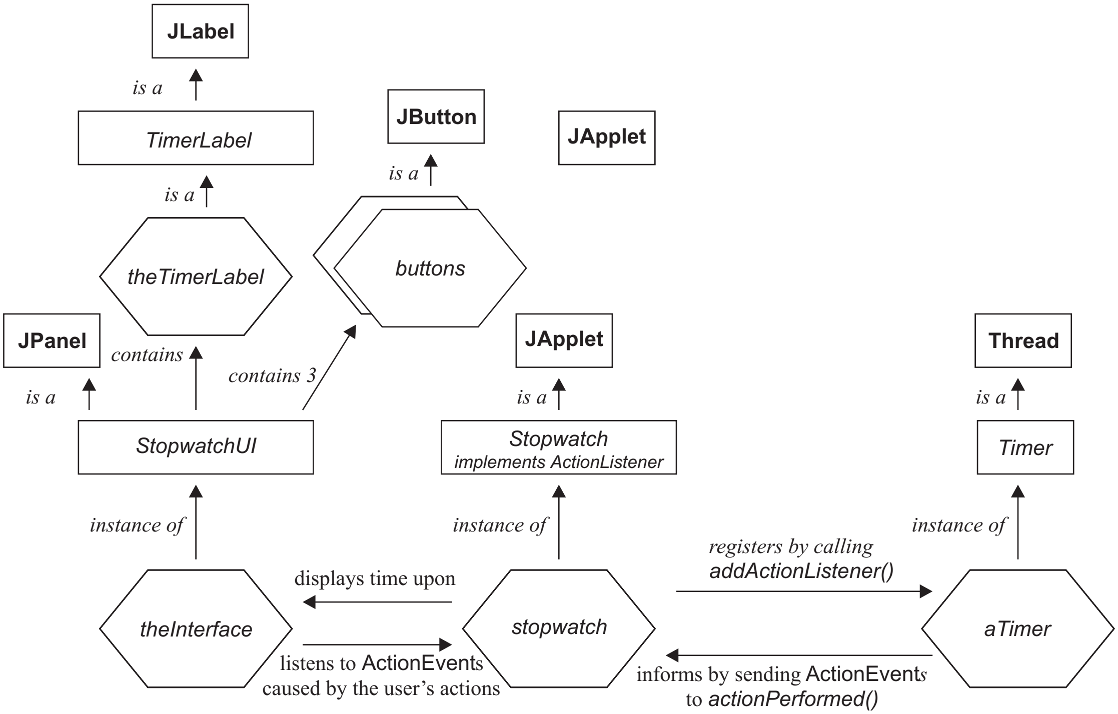


Figure 1.10 *Stopwatch* instance diagram.

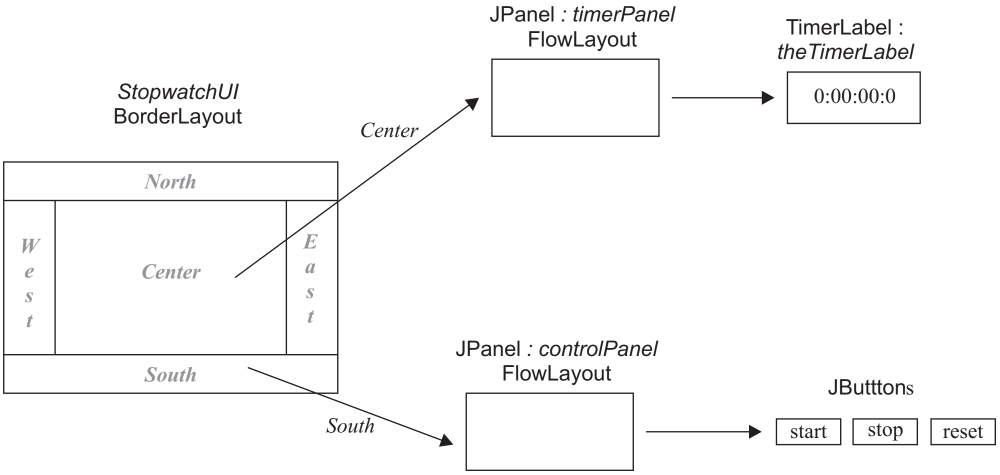


Figure 1.11 *StopwatchUI* layout management diagram.

The layout management diagram shows the relationships between the highest level window provided by the *StopwatchUI* and all other JFC components that are added to it. In order to obtain the required layout the window's contentPane has to have a BorderLayout management policy. This layout policy divides the window into five areas,

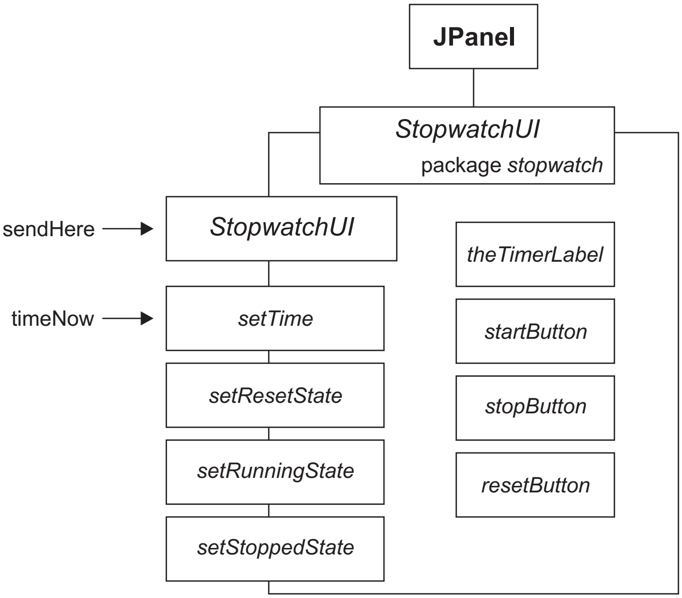


Figure 1.12 *StopwatchUI* class diagram.

identified as *North*, *South*, *East*, *West* and *Center*, with relationships as shown. Each area can contain a single component and there is no requirement for all components to be the same size. For the *Stopwatch* artifact only the center and south locations are occupied, each with an instance of the JPanel class as shown.

The JPanel instance in the *Center* location, *timerPanel*, contains a single component: an instance of the *TimerLabel* class called *theTimerLabel*. The JPanel instance in the *South* location, *controlPanel*, contains three components: each of the JButton class. A JPanel has, by default, a FlowLayout management policy; this policy allows any number of children and will lay them out in a left–right manner according to the sequence in which they are added to the panel.

When these relationships have been established, and before the interface is presented to the user, *layout negotiations* will take place to decide where each component will appear. This process will start with the *StopwatchUI* window asking both of the JPanels how large they would like to be. Each JPanel will decide by asking each of its children how large they would like to be. In this example the reply from the *timerPanel* will request a width and height based upon the preferred size of *theTimerLabel*. The *controlPanel* will request a width and height based upon the combined width of all three JButtons and the height of the highest.

In this example the window will probably give each of the panels its requested height and will give both panels a width wide enough to accommodate all three of the buttons. This will be too wide for *theTimerLabel*, which will be displayed centered within the available space. The width given to the *controlPanel* will be just wide enough for all three buttons to be laid out. If an intermediate *timerPanel* were not used then *theTimerLabel* would be forced to be as wide as the three buttons, which would not be as appropriate.

The other intermediate panel, the *controlPanel*, is required in order to provide FlowLayout control for the three buttons.

Having decided upon the layout relationships and management policies required by the interface the design and construction of the *StopwatchUI* can commence. The class diagram for the *StopwatchUI* class is given in Figure 1.12.

The constructor and methods have a non-public visibility for the same reasons as those previously explained for the *TimerLabel* class. The diagram shows as encapsulated attributes only the visible components that will appear on the interface. The constructor takes an argument, *sendHere*, which is the identity of the ActionListener to which the three buttons are to send their ActionEvents when the user presses them. The *setTime()* method takes an integer argument called *timeNow* and will cause it to be formatted and displayed on the encapsulated *TimerLabel*. The three remaining actions effect the visible transitions to each of the states identified in their names. For example, the *setResetState()* will make sure that the *TimerLabel* displays zero elapsed time and ensures that only the *start* button is sensitized.

The implementation of this design, as far as its constructor, is as follows.

```
0001  // StopwatchUI.java
0002  // Presentation class which contains a
0003  // TimerLabel and three buttons allowing
0004  // the user to operate it as a simple
0005  // stopwatch.
0006  //
0007  // Written for JFC book chapter 1 - see text.
0008  // Fintan Culwin, v0.1, Dec 1999.
0009
0010  package stopwatch;
0011
0012  import java.awt.*;
0013  import java.awt.event.*;
0014  import javax.swing.*;
0015
0016  class StopwatchUI extends JPanel {
0017
0018  private TimerLabel theTimerLabel = null;
0019  private JPanel     timerPanel    = null;
0020
0021  private JButton    startButton   = null;
0022  private JButton    stopButton    = null;
0023  private JButton    resetButton   = null;
0024  private JPanel     controlPanel  = null;
```

Following the header comments are the package declaration, the required importations and the class declaration; on lines 0018 to 0024 the components which are required for the assembly of the interface, as shown in the layout management diagram, are declared. The implementation of the constructor is as follows.

```
0026     protected StopwatchUI( ActionListener sendHere) {
0027
0028        super();
0029        this.setLayout( new BorderLayout());
0030
0031         theTimerLabel = new TimerLabel();
0032
0033         timerPanel    = new JPanel();
0034         timerPanel.setBackground( Color.white);
0035         timerPanel.add( theTimerLabel);
0036         this.add( timerPanel, BorderLayout.CENTER);
0037
0038         startButton = new JButton(     "Start");
0039         startButton.setActionCommand(  "start");
0040         startButton.addActionListener( sendHere);
0041
0042         stopButton = new JButton(     "Stop");
0043         stopButton.setActionCommand(  "stop");
0044         stopButton.addActionListener( sendHere);
0045
0046         resetButton = new JButton(     "Reset");
0047         resetButton.setActionCommand(  "reset");
0048         resetButton.addActionListener( sendHere);
0049
0050         controlPanel = new JPanel();
0051         controlPanel.setBackground( Color.white);
0052         controlPanel.add( startButton);
0053         controlPanel.add( stopButton);
0054         controlPanel.add( resetButton);
0055
0056         this.add( controlPanel, BorderLayout.SOUTH);
0057      } // End StopwatchUI default constructor.
```

The constructor commences, on line 0028, by calling the **super**, JPanel, constructor and then, on line 0029, establishes a BorderLayout policy for itself. The *TimerLabel* is then constructed, following which, on lines 0033 and 0034, the *timerPanel* is constructed and has its background color attribute set to white. Line 0035 then adds *theTimerLabel* to the *timerPanel*, following which the *timerPanel* is added to the center location of **this** *StopwatchUI* currently being constructed. These calls of various add() methods implement the relationships shown in the upper part of the layout management diagram in Figure 1.11.

Having constructed the upper part of the interface the next stage in the construction of the *StopwatchUI* is to construct the lower part. Lines 0038 to 0040, 0042 to 0044 and 0046 to 0048 comprise a repeated pattern, which constructs and configures each of the JButton instances in turn. Taking the *start* button as an example, line 0038 constructs the button by calling a JButton constructor, the argument to this constructor is the text label which the button is to display. Line 0039 then establishes an actionCommand for this instance. The JButton's actionCommand resource will be copied to the ActionEvents that it generates when it is pressed as their actionCommand resource. This mechanism, as will be illustrated below, allows the listener to determine which of the buttons, if any, dispatched the event. Finally, on line 0040 the button's addActionListener() method is used to register the identity of the ActionListener instance to which the ActionEvents are to be dispatched. The argument is *sendHere*, which is the ActionListener argument passed to the *StopwatchUI* constructor. The absence of a **try/catch** structure around the call of these methods allows it to be inferred that JButtons are event multicasters, not unicasters.

Once all three buttons have been constructed and configured the *controlPanel* is constructed on line 0050, has its background set to white on line 0051 and then has all three buttons added to it in the requisite sequence for the required visual appearance shown in Figure 1.11. As the *controlPanel* is now complete it is added into the *StopwatchUI*'s *South* location on line 0056.

The effect of this constructor is to construct, configure and assemble the components needed to produce the required interface layout. It also ensures that the ActionEvents produced when the user presses the buttons on the interface are dispatched to the listener supplied as an argument. The implementation of the *StopwatchUI* class continues with the *setTime()* method, as follows.

```
0061     protected void setTime( int timeNow) {
0062        theTimerLabel.setTime( timeNow);
0063        timerPanel.doLayout();
0064     } // End setTime.
```

The first step of this method is to call *theTimerLabel*'s *setTime()* method, passing on the argument received. This will cause the time value to be formatted and displayed, as described in the previous part of this chapter. This may cause the preferred size of *theTimerLabel* to change, and to ensure that it remains fully visible the *timePanel*'s doLayout() method is called. This will cause the panel to engage in layout negotiations with all of its children and to accommodate any requested changes in size so long as it has the space to do so. In this example *theTimerLabel* may ask for a width which is a little smaller or bigger than it currently has and it is very likely that this will be allowed. The effect of calling the *StopwatchUI setTime()* method is thus to cause the time to be displayed on the interface and ensure that it is always fully visible.

The next method to be implemented is the *setResetState()* method, as follows.

```
0067     protected void setResetState() {
0068        startButton.setEnabled( true);
0069        stopButton.setEnabled(  false);
0070        resetButton.setEnabled( false);
0071        this.setTime( 0);
0072     } // End setResetState.
```

All AWT and JFC components inherit a setEnabled() method from the AWT Component class. The argument to this method is a **boolean** value which, if **true**, ensures that the functionality of the interface is made available to the user and, if **false**, both prevents any interaction and changes the Component's appearance to make this apparent. In the reset state only the *startButton* should be available for the user to press and the sequence of calls on lines 0068 to 0070 accomplishes this. The only other aspect of the interface's visible state that needs to be ensured when it is in the reset state is the time value displayed. This should always be zero in the reset state and this is accomplished, on line 0071, by calling **this** *StopwatchUI's setTime()* method with the argument 0.

The two remaining methods, *setRunningState()* and *setStoppedState()*, are similar to the *setResetState()* method, but need not be concerned with the value displayed on the *TimerLabel*. They are implemented as follows.

```
0074     protected void setRunningState() {
0075        startButton.setEnabled( false);
0076        stopButton.setEnabled(  true);
```

```
      resetButton.setEnabled( false);
   } // End setRunningState;

   protected void setStoppedState() {
      startButton.setEnabled( false);
      stopButton.setEnabled(  false);
      resetButton.setEnabled( true);
   } // End setRunningState;

} // End class StopwatchUI.
```

This completes the description of the design and implementation of the *StopwatchUI* class. An instance of this class is constructed from the *init()* method of the *Stopwatch* class which will be described next. A class diagram for this class will not be given, as, apart from the names, it would be identical to that given for the *TimerDemonstration* class in Figure 1.7. The implementation of the *Stopwatch* class, as far as the start of the *init()* method, is as follows.

```
0001  // Stopwatch.java
0002  // A class which demonstrates receiving
0003  // ActionEvents from a unicaster, from
0004  // a GIU and responding to them.
0005  //
0006  // Written for the JFC book chapter 1 - see text.
0007  // Fintan Culwin, v0.1, Dec 1999.
0008
0009  package stopwatch;
0010
0011  import java.awt.event.*;
0012  import javax.swing.*;
0013
0014
0015  public class Stopwatch extends JApplet
0016                      implements ActionListener {
0017
0018  private final static int INITIAL = 0;
0019  private final static int RESET   = 1;
0020  private final static int RUNNING = 2;
0021  private final static int STOPPED = 3;
0022  private int theState = INITIAL;
0023
0024  private Timer       aTimer = null;
0025  private StopwatchUI theUI  = null;
```

Following the (by now) expected header comments, package declaration, importations and class declaration, lines 0018 to 0021 declare a set of manifest values to model the four states of the interface. This is followed, on line 0022, by the declaration of a primitive **int** variable called *theState* whose value is stated as *INITIAL*. The intention is that the value of this attribute will always reflect the state that the artifact is currently in. This part of the class declares an instance of the *Timer* class, called *aTimer*, and an instance of the *StopwatchUI* class, called *theUI*, both of which will be constructed by the *init()* method, as follows.

```
0028     public void init() {
0029
0030        theInterface = new StopwatchUI( this);
0031        this.getContentPane().add( theInterface);
0032        theUI.setResetState();
0033        theState = RESET;
0034
0035        aTimer = new Timer();
0036        try {
0037           aTimer.addActionListener( this);
0038        } catch ( java.util.TooManyListenersException exception){
0039           // do nothing
0040        } // End try/catch
0041        aTimer.start();
0042     } // End init.
```

As with the *TimerDemonstration init()* method, this method can be divided into two parts; the first concerned with the interface and the second with the timer. The interface part commences, on line 0030, by constructing a *StopwatchUI* instance referenced by *theUI*. The argument to this constructor is the identity of the ActionListener to which the ActionEvents from the buttons on the interface are to be dispatched. The listener specified is **this** instance of the *Stopwatch* class currently being initialized, so the events will arrive as arguments of its *actionPerformed()* method.

Having constructed the interface it is added to the *Stopwatch*'s contentPane and this step will ensure that the interface is made visible to the user as soon as the applet posts itself into the browser window or onto the desktop. When this happens the interface should be in its reset state, and to accomplish this the *setResetState()* method is called, on line 0032, with the value of *theState* updated to reflect this. The remaining steps of the *init()* method, concerned with constructing, configuring and starting the *Timer*, are identical to those previously presented in the *TimerDemonstration init()* method.

The consequence of this initialization is that *theInterface* will become visible to the user in the reset state and when the user presses the *start* button an ActionEvent containing "*start*" as its actionCommand attribute will be dispatched to the *actionPerformed()* method. This method will also be continually receiving ActionEvents from the *Timer* instance *aTimer* and is implemented as follows.

```
0044     public synchronized void actionPerformed(
0045                                     ActionEvent event) {
0046
0047     String theCommand = event.getActionCommand();
0048
0049        if ( theCommand.equals( "start")) {
0050           aTimer.resetTime();
0051           theUI.setRunningState();
0052           theState = RUNNING;
0053
0054        } else if ( theCommand.equals( "stop")) {
0055           theUI.setStoppedState();
0056           theState = STOPPED;
0057
0058        } else if ( theCommand.equals( "reset")) {
```

```
0059           theUI.setResetState();
0060           theState = RESET;
0061
0062        } else if ( theState == RUNNING) {
0063            theUI.setTime(
0064                    Integer.parseInt( theCommand, 10));
0065        } // end if.
0066     } // End actionPerformed.
```

The ActionEvents passed as arguments to this method could have originated from one of four different objects executing on two different threads of control. In order to ensure that the two different threads do not attempt to use this method at the same instant it is declared with the modifier **synchronized**. This will ensure that only one event will be processed at a time. The method itself is structured as a four-way **if/else if** structure with a branch corresponding to each possible event source.

Line 0047 retrieves the actionCommand attribute of the ActionEvent and stores it in a local String called *theCommand*. Line 0049 then determines whether this event originated from the *start* button. When the *startButton* was configured, within the *StopwatchUI* constructor, its actionCommand attribute was specified as "*start*". When the user presses the button an ActionEvent is constructed and its actionCommand initialized to that of the button, "*start*". Hence if the actionCommand retrieved from the event is "*start*" this indicates that it originated from the start button.

The steps taken when the *start* button is pressed are on lines 0050 to 0052 and cause the *Timer aTimer* to be reset, by having its *resetTime()* method called; this will cause it to start timing from zero again. The interface is then placed in the running state by having its *setRunningState()* method called, and *theState* is updated to reflect this.

The next branch in the **if** structure is associated with a press of the *stop* button, and the consequences, on lines 0055 and 0056, are to place the interface into the stopped state, using *setStoppedState()*, and to update *theState* to reflect this. Likewise, when the *reset* button is pressed the consequences, on lines 0059 and 0060, are to place the interface into the reset state, using *setResetState()*, and to update *theState* to reflect this.

If none of the three previous **if** guards evaluated true, then the ActionEvent must have originated from the *Timer* and not from any of the buttons. Its actionCommand will contain the number of tenths of a second that have elapsed, and this needs to be passed on to the interface to be shown to the user; but only if the interface is in the running state. Consequently the **if** guard on 0062 tests to see whether *theState* indicates that the interface is in the *RUNNING* state and, if so, lines 0063 and 0064 convert the actionCommand String into an **int** and passes it to *theInterface*'s *setTime()* method. This will result, as previously explained, in the elapsed time shown on the interface changing.

The consequences of this method, with the user first pressing the start button, then the stop button and finally the reset button, are illustrated in Figure 1.13.

The diagram, known as an *object interaction* diagram, can be thought of as a sequence of occurrences in time starting at the top and running towards the bottom. The *Timer*, shown on the right, is continually generating ActionEvents, but they only cause the time shown on the display to change when the artifact is in the running state, between the time when the user presses the *start* button and when the *stop* button is pressed. After pressing the *stop* button the last displayed time remains visible, allowing the user to take note of it, until the *reset* button is pressed.

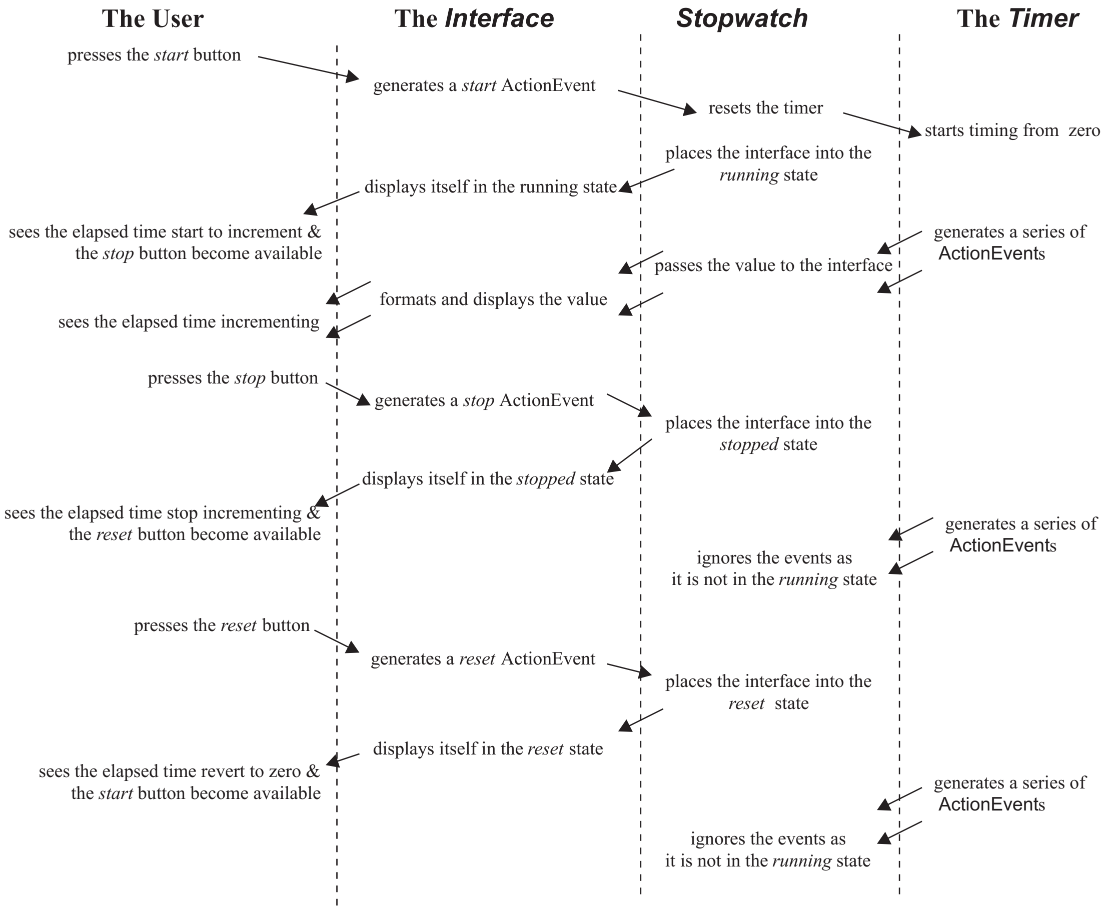

The User
The Interface
Stopwatch
The Timer
presses the start button
generates a start ActionEvent
resets the timer
starts timing from zero
places the interface into the running state
displays itself in the running state
sees the elapsed time start to increment & the stop button become available
generates a series of ActionEvents
passes the value to the interface
formats and displays the value
sees the elapsed time incrementing
presses the stop button
generates a stop ActionEvent
places the interface into the stopped state
displays itself in the stopped state
sees the elapsed time stop incrementing & the reset button become available
generates a series of ActionEvents
ignores the events as it is not in the running state
presses the reset button
generates a reset ActionEvent
places the interface into the reset state
displays itself in the reset state
sees the elapsed time revert to zero & the start button become available
generates a series of ActionEvents
ignores the events as it is not in the running state

Figure 1.13 *Stopwatch* in use.

The other thing to note about this diagram is that whenever the user does something, it starts a cycle, which results in a visible change to the appearance of the interface. This is a user interface design principle known as *closure*. Every action by users should be confirmed to them in some way, so that they can be certain that the action has been attended to and they can continue with the next action. Once they are experienced with the interface, and have confidence in its correct operation, they may no longer fully attend to the changes, and may even initiate the next action before the previous action has been confirmed. However, if they experience difficulties they will slow down and attend to the feedback provided. Hence closure is essential for novice users of an interface and useful to experienced users when communication with the artifact breaks down.

Summary

- ♦ The dispatch of an event from an event source to an event listener indicates that something has happened.

- ♦ An ActionEvent source will supply addActionListener() and removeActionListener() methods.
- ♦ If the addActionListener () method throws a TooManyListenersException, the event source is a unicaster, otherwise it is a multicaster.
- ♦ An ActionEvent listener must implement the ActionListener interface, which mandates a single method called *actionPerformed()* that takes a single ActionEvent argument.
- ♦ The actionCommand attribute of an ActionEvent can be used to pass additional information from a source to a listener.
- ♦ Artifacts that have a GUI should, in general, be implemented using a three-layer (presentation, translation and behavior) model.
- ♦ A state transition diagram is used to design the behavior of an artifact, and is subsequently implemented in the translation layer.
- ♦ A layout diagram is used to design the geometrical relationships between the components that comprise a visible interface. Subsequently, layout negotiations will position the components just before the interface becomes visible.
- ♦ An object interaction diagram can be used to show which messages are sent between objects and used to confirm that closure of every user's action will be effected.

Exercises

1.1 Reimplement the *Stopwatch* artifact omitting the **sychronized** keyword from the *Timer resetTime()* and *Stopwatch actionPerformed()* methods. Execute the artifact and show how its behavior subsequently fails.

1.2 Design, using an STD, a layout diagram and class diagrams a *LapTimer* artifact. This should provide the user with a button which when pressed will freeze the display; a second press of the button will unfreeze the display, restoring the continually updating total elapsed time display. Implement the designs and verify that the behavior of the artifact accords with the STD.

1.3 An alternative *LapTimer* artifact will contain a second *TimerLabel* instance that will continually indicate the elapsed time between two presses of the lap time button. Design and implement this version of the artifact.

1.4 Exercises 1.2 and 1.3 have indicated that the term 'lap timer' can be understood to mean different behaviors, illustrating the idiosyncratic nature of an individual's understanding. Describe, design and implement a third *LapTimer* artifact.

1.5 The javax.swing.Timer class supplies an alternative, multicasting, *Timer* class. Read the JFC documentation for the class and then redesign and reimplement the *Stopwatch* class to make use of this alternative class. Explain why the *Timer* class supplied in this chapter is, or is not, more suitable for *Stopwatch*-style artifacts.

1.6 Design and implement a *CountDownTimer* class that will dispatch a sequence of ActionEvents that contain a decrementing value in their actionCommand attributes. When an ActionEvent containing "0" in its actionCommand attribute is dispatched, no further events should be dispatched until the timer is *reset()*, with an argument indicating how many events to dispatch. Demonstrate the implementation using a window containing a single *TimerLabel* instance.

1.7 Attempt to make the *Timer* class more accurate. To do this it should have a *startTime* attribute which is initialized to the system time (obtained in milliseconds with the phrase **new** Date().getTime()). Every time the thread wakes up from its sleep it should obtain the current system time and subtract from it the *startTime*. This value, the number of milliseconds that have elapsed, should be converted into tenths of a second and the event dispatched.

2

A second artifact – consolidating fundamentals

2.1 Introduction

Chapter 1 provided a detailed introduction to the design and development of artifacts that have a graphical user interface. The intention of this chapter is to consolidate that understanding by providing a description of the design and development of a second artifact. This will involve introducing more of the JFC components and also different classes of events that are dispatched by them. The event source/listener protocol, which was emphasized in the previous chapter, will again be described, this time in the context of these different classes of event sources and listeners. The fundamental architecture, involving application, translation and presentation layers, will be reinforced, although there is little justification for such a division of responsibility in artifacts of this level of complexity. Hence the product can be argued to be over-engineered. However, if these design principles are exposed and explained in the context of these simple examples then they will be available and applicable to artifacts that merit such an approach.

The second part of this chapter will introduce the mechanisms by which user preferences for the appearance of artifacts can be customized. This is an important usability consideration as in order for an individual user to be fully comfortable when interacting with an application they must be able to change such things as the background colors, foreground colors and font used so as to suit their preferences and requirements.

Some years ago, when delivering a user interface construction course with X/Motif, one of my students was officially registered as blind but had some, very limited, sight remaining. She was able to configure her workstation to use bright yellow 60 point text on a pale blue background, a configuration that she was able to read with difficulty. Although this is an extreme example, it is a requirement of EU workplace law that such adjustments must be available to all users so that none are prevented from economic participation. Java, like X/Motif, has been designed to provide fundamental support for these accessibility issues.

2.2 The *FontViewer* artifact

The artifact that will be developed in this chapter is called *FontViewer* and is illustrated in Figure 2.1. The purpose of the artifact is to allow the user to investigate the appearance of

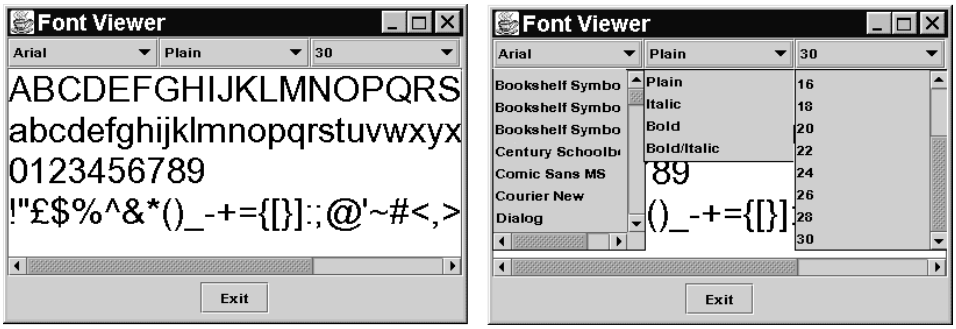


Figure 2.1 *FontViewer* artifact: physical appearance.

the fonts that are supplied by the platform that it is executing upon. Artifacts such as this have an important role in supporting accessibility, as they allow users to investigate what is available before deciding upon the font that is most suitable for them.

The left-hand illustration in Figure 2.1 shows that the artifact consists of a control panel, at the top, containing three pull-down lists allowing the font *family*, the font *style* and the font *size* to be selected. Fonts in the Java environment are defined by a combination of these three attributes, so, when any of the three controls are used, a different font will be installed for the user to view. The right-hand illustration is a simulated image showing the contents of the three lists when the artifact was executed in a Windows 95 environment.

The font families illustrated in Figure 2.1 include ***physical*** *fonts, such as Bookshelf or Courier. Java also supports* ***logical*** *fonts, called Monospaced, Serif, Sans-serif and Dialog. Logical fonts are always supported by Java and are mapped onto suitable physical fonts. Hence program code should always refer to logical fonts, which are guaranteed always to be available, and should never refer to physical fonts, which may not be available on a particular platform. This example, as will be shown, is requesting a complete list of fonts, physical and logical, from the environment it is executing within. Accordingly, the list of available fonts would be different if it was executed on a different platform.*

The currently selected font is illustrated in the middle area of the artifact, displayed as four lines of sample text showing the upper-case characters, lower-case characters, digits and common punctuation characters. It is possible, when a large font is selected, that this area will not be large enough for the entire image to be seen, and scroll bars are supplied for when this is the case. In the left-hand illustration the area is high enough to show all four lines, so no vertical scroll bar is required. However, the window is not wide enough to show the extent of all the lines, so a horizontal scroll bar is supplied. The user can interact with the scroll bar so as to adjust the *viewport*, as illustrated in Figure 2.2.

The illustration shows that the width of the area upon which the font is being illustrated is much wider than the width of the viewport that is being used to look at it; only approximately the middle half of the image can be seen. The extent and position of the

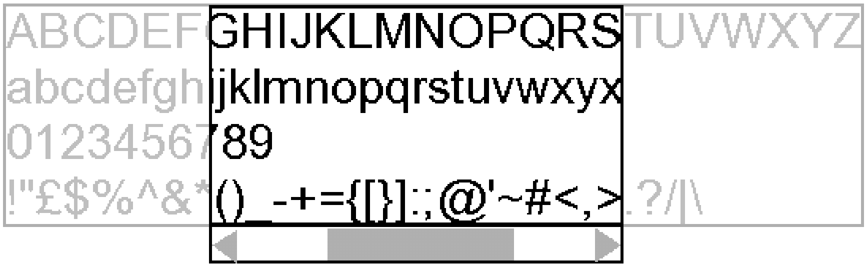
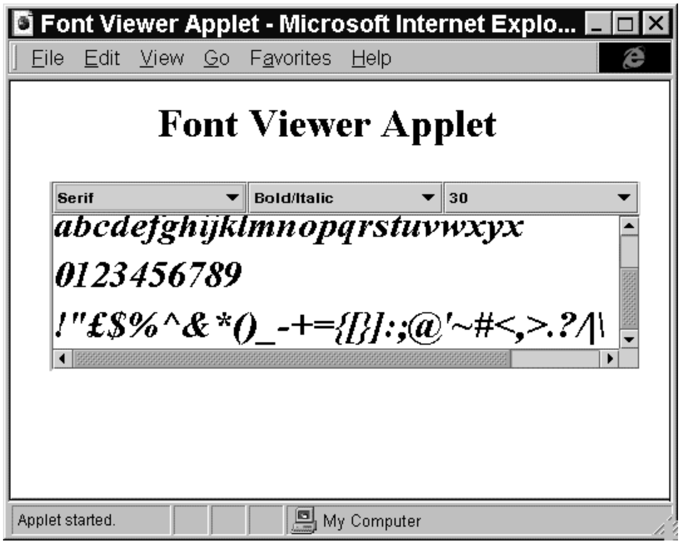


Figure 2.2 *FontViewer* artifact: viewport and horizontal scroll bar.


Font Viewer Applet - Microsoft Internet Explo...
File Edit View Go Favorites Help

Font Viewer Applet

Serif | Bold/Italic | 30

abcdefghijklmnopqrstuvwxyx
0123456789
!"£$%^&*()_-+={[}]:;@'~#<,>.?/|\

Applet started. | My Computer

Figure 2.3 *FontViewer* artifact executing within a browser.

slider within the horizontal scroll bar at the bottom is showing that approximately one quarter of the image exists to the left and to the right of the part that is visible. The user can use the controls on the scroll bar to adjust the position of the viewport, so as to bring the currently not-visible areas within the boundary of the viewport.

The lower part of the artifact contains a single button labeled *exit*, which, when pressed, will close the application removing the window from the desktop. This part of the interface is not shown when the artifact is executed within a browser, as illustrated in Figure 2.3. It is not possible to exit from an application within a browser; instead the browser's controls are used to move to another Web page when a user has finished with the artifact. Figure 2.3 also illustrates a situation where a vertical scroll bar is required.

The state transition diagram for the *FontViewer* artifact, when executing as an application, is shown in Figure 2.4. The transition from the initial state to the only application state, labeled *font viewing state*, causes the application to be posted onto the desktop showing a default font. The terminal transition is always taken when the user presses the *exit* button and results in the window being closed and removed from the desktop. Each of the other three transitions is associated with the user selecting one of the options from the drop-down lists and all result in the new font selected by the user being illustrated within the middle area.

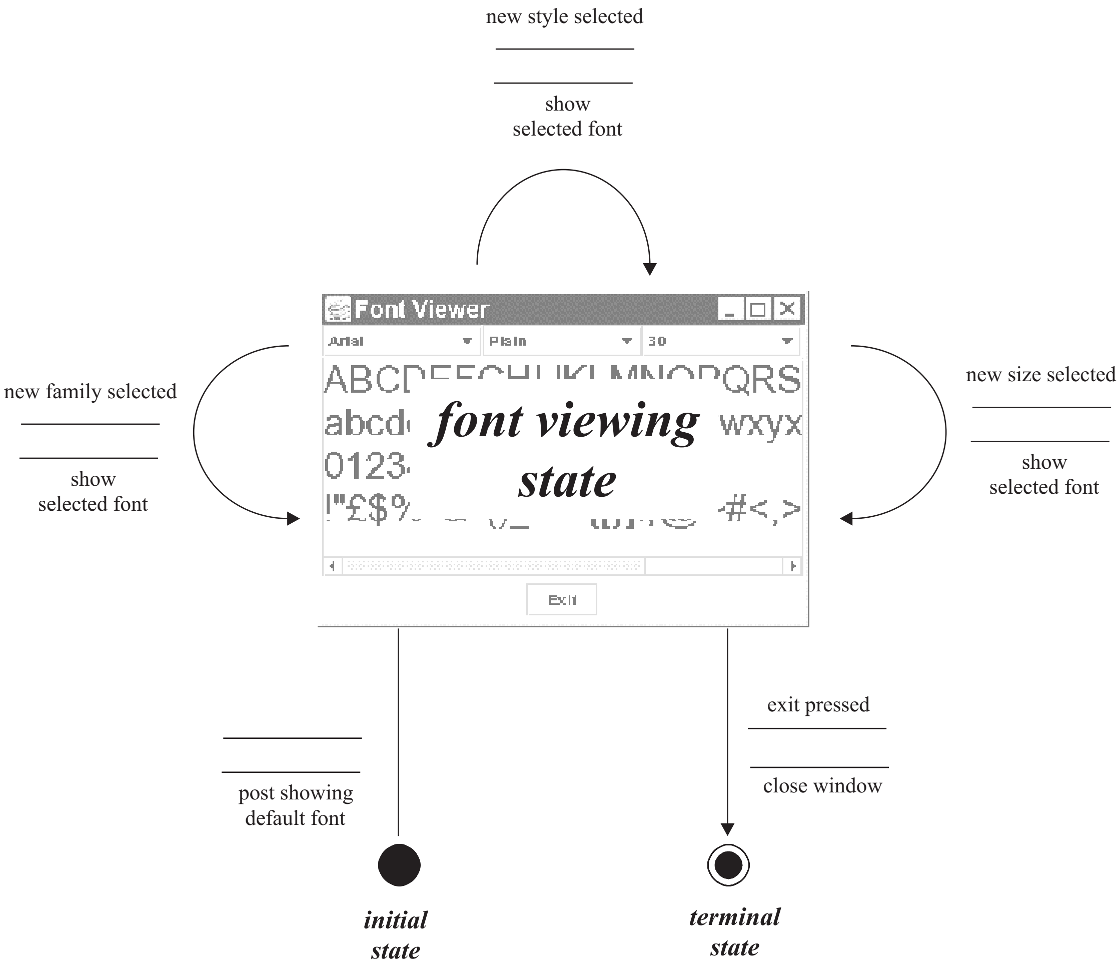


Figure 2.4 *FontViewer* artifact: state transition diagram.

The absence of pre-conditions on all of these transitions, coupled with the regularity of their effects and the single application state, allows a developer to predict at this stage that the artifact will be simple and intuitive to use. The transitions involved in showing and using the scroll bars are not shown on this diagram as they require no effort by the developer, as will be shown shortly.

2.3 *FontViewer*: design overview

An instance diagram for the *FontViewer* artifact is given in Figure 2.5. The artifact itself is an instance of the *FontViewer* class which extends the JApplet class and supplies the translation aspects of the three layer architecture. It is shown as implementing the ActionListener interface, as it has to listen to the ActionEvents from the JButton instance on *theInterface*. It is also shown as implementing the ItemListener interface as it has to listen to ItemEvents generated by the three JComboBoxes on the interface. The application level object, shown on the right, is an instance of the *FontStore* class which has a dual

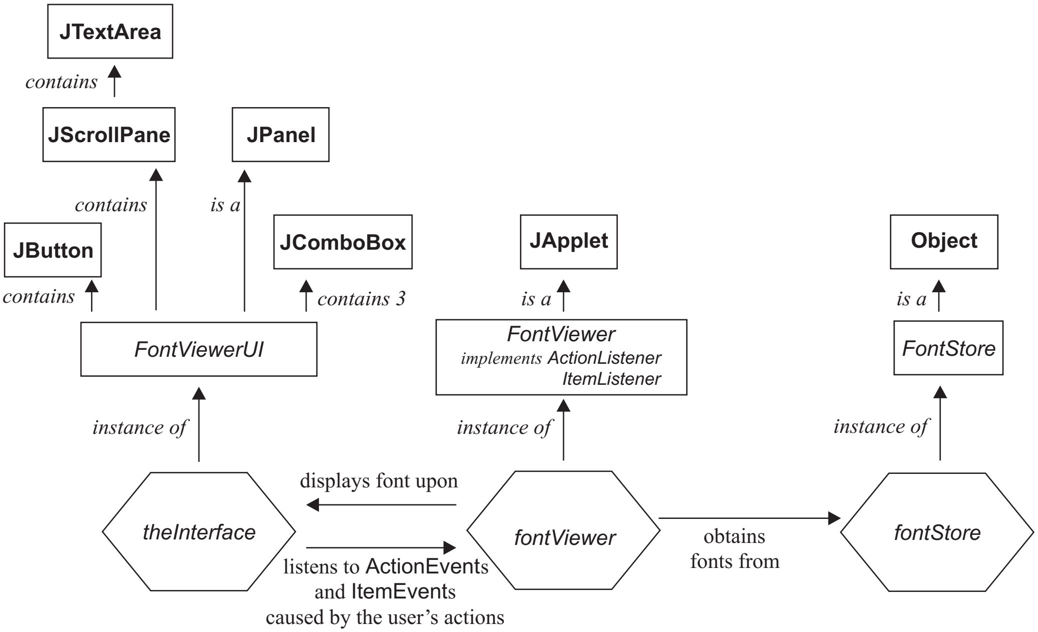


Figure 2.5 *FontViewer* artifact: instance diagram.

role of supplying information concerning which fonts are available in the environment and also constructing and supplying the Font requested by the user.

The presentation level object, *theInterface*, shown on the left, is an instance of the *FontViewerUI* class which extends the JPanel class. It contains the five components that the user will see on the interface: a JButton for the exit control; a JScrollPane which itself contains a JTextArea instance, for the middle area where the fonts are shown; and three instances of the JComboBox class to provide the three drop-down menus. The nature and characteristics of the new JFC components shown on this diagram will be described below when the *FontViewerUI* class is described.

Arguably, the most fundamental of these layers is the application layer, so the *FontStore* class will be described first.

2.4 *FontStore*: design and implementation

The *FontStore* class supplies the application level functionality in this artifact and has a dual role; its class diagram is given in Figure 2.6. Its first role is to supply a list of the font *families* available in the environment that the artifact is operating within. This list, called *familyNames*, will be requested by the translation object and passed to the presentation object where it will be used to initialize the options presented on the leftmost drop-down list. It also nominates one of these families to be the *defaultFamily.* These requirements are supported by the encapsulated *familyNames[]* and *defaultFamily* attributes and by the *getFamilyNames()* and *getDefaultFamily()* methods. As the contents of the list will vary, depending upon the platform that the artifact is executing upon, all of these are supplied as instance attributes and methods.

The *styleNames[]*, *defaultStyle*, *getStyleNames[]* and *getDefaultStyle()* resources are comparable to the *family* resources described above, but refer to the available *styles*.

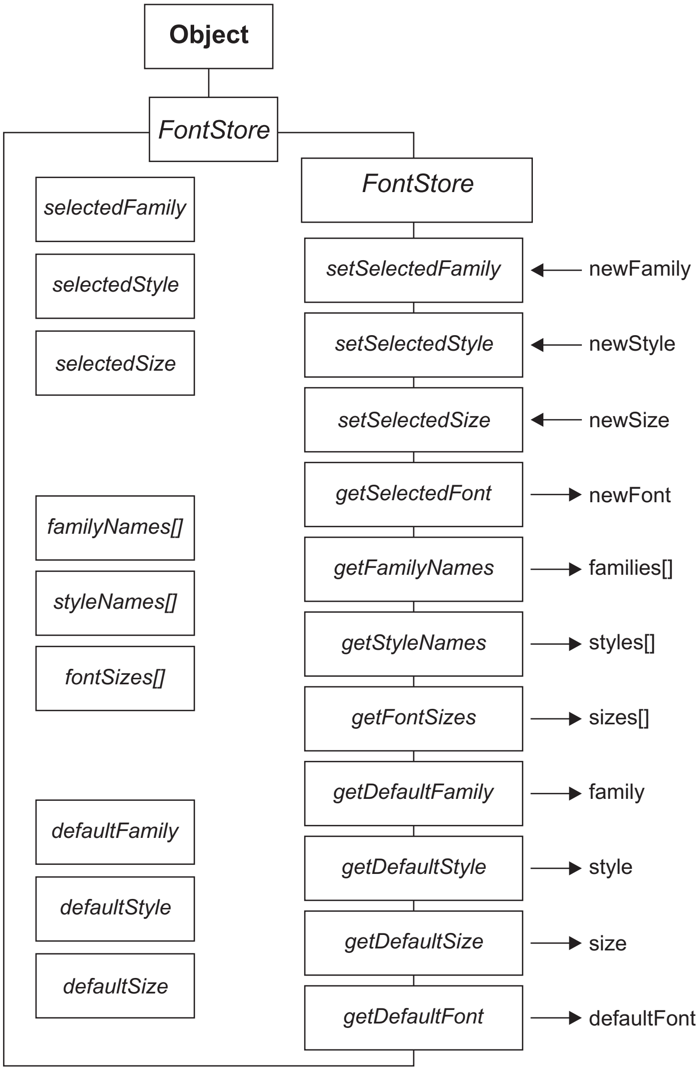


Figure 2.6 *FontStore* artifact: class diagram.

Likewise the *fontSizes[]*, *defaultSize*, *getFontSizes()* and *getDefaultFontSize()* resources support a list of font *sizes*. These two sets of resources do not change, depending upon the platform, and so are supplied as class-wide resources. The *getDefaultFont()* method will return a Font object reflecting the values of the three default resources. This method will be used to supply a Font to the interface when it first becomes visible to the user.

The second role of this class is to store details of the font that has been currently selected by the user. This requirement is supported by the *selectedFamily*, *setSelectedFamily()*, *selectedStyle*, *setSelectedStyle()*, *selectedSize* and *setSelectedSize()* resources. The *getSelectedFont()* method will return a Font corresponding to the values indicated.

The first part of the implementation of this class, as far as the start of the constructor, is as follows.

```
0010  package fontviewer;
0011
0012  import java.awt.*;
0013
0014  public class FontStore extends Object {
0015
0016  private static final String[] styleNames = {
0017                  "Plain", "Italic", "Bold", "Bold/Italic"};
0018  private static final String defaultStyle = styleNames[ 0];
0019
0020  private static final String[] fontSizes = {
0021                         "8",  "10", "12", "14", "16", "18",
0022                         "20", "22", "24", "26", "28", "30"};
0023  private static final String defaultSize = "20";
0024
0025  private String[] familyNames   = null;
0026  private String   defaultFamily = null;
0027
0028  private String selectedFamily = null;
0029  private String selectedStyle  = defaultStyle;
0030  private String selectedSize   = defaultFontSize;
```

The class commences by stating that it is a member of the *fontstore* package of classes, and the class declaration, on line 0014, is exactly as shown on the class diagram. Lines 0016 to 0023 then declare the class-wide attributes from the class diagram, initializing them as appropriate. These two lists will be requested from the *fontStore*, application, object by the translation object and used to initialize the contents of the middle and rightmost drop-down menu, as shown in Figure 2.1. The list of available styles is determined by the styles supported by Java, which does not include underline. The list of sizes is arbitrary, and a technique to extend the artifact to allow the user to select any size of font will be introduced later in this chapter.

The two instance attributes associated with the first role of the class are then declared, but not initialized, on lines 0025 and 0026. The three *selected* attributes are declared on lines 0028 to 0030 with the *selectedStyle* and *selectedSize* instances initialized to the corresponding *default* class-wide attributes. The constructor is implemented as follows.

```
0035     public FontStore() {
0036        super();
0037        GraphicsEnvironment environment =
0038           GraphicsEnvironment.getLocalGraphicsEnvironment();
0039        familyNames = environment.
0040                                  getAvailableFontFamilyNames();
0041        defaultFamily  = familyNames[ 0];
0042        selectedFamily = defaultFamily;
0043     } // End FontStore constructor.
```

Object → java.awt.GraphicsEnvironment

The GraphicsEnvironment object allows an artifact to obtain information about the various GraphicsDevices (screen(s), printer(s), image buffer(s)) that might be available on the platform it is executing upon.

```
public static GraphicsEnvironment getLocalGraphicsEnvironment()
```

Obtains the identity of the local (default) graphics output device.

```
public Font[] getAllFonts()
public String[] getAvailableFontFamilyNames()
```

Obtains a list of the Fonts available, or just the names of the families.

The constructor commences by calling its **super**, Object, constructor and then declares and initializes an instance of the GraphicsEnvironment class called *environment*. Instances of the GraphicsEnvironment class encapsulate information concerning the graphical capabilities of the environment that the artifact is executing upon. The *environment*'s getAvailableFontFamilyNames() method is called, on lines 0039 to 0040, to obtain a list of the font families available, and the list is used to initialize the *familyNames* String array attribute. Having obtained the list of available families the *defaultFamily* and *selectedFamily* attributes can be initialized from it, on lines 0041 and 0042.

The three *selected* state setting methods are implemented as expected, as follows, and the *getSelectedFont()* is implemented as a call of the **private** *makeFont()* method, which will be described shortly.

```
0049     public void setSelectedFamily( String newFamily) {
0050        selectedFamily = new String( newFamily);
0051     } // End setSelectedFamily.
0052
0053     public void setSelectedStyle( String newStyle) {
0054        selectedStyle = new String( newStyle);
0055     } // End setSelectedStyle.
0056
0057     public void setSelectedSize( String newSize) {
0058        selectedSize = new String( newSize);
0059     } // End setSelectedSize.
0060
0061     public Font getSelectedFont() {
0062        return this.makeFont( selectedFamily,
0063                              selectedStyle,
0064                              selectedSize);
0065     } // End getSelectedFont
```

The implementation continues with the six inquiry methods associated with the remaining six attributes, as follows.

```
0068     public String[] getFamilyNames() {
0069        return familyNames;
0070     } // End getFamilyNames.
0071
0072     public String getDefaultFamily() {
0073        return defaultFamily;
0074     } // End getDefaultFamily;
0075
0076
0077     public static String[] getStyleNames() {
0078        return styleNames;
0079     } // End getStyleNames.
0080
0081     public static String getDefaultStyle() {
0082        return defaultStyle;
0083     } // End getDefaultStyle;
0084
0085
0086     public static String[] getFontSizes() {
0087        return fontSizes;
0088     } // End getFontSizes.
0089
0090     public static String getDefaultSize() {
0091        return defaultSize;
0092     } // End getDefaultSize.
0093
0094
0095     public Font getDefaultFont() {
0096        return this.makeFont(
0097                    this.getDefaultFamily(),
0097                    this.getDefaultStyle(),
0098                    this.getDefaultSize());
0099     } // End getDefaultFont.
```

This part of the implementation concludes with the *getDefaultFont()* method, which, like the *getSelectedFont()* method above, indirects to the **private** *makeFont()* method whose implementation, as follows, concludes the class's implementation.

```
0102     private static Font makeFont( String requiredFamily,
0103                                   String requiredStyle,
0104                                   String requiredSize) {
0105
0106     int  fontSize     = Integer.parseInt( requiredSize, 10);
0107     int  fontStyle    = Font.PLAIN;
0108
0109         if ( requiredStyle.equals( styleNames[1])) {
0110            fontStyle = Font.ITALIC;
0111         } else if ( requiredStyle.equals( styleNames[2])) {
0112            fontStyle = Font.BOLD;
0113         } else if ( requiredStyle.equals( styleNames[3])) {
0114            fontStyle = Font.BOLD | Font.ITALIC;
0115         } // End if.
0116
0117         return new Font( requiredFamily,
0118                          fontStyle, fontSize);
0119     } // End makeFont.
0120
0121  } // End class FontStore.
```

The three attributes that define a Font, its family, style and size, are passed to this method in the three arguments as Strings. A String is only appropriate for the *family* argument to the Font constructor, and the other two arguments must be converted into **int**eger values. The *requiredSize* argument is converted directly into an **int** when the local variable *fontSize* is declared on line 0106.

Object → java.awt.Font

```
public Font( String family, int style, int size)
```

Constructs a Font instance of the *family* specified, which should be one of the logical font family names: "*Monospaced*", "*Serif*", "*Sans serif*", "*Dialog*" or "*Dialog input*". The *style* argument is a bitwise ored (|) combination of Font.PLAIN, Font.BOLD and Font.ITALIC. The *size* argument can be arbitary and the platform will either supply a scaled font to the size requested or the nearest available size.

```
public String getFamily()
public int    getStyle()
public int    getSize()
```

Enquiry methods for the three defining attributes.

The conversion of the *requiredStyle* String is more complex, as font styles are described by manifest values supplied by the Font class. The local variable *fontStyle* is declared, on line 0107, with the default value Font.PLAIN. If the *requiredStyle* is not "*Plain*", then the three-way selection, on lines 0109 to 0115, will change it as appropriate. Once this has been accomplished a new Font instance is constructed, passing the three required arguments, and **return**ed on lines 0117 and 0118.

2.5 *FontViewerUI*: design and implementation

The *FontViewerUI* class will be described before the *FontViewer* class, as *FontViewer* makes use of *FontViewerUI* and thus cannot be fully understood before understanding the classes it makes use of. As with the *StopwatchUI* class from Chapter 1, a major part of the implementation of this class will be involved with the construction and configuration of the components that it presents to the user and also with establishing the geometrical relationships between the components. Accordingly, the design phase will commence with the layout diagram, as shown in Figure 2.7.

The *fontViewerUI* is an extended JPanel that requires a BorderLayout with its North, Center and South locations occupied. The North location contains an intermediate JPanel, called *fontPanel*, whose layout manager controls the positioning of the three JComboBox instances. A three column by one row GridLayout manager has been chosen, as this will force all three controls to be of equal width. The alternative would have been the default FlowLayout manager, but as this would allow the three controls to be of different widths it was not felt to be as appropriate. The South location also contains an intermediate JPanel, called *exitPanel* whose default FlowLayout manager controls the positioning of the *exitButton*. The CENTER location is occupied by an instance of the JScrollPane class, called *fontPane*, whose *viewport* contains an instance of the JTextArea class called *fontArea*. The six visible components are shown as encapsulated attributes of the *FontViewerUI* class in the class diagram given in Figure 2.8.

The class diagram shows that the **protected** constructor takes a number of arguments. Two of these identify the *listeners*, one for the ActionEvents generated by the *exitButton* and one for the ItemEvents generated by the JComboBoxes. The remaining six arguments define the contents and the default selection of each of the three JComboBoxes. The constructor is **protected** as it is only ever intended to be called from within the *fontviewer* package, by an instance of *FontViewer*. The *setFontToShow()* method is also **protected** for the same reason; this method takes a Font *toShow* as an argument and causes the

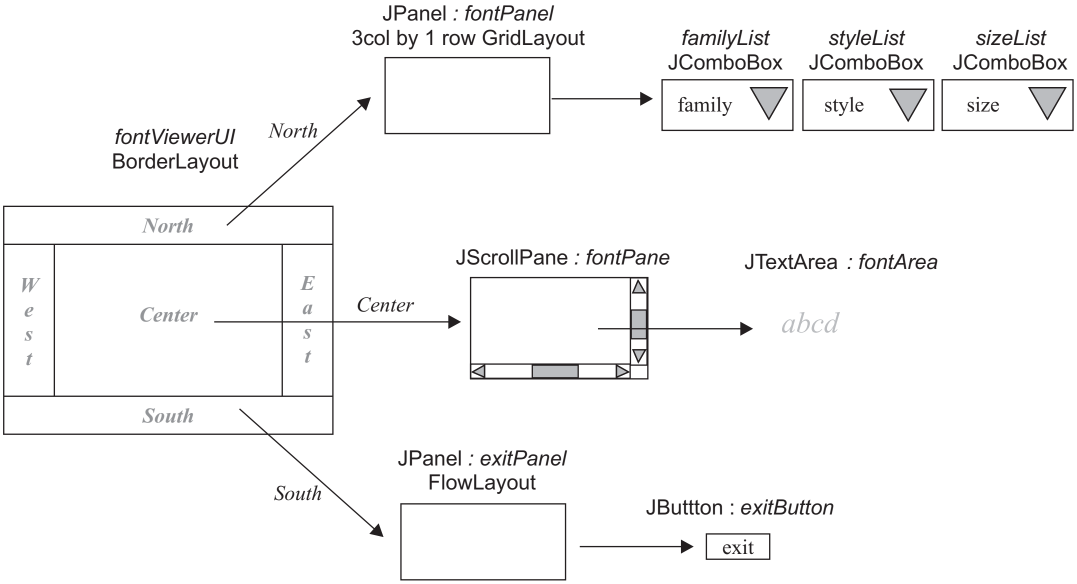


Figure 2.7 *FontViewerUI* layout diagram.

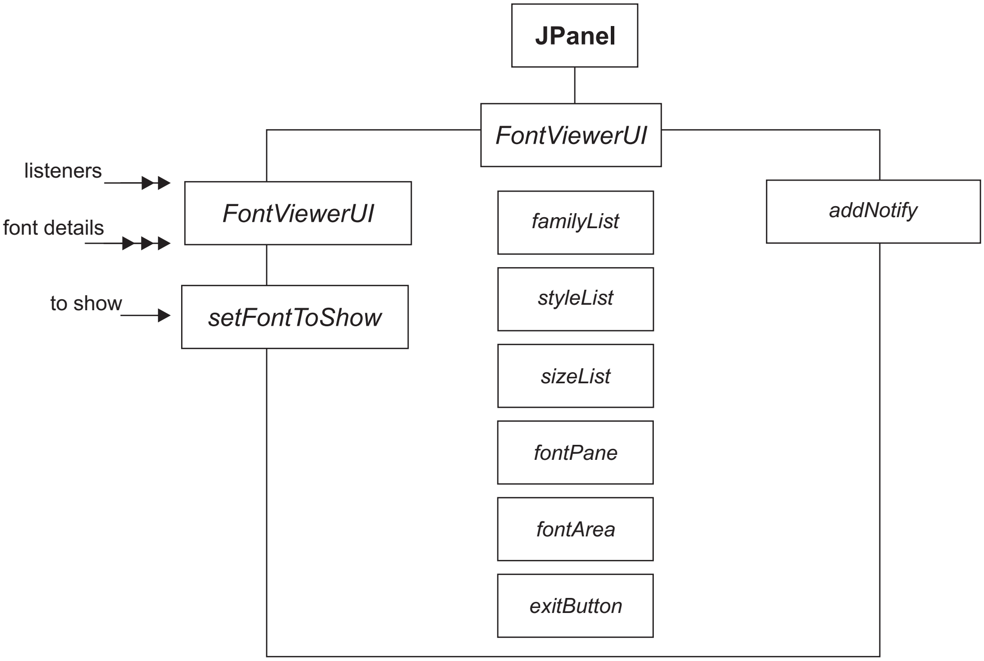


Figure 2.8 *FontViewerUI* class diagram.

contents of the *fontArea* to be redrawn using the new font. The only other method, *addNotify()*, overrides the **public** addNotify() method from Component and so must also be declared **public**, as an overriding method cannot have a more restricted visibility than the method it is overriding. The reason why this method has to be overridden will be explained below, when its implementation is described.

The implementation of this class, as far as the constructor, is as follows.

```
0010  package fontviewer;
0011
0012  import javax.swing.*;
0013
0014  import java.awt.*;
0015  import java.awt.event.*;
0016  import java.applet.*;
0017
0018  class FontViewerUI extends JPanel {
0019
0020  private JComboBox   familyList  = null;
0021  private JComboBox   styleList   = null;
0022  private JComboBox   sizeList    = null;
0023  private JPanel      fontPanel   = null;
0024
0025  private JScrollPane fontPane    = null;
0026  private JTextArea   fontArea    = null;
0027
0028  private JButton     exitButton  = null;
0029  private JPanel      exitPanel   = null;
0030
0031  private static final String sampleText =
0032                 "ABCDEFGHIJKLMNOPQRSTUVWXYZ\n" +
0033                 "abcdefghijklmnopqrstuvwxyx\n" +
0034                 "0123456789\n"                 +
0035                 "!\"£$%^&*()_-+={[}]:;@'~#<,>.?/|\\";
```

After declaring that this class is a member of the *fontviewer* package of classes and the necessary **import**ations, the class is declared, without **public** visibility, on line 0018. The components identified on the layout diagram are declared on lines 0020 to 0029 and the *sampleText* that is to be shown is declared as a class-wide String constant on lines 0031 to 0035. The implementation of the constructor is as follows.

```
0037     protected FontViewerUI( ActionListener sendActionsHere,
0038                             ItemListener   sendItemsHere,
0039                             String[]       namesToShow,
0040                             String         selectedName,
0041                             String[]       stylesToShow,
0042                             String         selectedStyle,
0043                             String[]       sizesToShow,
0044                             String         selectedSize){
0045
0046        super();
0047        this.setLayout( new BorderLayout());
0048
0049        familyList = new JComboBox( namesToShow);
0050        familyList.setSelectedItem( selectedName);
0051        familyList.addItemListener( sendItemsHere);
0052        familyList.setName( "family");
0053
0054        styleList  = new JComboBox( stylesToShow);
0055        styleList.setSelectedItem(  selectedStyle);
0056        styleList.addItemListener(  sendItemsHere);
0057        styleList.setName( "style");
0058
0059        sizeList   = new JComboBox( sizesToShow);
0060        sizeList.setSelectedItem(   selectedSize);
0061        sizeList.addItemListener(   sendItemsHere);
0062        sizeList.setName( "size");
0063
0064        fontPanel = new JPanel();
0065        fontPanel.setLayout( new GridLayout( 1,3));
0066        fontPanel.add( familyList);
0067        fontPanel.add( styleList);
0068        fontPanel.add( sizeList);
0069
0070
0071        fontArea = new JTextArea( sampleText);
0072        fontArea.setEditable( false);
0073        fontPane = new JScrollPane(
0074                JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
0075                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
0076        fontPane.getViewport().add( fontArea);
0077        fontPane.setMinimumSize(   new Dimension( 150, 100));
0078        fontPane.setPreferredSize( new Dimension( 400, 200));
0079
0080
0081        exitPanel  = new JPanel();
0082        exitButton = new JButton(    "Exit");
0083        exitButton.setActionCommand( "exit");
0084        exitButton.addActionListener( sendActionsHere);
0085        exitPanel.add( exitButton);
0086
0087
0088        this.add( fontPanel, BorderLayout.NORTH);
0089        this.add( fontPane,  BorderLayout.CENTER);
0090        this.add( exitPanel, BorderLayout.SOUTH);
0091     } // End FontViewerUI constructor.
```

The constructor, declared on lines 0037 to 0044, has eight arguments. The first two of these, *sendActionsHere* and *sendItemsHere*, are of the appropriate listener interface types to be installed as the *exitButton*'s actionListener resource and as each JComboBoxes itemListener resource, respectively. The remaining six arguments can be thought of as three groups of two arguments, specifying the list of items that are to be installed in the appropriate JComboBox and the item from the list that is to be selected by default. For example, the *sizesToShow* String array contains the list of font sizes to be shown on the rightmost of the controls and the *selectedSize* argument indicates which of them is to be selected by default when the interface first becomes visible to the user.

The first step of the constructor, on lines 0046 and 0047, is to call the super, JPanel, constructor and to install the required BorderLayout, replacing the default FlowLayout, manager. The next part of the constructor, between lines 0049 and 0062, consists of three very similar stages, each of which constructs and configures one of the JComboBoxes. Line 0049 constructs the *familyList*, the argument to the constructor indicating the list of Strings that it is to show on its drop-down menu. Line 0050 then uses the setSelectedItem() method of the *familyList* to indicate which of the items on the list is to be selected and displayed, even when the drop-down list is not posted.

Line 0051 calls the *familyList* addItemListener() method which is comparable to the JButton addActionListener() method, as described in the previous chapter. The argument to this method, one of the constructor's arguments, identifies the object to which the *familyList* is to dispatch *ItemEvent*s as the user interacts with the control. More details will be given of ItemEvents in the next section, which describes the *FontViewer* class.

The final step in this stage of the constructor is to set a non-default value for the name attribute of the JComboBox. All Components have a name attribute; by default this would be, in this example, something like "*JComboBox23*". There are three JComboBoxes in this interface and all three of them will be dispatching ItemEvents to the same listener. As JComboBoxes do not have an actionCommand attribute the technique used in the *Stopwatch* example to identify the source of an event cannot be used, so a more general technique, using the name attribute, is being used. The description of the method that receives the ItemEvents in the *FontViewer* class will show how this technique is effected.

JComponent → javax.swing.JComboBox

```
public JComboBox( Object[] items)
```

Constructs a JComboBox instance displaying the list of *items* supplied on its drop-down menu with the first item *selected* by default.

```
public void    addItemListener()
public void    removeItemListener()
public void    addActionListener()
public void    removeActionListener()
public void    setActionCommand()
public String  getActionCommand()
public boolean isEditable()
public void    setEditable( yesOrNo)
```

ItemEvents are generated when a different selection is made from the drop-down list and also, if the combo box is editable, when a new item is entered. ActionEvents are also generated by an editable combo box when the user uses the editor, with the actionCommand if specified.

```
public void   addItem( Object toAdd)
public void   insertItemAt( Object toInsert, int location)
public void   removeItem( Object toRemove)
public void   removeItemAt( int  location)
public void   removeAllItems()
public Object getItemAt( int location)
public int    getItemCount()
public int    getMaximumRowCount()
```

Methods to count, add to, remove from or obtain items from the list. The maximumRowCount attribute is the number of items that can be shown on the drop-down menu without requiring a vertical scroll bar.

```
public Object getSelectedItem()
public void   setSelectedIndex( int location)
public void   setSelectedItem( Object toSelect)
```

Methods to obtain, or set, the selected item.

The construction and configuration of the two remaining JComboBoxes, the *styleList* on lines 0054 to 0057 and the *sizeList* on lines 0059 to 0062, are essentially identical and will not be commented upon further. Once all three of these controls have been prepared, line 0064 constructs the JPanel, *fontPanel*, that they will be mounted upon; and line 0065 installs a one row by three column GridLayout manager into this panel.

A GridLayout manager can accommodate an indefinite number of children laid out in an equal-sized grid, optimally determined by the widest and highest of the children. The sequence in which children are added to the GidLayout's associated Container determines where they will appear, with the grid being populated in a left/right, top/down manner. Lines 0066 to 0068 add the three JComboBoxes to the *fontPanel* in a sequence that will produce the appearance shown at the top of Figure 2.1.

Object → java.awt.GridLayout

```
public Gridlayout( int rows, int columns)
public Gridlayout( int rows, int columns, int hgap int vgap)
```

Constructs a GridLayout instance capable of laying out children in a grid with the number of *rows* and *columns* specified, with a horizontal (*hgap*) and vertical gap (*vgap*) between the cells if specified. The children will be added in a left/right, top/down manner and all will be forced to be the same size.

```
public void setRows( int newRows)
public int  getRows()
public void setColumns( int newColumns)
public int  getColumns()
public void setHgap(int newGap)
public int  getHgap()
public void setVgap(int newGap)
public int  getHgap()
```

State setting and inquiry methods for the four defining attributes.

The next part of the constructor, between lines 0071 and 0077, constructs and configures the middle part of the interface. Line 0071 constructs the *fontArea* as an instance of the JTextArea class. A JTextArea instance is capable of storing and showing a multi-line String. The argument to the constructor, *sampleText*, is the four-line String declared on lines 0031 to 0035 and defines the text that the instance will display. By default a JTextArea is editable, allowing the user to change the contents, and as this is not appropriate in this artifact it is set non-editable on line 0072.

JComponent → JTextComponent → javax.swing.JTextArea

```
public JTextArea ()
public JTextArea ( String toShow)
public JTextArea ( int rows, int columns)
public JTextArea ( String toShow, int rows, int columns)
```

Constructs a JTextArea instance with various combinations of the String *toShow* and the (approximate) number of characters in a *row* and the number of *columns*.

```
public int  getColumns()
public void setColumns( int newColumns)
public int  getRows()
public void setRows( int newRows)
```

Inquiry and setting methods for the number of *rows* and *columns*.

```
public void append( String toAppend)
public void insert( String toInsert, int offset)
public int  replaceRange( String replaceWith
                          int fromHere, int toHere)
public String getText()
public String getText( int fromHere, int toHere)
```

Methods to manipulate the *text* contained in the component. The getText() methods are inherited from JTextComponent.

```
public boolean getEditable()
public boolean setEditable( boolean yesOrNo)
```

The *editable* attribute determines whether the user can change the text from the keyboard; default **true**.

```
public int getLineCount()
public int getLineStartOffset( int lineNumber)
public int getLineEndOffset( int lineNumber)
public int getLineOfOffset( int offset)
public boolean getLineWrap()
public void    setLineWrap( boolean yesOrNo)
public boolean getWrapStyleWord()
public void    setWrapStyleWord( boolean yesOrNo)
```

The text is considered as a sequence of lines. These methods allow the number of lines and their start and end locations within the text to be determined. The *lineWrap* attribute determines whether lines that are wider than the width of the window are broken and continued on the next line (default **false**). The *wrapStyleWord* attribute works in concert to break lines at word boundaries.

Lines 0073 to 0075 construct an instance of the JScrollPane class called *frontPane*. The arguments to the constructor indicate that the JScrollPane instance should automatically supply the horizontal and vertical scroll bars whenever they are needed and should not display them if they are not needed. A JScrollPane instance is a Container that can have only one child, which is added to its viewport. This child is then shown within the component's window and the scroll bars can be used to *pan* the viewport across the underlying child if it is larger than the viewport's window. Accordingly, line 0076 adds the *fontArea* to the *fontPane*'s viewport, implementing the relationship illustrated in Figure 2.2. This relationship, where the scroll bars are connected to control the view of the *fontArea*, is supplied automatically by the *fontPane* and does not need any effort by the developer to effect it. The final stage in the configuration of the *fontPane*, on lines 0077 and 0078, is to establish a minimum and preferred size for it. These resources, instances of the AWT Dimension class, will be used when layout negotiations take place and specify a reasonable size given the nature of this artifact.

JComponent → javax.swing.JScrollPane

```
public JScrollPane()
public JScrollPane( Component toShow)
public JScrollPane( int verticalpolicy, int horizontalPolicy)
public JScrollPane(Component toShow ,
                   int verticalpolicy, int horizontalPolicy)
```

Constructs a JScrollPane instance with various combinations of the component in the viewport and specified scroll bar policies: VERTICAL_SCROLLBAR_AS_NEEDED (default), VERTICAL_SCROLLBAR_NEVER and VERTICAL_SCROLLBAR_ALWAYS (and corresponding for HORIZONTAL).

```
public JScrollBar getHorizontalScrollBar()
public JScrollBar getVerticalScrollBar()
public JScrollBar getViewport()
```

Inquiry methods for the components of the JScrollPane.

```
public int  getHorizontalScrollBarPolicy()
public void setHorizontalScrollBarPolicy( int newPolicy)
public int  getVerticalScrollBarPolicy() ()
public void setVerticalScrollBarPolicy(   int newPolicy)
```

Inquiry and setting methods for the scroll bar policies.

The constructor continues, on lines 0081 to 0085, by constructing and configuring an instance of the JButton class, called *exitButton*, and mounting it upon its own JPanel, called *exitPanel*. These techniques do not differ from those used to construct and configure the JButtons on the *Stopwatch* interface, as described in the previous chapter. As there is only one button on this interface there can be no confusion regarding where ActionEvents originated, so the actionCommand attribute of the *exitButton* need not be set.

The final step in the *FontViewerUI* constructor adds the three child components to **this** instance of the *FontViewerUI* instance being constructed. This stage, and the previous stages where components have been added, implement the layout requirements established in Figure 2.7. Consequently, when the artifact becomes visible to the user, assuming that it is being executed as an application, it will appear as shown in Figure 2.1. The implementation of the *setFontToShow()* method is as follows.

```
0092     protected void setFontToShow( Font toShow) {
0093        fontArea.setFont( toShow);
0094     } // End setFontToShow
```

This is a wrapper method which passes its Font argument, *toShow*, directly on to the *fontArea* setText() method inherited from JTextArea. This will cause the component to install the new font and re-render the text it contains using the new font. The only remaining method is *addNotify()*; this method is called by the Java environment before the component becomes visible, but immediately after its parent component has been created. It is only at this stage that the *FontViewerUI* instance can determine whether it is being used within an applet or within a standalone application. As illustrated in Figure 2.3, and explained above, in this configuration the *exit* button should be suppressed, as it is not possible to exit an artifact when it is running within an applet.

Consequently, the *FontViewerUI addNotify()* method has to decide whether it is executing within a browser and, if so, remove the *exitButton* from the interface. It accomplishes this by ascending the layout window hierarchy until it locates an instance of the Applet class (which includes JApplets) and can then ask it to supply an AppletContext. An AppletContext encapsulates information about the browser environment that it is operating within. If the artifact is executing as an application then this attempt to obtain an AppletContext will throw an exception.

It is vitally important that an overriding *addNotify()* method calls the overridden addNotify() method, either at the start or at the end, otherwise the interface will not become visible. In this example the overridden addNotify() method should be called at the end of the method so that layout negotiations do not include the *exitButton*, in case it has to be removed. The implementation of the *FontViewerUI addNotify()* method, which completes the class, is as follows.

```
0097     public void addNotify() {
0098
0099     Component     itsParent  = this;
0100     AppletContext itsContext = null;
0101
0102        while ( ! (itsParent instanceof Applet)) {
0103           itsParent = itsParent.getParent();
0104        } // End while.
0105
0106        try {
0107           itsContext =
0108                     ((Applet) itsParent).getAppletContext();
0109           this.remove( exitPanel);
0110        } catch ( NullPointerException exception) {
0111           // do nothing.
0112        } // end try/catch.
0113        super.addNotify();
0114     } // End addNotify.
0115
0116  } // End class FontViewerUI.
```

The method commences, on line 0102 to 0104, by ascending the layout window hierarchy until the Applet instance is located. This is accomplished by using the AWT component getParent() method, which returns the identity of the Container to which a component has been added. At the conclusion of this loop the *itsParent* local variable contains the identity of the artifact's Applet, and on lines 0107 and 0108 an attempt is made to obtain its AppletContext by calling the Applet getAppletContext() method. This method will throw a

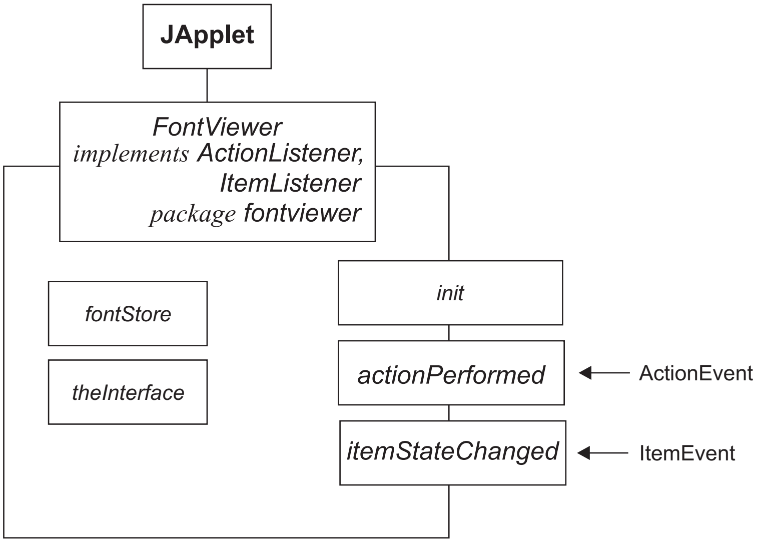


Figure 2.9 *FontViewerUI* class diagram.

NullPointerException if the artifact is executing as an application and so must be contained within a **try/catch** structure.

If the exception is not thrown then the artifact is executing as an applet, so, on line 0109, the *exitPanel* is removed from **this** instance of *FontViewerUI*. The remove() method can be used to remove a component that has previously been added with add(). If an exception is thrown then the artifact is executing as an application and the *exitPanel* should not be removed; accordingly the **catch** clause, on line 0111, does nothing. In all circumstances the **super** addNotify() method must be called, thus concluding the method.

2.6 *FontViewer*: design and implementation

The class diagram for the *FontViewer* class is given in Figure 2.9. It shows that the *FontViewer* class extends the JApplet class, is a member of the *fontviewer* package of classes and implements the ActionListener and ItemListener interfaces. It has two encapsulated attributes: an instance of *FontStore*, called *fontStore*, and an instance of *FontViewerUI*, called *theInterface*.

As an extended JApplet it has an *init()* method instead of a constructor. The ActionListener interface mandates an *actionPerformed()* method, to which ActionEvents will be sent to be processed. The ItemListener interface mandates a method called *itemStateChanged()*, which takes a single argument of the ItemEvent class. As with the ActionListener protocols, only if a class states that it implements the ItemListener interface can it be registered with an ItemEvent source as its itemListener resource. When the ItemEvent source generates and dispatches an ItemEvent it is passed as an argument to the *itemStateChanged()* method, which is guaranteed to exist as it is mandated by the ItemListener interface.

java.awt.event.ItemListener interface

```
public void itemStateChanged( ItemEvent event)
```

Single method mandated by the ItemListener **interface**.

Object → EventObject → AWTEvent → java.awt.event.ItemEvent

```
public ItemEvent( int source,  int id,
                  Object item, int stateChange)
```

Constructor specifying the source of the event and the item which was selected or deselected causing it to be dispatched, the manifest reason for dispatching the event (id) and the stateChange (ItemEvent.SELECTED or ItemEvent.DESELECTED).

```
public Object         getSource()
public ItemSelectable getItemSelectable()
public int            getID()
public Object         getItem()
public int            getStateChange()
```

Inquiry methods for the attributes of the event. ItemSelectable is an interface that mandates certain methods for components that generate ItemEvents.

The implementation of the *FontViewer* class as far as the start of the *init()* method is as follows.

```
0010  package fontviewer;
0011
0012  import javax.swing.*;
0013
0014  import java.awt.*;
0015  import java.awt.event.*;
0016
0017  public class FontViewer extends    JApplet
0018                          implements ActionListener,
0019                                     ItemListener  {
0020
0021  private FontViewerUI theInterface = null;
0022  private FontStore    fontStore    = null;
```

The class is declared as a member of the *fontviewer* package of classes and, following the necessary **import**ations, the class is declared exactly as indicated on its class diagram. Lines 0021 and 0022 then declare, but do not construct, the two encapsulated attributes. The implementation of the *init()* method is as follows.

```
0025     public void init() {
0026
0027     String[] styleNames   = FontStore.getStyleNames();
0028     String   defaultStyle = FontStore.getDefaultStyle();
0029     String[] fontSizes    = FontStore.getFontSizes();
0030     String   defaultSize  = FontStore.getDefaultSize();
0031
0032        fontStore = new FontStore();
0033
0034        theInterface = new FontViewerUI( this, this,
0035                                fontStore.getFamilyNames(),
0036                                fontStore.getDefaultFamily(),
0037                                styleNames,  defaultStyle,
0038                                fontSizes,   defaultSize);
0039
0040        theInterface.setFontToShow(
0041                                  fontStore.getDefaultFont());
0042        this.getContentPane().add( theInterface);
0043     } // End init.
```

Lines 0027 to 0030 call the static methods of the *FontStore* class to obtain the list of styles and font sizes, together with the default values, storing them in appropriately named variables. Line 0032 then constructs an instance of the *FontStore* class, storing its reference in the *fontStore* attribute. The next stage, on lines 0034 to 0038, constructs an instance of the *FontViewerUI* class called *theInterface*. The first two arguments are the identity of the object to dispatch ActionEvents and ItemEvents to, and both of these are specified as **this** instance of *FontViewer* currently being constructed. The next two arguments are the list of font families and the family to be selected by default. These are obtained by calling the appropriate instance method from the *fontStore* instance. The remaining four arguments are the previously initialized local variables, specifying details of the font styles and sizes.

Once *theInterface* has been constructed, lines 0040 and 0041 install the *defaultFont* obtained from the *fontStore* instance into it. This will ensure that when the user first sees the interface the font used in the text area and the state of the three drop-down menus will be congruent. The final step in the *init()* method is to add *theInterface* to **this** applet's *contentPane*, so that it will subsequently become visible to the user.

The *FontViewer* class declares that it implements the ActionListener interface in order that it can be registered as the *exitButton*'s actionListener resource, and so must supply an *actionPerformed()* method, as follows.

```
0046     public void actionPerformed( ActionEvent event) {
0047
0048     Component itsParent = this;
0049
0050        while (! ( itsParent instanceof Window)) {
0051           itsParent = itsParent.getParent();
0052        } // End while.
0053
0054        ((Window) itsParent).dispose();
0055     } // End actionPerformed.
```

This method will only be called if the artifact is executing as an application; the *exitButton* will not be shown if it is executing within a browser, as explained above. The *exitButton* is the only source of ActionEvents in the interface, so there is no need to determine which control generated the *event*. The method should terminate the application and remove its window from the desktop. To accomplish this it must first determine the identity of the top-level Window; this is the JFrame instance that the *main()* method adds a FontViewer into. To achieve this it ascends the layout window hierarchy, using techniques similar to those described in the FontViewerUI's *addNotify()* method, as described above. Having obtained the identity of the top-level Window in *itsParent*, its dispose() method can be called. Disposing of a top-level window destroys it and all its children, causing the application to disappear from the desktop.

The *FontViewer* class declares that it implements the ItemListener interface in order that it can be registered as the *JComboBox*es itemListener resource, and so must supply an *itemStateChanged()* method, as follows.

```
0058     public void itemStateChanged( ItemEvent event) {
0059
0060     String origin   = ((Component) event.getSource()).
0061                                                      getName();
0062     String selected = (String) event.getItem();
0063     Font   newFont  = null;
0064
0065        if ( event.getStateChange() == ItemEvent.SELECTED) {
0066           if ( origin == "family") {
0067              fontStore.setSelectedFamily( selected);
0068           } else if ( origin == "style") {
0069              fontStore.setSelectedStyle( selected);
0070           } else if ( origin == "size") {
0071              fontStore.setSelectedSize( selected);
0072           } // End if.
0073
0074           newFont = fontStore.getSelectedFont();
0075           theInterface.setFontToShow( newFont);
0076        } // End if.
0077     } // End itemStateChanged.
```

This method will be called twice when the user selects one of the options from the drop-down list on one of the JComboBoxes. It will be called once to indicate that the currently selected item has been deselected and again to indicate that a different item has been selected. An ItemEvent has a getStateChange() method that returns one of the manifest values ItemEvent.SELECTED or ItemEvent.DESELECTED, so that the reason why the event was dispatched can be decided.

The ItemEvent could have originated from any one of the three JComboBoxes on the interface and the method has to decide which of them it was. When the JComboBoxes were constructed their name attributes were set in order that this identification could be effected. The name attribute is introduced into the Component hierarchy by the Component class itself, so all Components, including all JComponents, will support this attribute.

The first step in the *itemStateChanged()* method, on lines 0060 and 0061, is to retrieve the name attribute of the Component that generated the event. All Event instances, including ItemEvent instances, have a source attribute that contains the identity of the Object that dispatched the event. The getSource() method returns this identity and as in this situation it is certain that the ItemEvent originated from a JComboBox, an extended AWT Component, it can be safely cast to Component so that the getName() method can be called. The name is stored in the String *origin* for future use.

An ItemEvent also supports an item attribute that contains the identity of the element on the list that was selected or deselected. In this example all items on the list are String instances, so, on line 0062, the *event* getItem() method is called to obtain the item attribute, which is stored in the local variable *selected*.

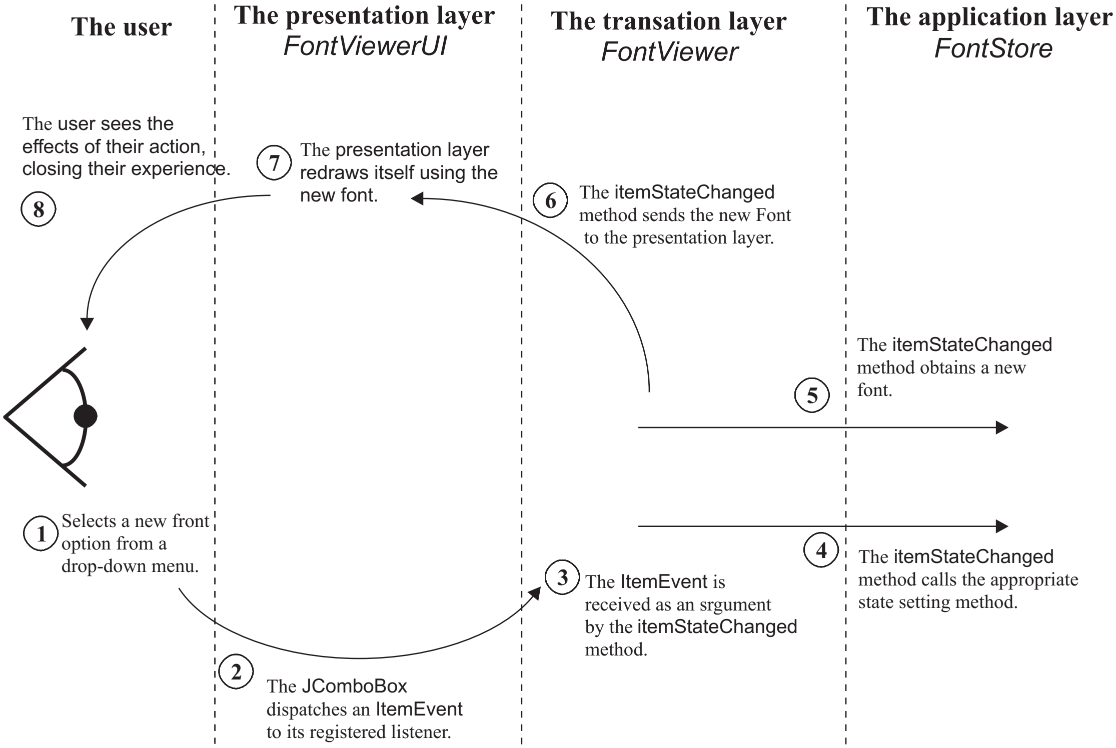

The user
The presentation layer
FontViewerUI
The transation layer
FontViewer
The application layer
FontStore
The user sees the effects of their action, closing their experience.
8
7
The presentation layer redraws itself using the new font.
6
The itemStateChanged method sends the new Font to the presentation layer.
The itemStateChanged method obtains a new font.
5
1
Selects a new front option from a drop-down menu.
4
The itemStateChanged method calls the appropriate state setting method.
3
The ItemEvent is received as an srgument by the itemStateChanged method.
2
The JComboBox dispatches an ItemEvent to its registered listener.

Figure 2.10 *FontViewer*: user interaction and closure.

ItemEvents received as a consequence of an item on the list being deselected can be ignored; only those generated as a consequence of an item being selected need be responded to. Accordingly, the major part of the method is contained within a one-way selection structure controlled by the **if** condition on line 0065. Only if the *event* is shown to have originated as a consequence of an item being selected will the body of the **if** structure be executed.

The first part of the **if** structure is itself a three-way selection structure containing a branch for each of the JComboBoxes, as determined by the value of the *origin* variable. Each branch calls the appropriate *fontStore* method, passing on as an argument the item *selected* by the user. As one attribute of the *selectedFont* within the *fontStore* has changed, a *newFont*, corresponding to the changed selection is obtained, on line 0074, and immediately passed to *theInterface*'s *setFontToShow()* method.

The effect is that whenever the user selects a new font item from one of the drop-down lists on *theInterface*, an ItemEvent containing this information is dispatched to its listening object. The listener method calls an appropriate state setting method in the *fontStore* and subsequently asks it for a *newFont*, which it passes back to *theInterface*. The user will see the sample text in the text area part of *theInterface* change to illustrate the font selected. This cycle, which closes the user's experience, is illustrated in Figure 2.10.

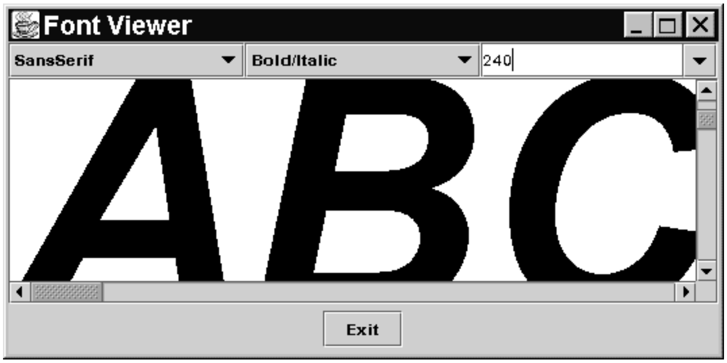


Figure 2.11 *FontViewer* with editable size control.

2.7 The *FontViewer* revisited

The JComboBox is so called because (in its default configuration) it is a combination of a drop-down list and a text entry area; by default the text entry area is non-editable. When it is editable the user can either select an option from the drop-down list or can type an item into the text entry area. In the *FontViewer* artifact it would be possible to us an editable JComboBox for the control that determines the font size, allowing the user to investigate fonts of any size. This can be effected by adding the line *sizeList*.setEditable(**true**) to the configuration of the *sizeList* component in the *FontViewerUI* constructor, and the resulting appearance of the interface is illustrated in Figure 2.11.

The illustration shows that the user has requested a 240 point, bold italic, sans serif font and that the platform has been able to supply it. However, there is no guarantee that the user will always enter an integer value into the text entry box and, with the existing implementation, this will cause a NumberFormatException to be thrown when the *fontStore* attempts to interpret the user's input as a number in its *makeFont()* method. This exception will cause the application to terminate prematurely and is unacceptable. Figure 2.12 presents an amended state transition diagram which indicates how the behavior will be changed to accommodate this consideration.

The diagram shows that if the item entered by the user, or selected from the list, is valid then the behavior is as previously specified. Otherwise the font is not changed and the font size entered shown in the area before the user typed in an invalid value is restored. This will require a change to the *FontStore itemStateChanged()* method, as follows. In this design, and subsequent implementation, a new valid entry made by the user into the text area will not be added to the list on the pull-down menu.

```
    public void itemStateChanged( ItemEvent event) {

    String origin   = ((Component) event.getSource()).
                                              getName();
    String  selected        = (String) event.getItem();
    Font    newFont         = null;
    int     newSize         = 0;
    boolean fontHasChanged  = true;

       if ( event.getStateChange() == ItemEvent.SELECTED) {
          if ( origin == "family") {
```

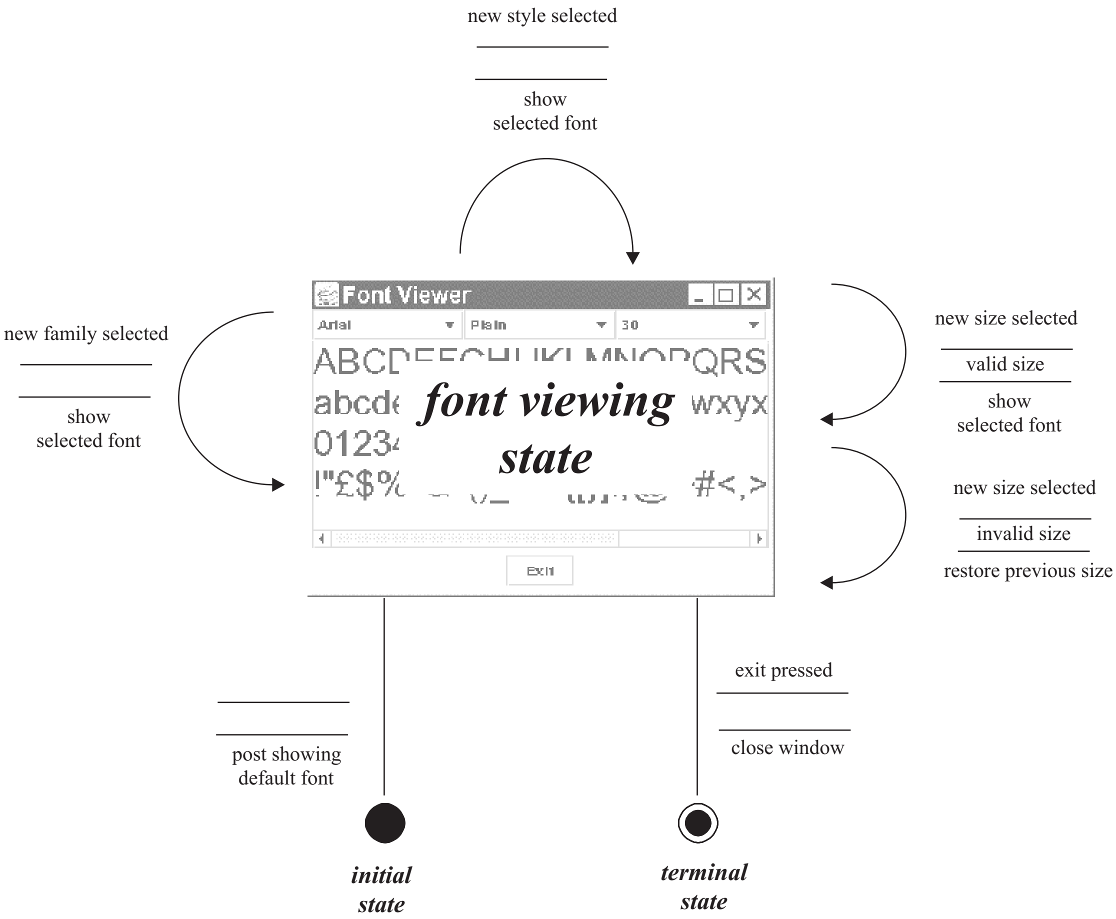


Figure 2.12 *FontViewer* artifact: amended state transition diagram.

```
      fontStore.setSelectedFamily( selected);
   } else if ( origin == "style") {
      fontStore.setSelectedStyle( selected);
   } else if ( origin == "size") {
      try {
         newSize = Integer.parseInt( selected, 10);
         fontStore.setSelectedSize(  selected);
      } catch ( NumberFormatException exception) {
          theInterface.restoreSize(
                       fontStore.getSelectedSize());
          fontHasChanged = false;
      } // End try/catch.
   } // End if.

   if ( fontHasChanged) {
      newFont = fontStore.getSelectedFont();
      theInterface.setFontToShow( newFont);
   } // End if.
} // End if.
```

```
0088      } // End itemStateChanged.
```

The major changes in this version of the method are in the part of the inner **if** structure that deals with events received from the *size* control, on lines 0073 to 0080. It commences, on line 0074, by attempting to convert the *selected* item into an integer, in *newSize*. As this may throw a NumberFormatException it is contained within a **try/catch** structure. If the exception is not thrown then the user's input was a valid integer, and the method proceeds to call the *fontStore setSelectedSize()* method, as before.

However, if the attempt to convert *selected* does throw an exception then the exception handler, on lines 0077 to 0079, will be executed. Its first step is to call a new method of the *FontStoreUI* class called *restoreSize()*, which will be described shortly. The argument to this call is the current selected size obtained by a call of a new *fontStore* inquiry method called *getSelectedSize()*. If the exception handler has been invoked then the font has not been changed and the *fontHasChanged* flag, declared on line 0065, is set to indicate this. The state of this flag determines, on lines 0083 to 0086, whether a message is to be sent to *theInterface* to redisplay itself with a new font.

The old font size shown on the size control could be restored, on lines 0077 to 0078, by the following fragment.

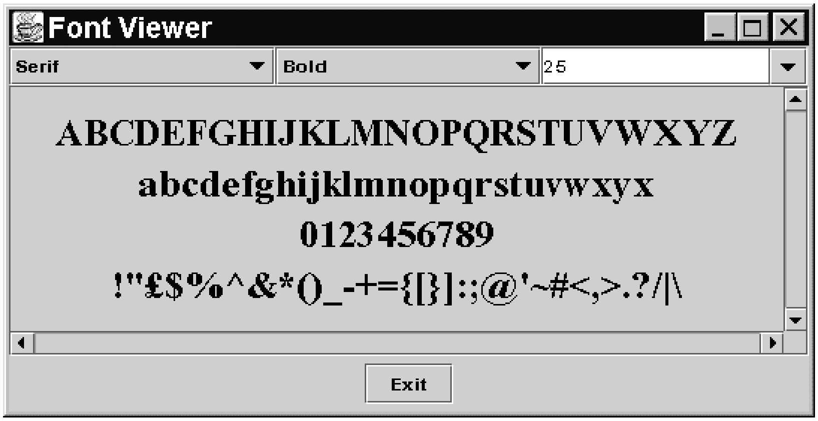
```
((JComboBox) event.getItemSelectable()).
                         setSelectedItem(
                            fontStore.getSelectedSize());
```

The *itemSelectable* attribute of the event identifies the component that generated the event and this can be safely cast to JComboBox in this situation. Having obtained its identity, its setSelectedItem() method can be called, passing as an argument the *selectedSize* attribute of the *fontStore*, which has not been changed by the *itemStateChanged()* method. This will restore the unchanged *selectedSize* from the *fontStore* into the size text entry box on the interface, but in doing so it will also generate two further ItemEvents. One will indicate that the invalid value has been deselected and the second will indicate that the previous (valid) value has been restored. This second event will be processed as a valid size change and will cause the interface to install an identical font and redraw itself.

Although this is probably not significant in this situation, in general such unwanted side effects should be avoided. The new *FontViewerUI restoreSize()* method will change the selected item in the *sizeList*, without dispatching any event. The implementation of this new method is as follows.

```
0100      protected void restoreSize( String toRestore) {
0101         sizeList.removeItemListener( itsItemListener);
0102         sizeList.setSelectedItem(    toRestore);
0103         sizeList.addItemListener(    itsItemListener);
0104      } // End restoreSize.
```

This method makes use of an additional encapsulated attribute of the class of the *ItemListener* interface class called *itsItemListener*, which is initialized in the constructor to the identity of the interface's item listener passed to it in the *sendItemsHere* argument. The *restoreSize()* method effects the change of the selected item on the *sizeList* control without causing it to dispatch an ItemEvent by first removing *itsItemListener*, then changing its *selectedItem* to the argument *toRestore* and finally reinstalling *itsItemListener*.



ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyx
0123456789
!"£$%^&*()_-+={[}]:;@'~#<,>.?/|\

Figure 2.13 *FontViewer* artifact with text centered.

This change to the *FontViewer* is not optimal as it allows the user to make a '*mistake*' by typing in an entry that cannot be interpreted as an integer, and then having to correct the consequences of this. It would be preferable to prevent the user from making the '*mistake*' in the first instance by the provision of a specialized component that only allowed numeric inputs. The construction of such a component will be described in Chapter 4.

The second change that will be made to the *FontViewer* interface is to replace the JTextArea, where the font is illustrated, with an instance of a new class *FontViewerTextArea*. The appearance of the *FontViewer* as illustrated in Figure 2.1 shows that the sample text is displayed left justified. It is a small improvement to the appearance to make it centered, as illustrated in Figure 2.13. However, this illustrates the user interface design principle that details matter.

> *Details matter! Although the essential functionality and usability of an artifact are very necessary, by themselves they are insufficient. The fine details of behavior and presentation are essential if the user is to maintain an overall favorable attitude towards the product.*

The *FontViewerTextArea* class diagram is shown in Figure 2.14. It shows that the *FontViewerTextArea* class extends the simplest of the JFC classes, the JComponent class. This allows it to inherit many JFC capabilities, such as the ability to display a border as will be explained in the next chapter, without inheriting any specialized resources, such as a JButton's ability to respond to the user. It contains two encapsulated attributes: *showThis* is the content to be shown and the font *toUse* is used to render it. It also contains a private method called *establishSize()*, whose purpose will be described below.

The protected constructor takes the String which the component is *toShow*. The *setFont()* method allows a *newFont* to be specified; this method must be declared **public** as it overrides an existing method with **public** visibility introduced into the hierarchy by the AWT Component class. The **public** *addNotify()* method is supplied, as some essential initialization operations cannot be performed until the component is almost ready to be

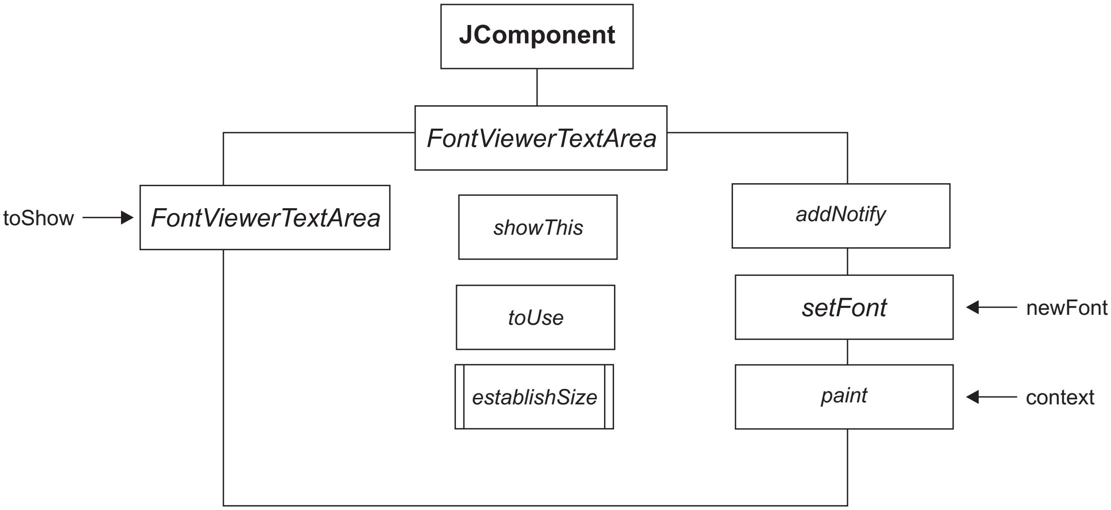


Figure 2.14 *FontViewerTextArea* class diagram.

displayed. The final method, *paint()*, will be described in detail shortly. The implementation of this class, as far as the start of the constructor, is as follows.

```
0010  package fontviewer;
0011
0012  import javax.swing.*;
0013
0014  import java.awt.*;
0015  import java.util.*;
0016
0017  class FontViewerTextArea extends JComponent {
0018
0019  private String[] showThis = null;
0020  private Font     toUse    = null;
```

The class is declared, without public visibility outside the *fontviewer* package, on line 0017. The *showThis* attribute is declared as an array of String, on line 0019, and the Font *toUse* on line 0020. The constructor is implemented as follows.

```
0024     protected FontViewerTextArea( String toShow) {
0025
0026        super();
0027
0028        StringTokenizer tokenizer = new
0029                              StringTokenizer( toShow, "\n");
0030        int numberOfLines = tokenizer.countTokens();
0031        showThis = new String[ numberOfLines];
0032        for ( int index =0; index < numberOfLines; index++) {
0033          showThis[ index] = ((String) tokenizer.nextToken());
0034        } // End for.
0035        toUse = this.getFont();
0036     } // End FontViewerTextArea constructor.
```

After calling the **super**, JComponent, default constructor a StringTokenizer instance called *tokenizer* is constructed, initialized to split the String *toShow* into lines by tokenizing on new line ("\n") characters. The *tokenizer* can then be used to count the *numberOfLines* in *toShow* by calling its countTokens() method. As the *numberOfLines* is now known, the bounds of the *showThis* array can be established on line 0031. The loop on lines 0032 to 0034 will iterate through the array extracting each line, using *nextToken()*, from *toShow* and storing them in the array *showThis*. The last step of the constructor, on line 0035, is to initialize the Font attribute *toUse* to the JComponent default font, obtained by calling the getFont() method inherited from the AWT Component class.

Further initialization of the *FontViewerTextArea* instance occurs in the addNotify() method, which, as explained above, will be called by the Java run-time environment just before the component is engaged in layout negotiations. The purpose of the *FontViewerTextArea addNotify()* method is to establish the required height and width of the instance so that the layout negotiations will reflect its requirements. The implementation of *addNotify()* is as follows.

```
0039     public void addNotify() {
0040        this.establishSize();
0041        super.addNotify();
0042     } // End addNotify.
```

The method first calls the private *establishSize()* method and then calls its **super** addNotify() method to allow layout negotiations, and the subsequent display of the component, to continue. The implementation of the *establishSize()* method is as follows.

```
0054     private void establishSize() {
0055
0056     Dimension   bestSize  = new Dimension( 0, 0);
0057     FontMetrics metrics   = this.getFontMetrics( toUse);
0058     int         thisWidth = 0;
0059     int         widest    = 0;
0060
0061        bestSize.height = showThis.length * metrics.getHeight();
0062
0063        for ( int index =0; index < showThis.length; index++){
0064           thisWidth = metrics.stringWidth( showThis[ index]);
0065           if ( thisWidth > widest) {
0066              widest = thisWidth;
0067           } // End if.
0068        } // End for.
0069        bestSize.width = widest;
0070
0071        this.setMinimumSize(   bestSize);
0072        this.setMaximumSize(   bestSize);
0073        this.setPreferredSize( bestSize);
0074        this.setSize( bestSize);
0075     } // End establishSize.
```

The purpose of this method is to determine the appropriate size of the component, given the Font that it is *toUse*. To accomplish this it commences by constructing an instance of the Dimension class, called *bestSize*, initialized with a 0 by 0 width and height. Dimension is a very simple class, supplied by the AWT, that has two public attributes called width and height. The details of the dimensions of the individual *glyphs* that make up a Font are encapsulated within instances of the FontMetrics class and, on line 0057, an instance of the FontMetrics class, called *metrics*, is obtained for the Font *toUse*. The two remaining local declarations will be used to record the width of each line in turn, *thisWidth*, and to record the width of the *widest* line, which will determine the required width of the component.

Object → java.awt.FontMetrics

```
public FontMetrics( Font toObtainMetricsFor)
```

Constructs a FontMetrics instance containing details of the Font supplied as an argument.

```
public Font getFont()
public int  getAscent()
public int  getDescent()
public int  getHeight()
public int  stringWidth( String toObtainTheWidthOf)
```

Inquiry methods. The heights of every character glyph in a font are identical; the width depends upon the characters contained in the string *toObtainTheWidthOf*.

The steps of the method commence on line 0061, where the number of lines in *showThis* is multiplied by the height of every character in the font. The resulting value is the required height of the *FontViewerTextArea* instance and is used to initialize the value of the *bestSize* height attribute. Every glyph in a font has the same height, consisting of the maximum extent to which some characters extend below the baseline and the maximum extent to which some extend above the baseline. The baseline is an imaginary line along which characters are laid out. For example, in the character sequence *goh* the baseline is at the bottom of the 'o' and the tail of the 'g' extends below this; the maximum height is indicated by the tip of the 'h'.

Although all characters in a font have the same height they do not always have the same width. For example, in the character sequence *miw* the 'i' character requires less width than the 'm' or the 'w'. Accordingly, the width of the lines in the String *toShow* can only be established by supplying each line in turn to the *metrics* stringWidth() method. This is accomplished within the loop between lines 0063 and 0067, which stores the width of each line in turn in *thisWidth* and stores the widest of them in *widest*. After the end of the loop, on line 0069, the value in *widest* is used to initialize the width attribute of *bestSize*.

The effect of this first stage of the *establishSize()* method is to store in *bestSize* the required height and width of the *FontViewerTextArea* instance with regard to the Font specified in *toUse*. Lines 0071 to 0074 use *bestSize* as the argument to methods that set the values of the *minimumSize*, *maximumSize* and *preferredSize* attributes. These attributes will be used during layout negotiations to determine the range of sizes that the component would like; however, there is no guarantee that it will be accorded a size anywhere within this range. The final step of the *establishSize()* method is to request that it be resized by calling its setSize() method; this will initiate layout negotiations which may cause it to be accorded the space it has asked for.

The *establishSize()* method is also called from the *setFont()* method, as a change of Font will require a different amount of space on the screen. The implementation of *setFont()* is as follows.

```
   public void setFont( Font newFont) {
      toUse = newFont;
      if ( this.isShowing()) {
         this.establishSize();
         this.repaint();
      } // End if.
   } // End setFont.
```

After storing the *newFont* in the *toUse* attribute the component calls its *establishSize()* method to attempt to obtain a suitably sized window and then calls the inherited repaint() method to redisplay the component with the *newFont*. However, it is possible that this method may be called between the time that the instance is constructed, but before *addNotify()* has been called. In these cases the attempt to obtain FontMetrics in *establishSize()* will throw an exception, and to prevent this the latter part of *setFont()* is only executed if the component isShowing().

A component should, in general, never render itself directly onto the screen. Instead it should supply a paint() method which does this and wait until it is called. It will be called when the component first becomes visible and subsequently when the window needs refreshing after being occluded by another window or after being minimized. A call of paint() can be explicitly requested by calling the repaint() method, as at the end of the *setFont()* method, which will arrange for the paint() method to be called on a separate flow of control so as not to interfere with processing the user's requests. Redisplaying the contents of a window can be a time-consuming process and if it were performed on the main flow of control this would result in the interface becoming unresponsive while this happened. This indirect mechanism for refreshing a window avoids this problem. The implementation of the *FontViewerTextArea paint()* method is as follows.

```
   public void paint( Graphics context) {

   FontMetrics metrics         = this.getFontMetrics( toUse);
   int         characterHeight = metrics.getHeight();
   int         totalHeight     = showThis.length *
                                              characterHeight;
   int         itsWidth        = this.getSize().width;
   int         fromTop         = ((this.getSize().height -
                                   totalHeight) /2) +
                                   metrics.getMaxAscent();
   int         leftOffset      = 0;

      super.paint( context);
      context.setFont( toUse);
      for ( int index =0; index < showThis.length; index++) {
         leftOffset = ( itsWidth -
                 metrics.stringWidth( showThis[ index]))/2;
         context.drawString( showThis[ index],
                                      leftOffset, fromTop);
         fromTop += characterHeight;
      } // End for.
   } // end paint.

} // End FontViewerTextArea.
```

When the Java run-time environment calls a paint() method it supplies an instance of the AWT Graphics class. This is known as the Graphics *context* and encapsulates a large amount of information, including the size of the window to draw in, the Font to use, the foreground and background colors to use, the line style and width to use, and the plotting mode to use. It also supplies a large number of methods to draw lines and shapes and to render text into the window. The paint() method uses the resources of the *context* passed to it to redisplay the contents of the window to reflect the state of the object. In this case it shows the String *toShow*, using the Font *toUse*, horizontally and vertically centered within the available extent.

To accomplish this it needs a number of local variables, declared and initialized on lines 0081 to 0089. The first of these is an instance of FontMetrics, called *metrics*, initialized to describe the characteristics of the Font *toUse*, as described above. The *characterHeight* is the height of every glyph in *toUse* and *totalHeight* is the height required to display all the lines in *showThis*. Line 0085 declares *itsWidth* and initializes it to the actual width of the window as determined by the *getSize()* method, which returns a Dimension instance whose **public** width attribute can be accessed directly. Lines 0086 to 0088 compute the vertical offset from the top, *fromTop*, of the window where the first line should be rendered. This is determined by the height of the window (line 0086): half the *totalHeight* required for the whole of the contents (line 0087) and the distance from the top of a glyph to its baseline (line 0088). The final declaration, on line 0089, of *leftOffset*, will be used to decide the distance from the left-hand side of the window where each line in turn is to be displayed. These considerations are illustrated, in outline, in Figure 2.15.

The first step of the method, on line 0091, is to call the **super**, JComponent, paint() method passing on the context. This will clear any existing contents of the window and fill it with the background color. Line 0092 then informs the *context* of the Font that it is *toUse* when rendering Strings.

The loop between lines 0093 and 0099 renders each line in turn into the window. The first step, on lines 0094 to 0095, is to compute the extent of the *leftOffset* taking into account the width of the window (*itsWidth*) and the width required to display the current

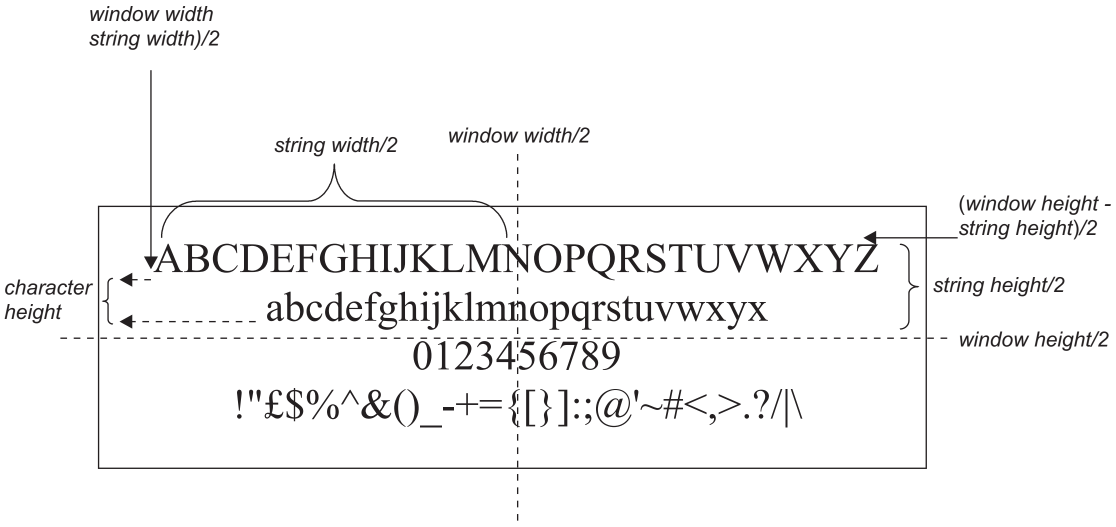


Figure 2.15 *FontViewerTextArea*: *paint()* method implementation details.

line (*showThis*[*index*]). Having determined this the line is rendered into the window using the *context* drawString() method; the arguments to this method are the String to be rendered, the offset from the left-hand side of the window and the offset from the top of the window. Before the loop finishes, the value of *fromTop* is incremented by the *characterHeight*, so that the next line will be positioned below the line that has just been rendered. The end of the loop is also the end of the *paint()* method, which, in turn, completes the *FontViewerTextArea* class declaration. The *FontViewerUI* class can be amended to use a *FontViewerTextArea* rather than a *TextArea* instance, by declaring and constructing an instance of the new class and also removing the setEditable() call, as a *FontViewerTextArea* does not support editability.

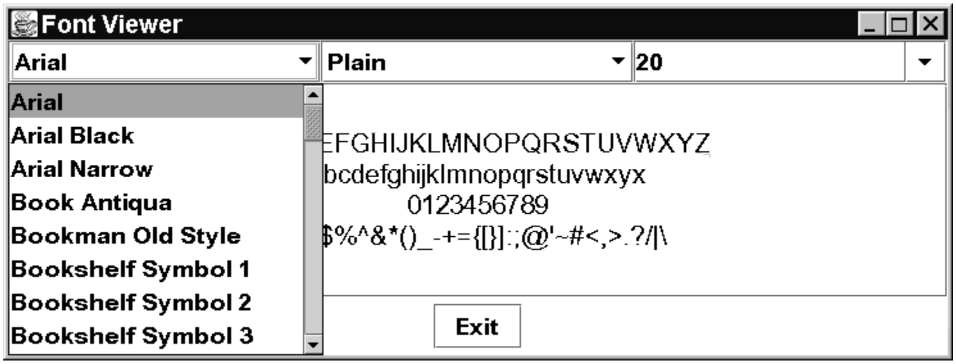
The implementation of the *FontViewerTextArea* class has not taken into consideration the environment in which it will be used, in this example the *FontViewer* artifact. This is essential if the class is to be successfully reused in other circumstances where it might not be contained within a JScrollPane's viewport. In this circumstance it is almost certain that the instance will always be accorded its preferred size, as only the part of it which appears within the viewport need actually be displayed. The JScrollPane will, during layout negotiations, take the *FontViewerTextArea* instance's size into account to decide whether the scroll bars should be provided. It will also arrange for the *paint()* method to be called, with the *context* supplied constraining output to the screen to those parts that are actually visible within the viewport, as the user interacts with the scroll bars. Hence although the *paint()* method will redisplay the entire contents of the window, only that part of it which is visible will actually be shown on the screen, as illustrated in Figure 2.1. When the user interacts with the scroll bars, the JScrollPane instance will arrange for the *FontViewerTextArea*'s *paint()* method to be repeatedly called, with the context supplied constraining output to the appropriate area.

There is the possibility, when a small font size is chosen, that the *preferredSize* of the *FontViewerTextArea* instance will be smaller than the viewport. In these circumstances the JScrollPane will accord it a window size larger than it has requested, equal to the size of the viewport. However, the *paint()* method is constructed so as to continue to display its contents centered within the available space, as illustrated in Figure 2.13. This illustration also shows that, by default, the *FontViewerTextArea* will use a gray background, rather than the white background provided by the TextArea used in Figure 2.1. The next part of this chapter will consider the techniques available to control *resources* such as this.

2.8 Resource management

The illustrations of the artifacts presented so far in this book have used the default appearance supplied by Java. The way in which a component appears, and the manner in which it interacts with the user, is known as its *look and feel*. The default look and feel supplied by the JFC is known as "Metal" and features black or blue/gray characters on gray backgrounds with a light purple used for highlighting. It has a high degree of consistency between different classes of component and has been produced so as to be visually appealing without looking and feeling like any of the existing well-known user interface toolkits such as MS Windows, Apple Macintosh or X/Motif. Where one of these native look and feels is required it can be requested and installed on an application-wide basis, as will be demonstrated in the next chapter.

However, some aspects of the look and feel, particularly the look part, may be inappropriate to some categories of users. There are three fundamental attributes, introduced by the AWT Component class, that determine the most important aspects of how a



EFGHIJKLMNOPQRSTUVWXYZ
bcdefghijklmnopqrstuvwxyx
0123456789
$%^&*()_-+={[}]:;@'~#<,>.?/|\

Figure 2.16 *FontViewer*: amended appearance.

component presents itself to the user. These are the Font to use, the foreground Color to use and the background Color to use. These attributes are inherited by every AWT and JFC component class and their state-setting methods can be used so as to alter the appearance of an artifact's interface. For example, the setFont(), setBackground() and setForeground() methods of all the components used in the *FontViewerUI* interface could be specified as it is initialized. Figure 2.16 shows the appearance of the interface after the backgrounds have been consistently set to white, the foregrounds to black and the font to Dialog bold 18.

This was achieved, for example, with the *familyList* JComboBox with the following code fragment.

```
      familyList = new JComboBox( namesToShow);
      familyList.setSelectedItem( selectedName);
      familyList.addItemListener( sendItemsHere);
      familyList.setName( "family");
      familyList.setBackground( Color.white);
      familyList.setBackgroun(  Color.black);
      familyList.setFont( new Font( "Dialog". Font.Plain, 18));
```

Additionally, in order for the configuration of the Panels to be effective their opaque attribute must be specified as **true** in order to ensure that their backgrounds are actually painted; for example *fontPanel*.setOpaque(**true**). By default panels are transparent, which allows any underlying image to show through.

Although this technique for controlling the appearance of components is effective, it is piecemeal and laborious. It can be predicted that during construction, or maintenance, an attribute that should be set will be omitted or misset, causing parts of the interface to have an inconsistent appearance. A more effective approach is to inform the UIManager of the defaults for all resources on a class-wide basis.

Object → javax.swing.UIManager

The object which maintains knowledge about the look and feel resources using three levels: user, look and feel, and system. When a resource (e.g. a color or a font) is required they are checked in the sequence stated so that the user's requirements are paramount.

```
public static Object get( Object key)
public static Object put( Object key, Object value)
```

User resource preferences are stored as *key/value* pairs. The put() method will return the default value for the resource.

When a component is rendering itself on the screen it consults the static UIManager to obtain values for the various resources it should use. If the UIManager is informed of the preferred resources to use before any of the components are rendered then this approach will ensure consistency and be considerably less laborious than setting all the resources individually. The same appearance as shown in Figure 2.16 can be obtained by including a new **private** method called *setResources()*, implemented as follows, and calling it at the start of the *FontViewerUI* constructor, immediately after the **super** constructor has been called.

```
private void setResources() {

ColorUIResource defaultBackground = new
                          ColorUIResource( Color.white);
ColorUIResource defaultForeground = new
                          ColorUIResource( Color.black);
FontUIResource  defaultFont = new FontUIResource(
                     new Font( "Dialog", Font.BOLD, 18));

    UIManager.put( "Button.background",
                                      defaultBackground);
    UIManager.put( "Button.foreground",
                                      defaultForeground);
    UIManager.put( "Button.font",           defaultFont);


    UIManager.put( "Panel.background",
                                      defaultBackground);
    UIManager.put( "Panel.foreground",
                                      defaultForeground);


    UIManager.put( "ComboBox.background",
                                       defaultBackground);
    UIManager.put( "ComboBox.foreground",
                                       defaultForeground);
    UIManager.put( "ComboBox.font",          defaultFont);

    UIManager.put( "TextField.font",         defaultFont);
} // End setResources.
```

The majority of the implementation of the method is calls of the static UIManager put() method. This method takes two parameters: a *key* identifying the resource and a *value* for the resource. Java defines the keys, and a technique to obtain a list of them will be described shortly. The *value*s must be instances of one of the appropriate javax.swing.plaf resource classes, for example ColorUIResource.

The method commences by constructing two instances of the ColorUIResource class, called *defaultBackground* and *defaultForeground*, the argument to the constructor being one of the class-wide constants supplied by the Color class. (The Color class will be described in detail in the next chapter.) An instance of the FontUIResource class, called *defaultFont*, is constructed, specifying as an argument a **new**ly constructed instance of the Font class.

These three resources are then used as value arguments in the rest of the method to ensure that all components have the same resources. For example lines 0156 and 0157 establish the *defaultBackground*, white, as the background color of all JComboBox instances that are used in the interface. The last resource configuration is of the font resource of the JTextField class, as an instance of this class is used as the text entry area of the editable *sizeList* JComboBox.

The following static program will output a list of all known resource keys and their values on the terminal. The UIManager getDefaults() method returns a Hashtable, called *defaults*, containing all known keys and values. Having obtained the Hashtable an Enumeration, called *enum*, is prepared to iterate through all of its *key*s. Within the body of the loop each *key* is obtained from the enumeration in turn and used as an argument to the *defaults* get() method to retrieve the *value* associated with it. Having obtained a *key* and its *value* they are output on the terminal.

```
import java.util.*;
import javax.swing.plaf.*;

public class ResourceDemo extends Object {

  public static void main( String args[]) {

   Hashtable   defaults = UIManager.getDefaults();
   Enumeration enum     = defaults.keys();

   Object key   = null;
   Object value = null;

      while ( enum.hasMoreElements()) {
         key   = enum.nextElement();
         value = defaults.get( key);
         System.out.println( key + "\n " + value +"\n");
      } // End while.
   } // End main.
} // End class ResourceDemo.
```

When this program was executed on a Windows 95 platform the output consisted of over 440 key/value pairs, commencing as follows.

```
ToolBar.background
 javax.swing.plaf.ColorUIResource[r=204,g=204,b=204]
Tree.selectionBorderColor
 javax.swing.plaf.ColorUIResource[r=255,g=255,b=255]
TextPane.caretForeground
 javax.swing.plaf.ColorUIResource[r=0,g=0,b=0]
text
 javax.swing.plaf.ColorUIResource[r=255,g=255,b=255]
```

When a resource is specified to the UIManager using its put() method, it is installed into its first-level resource table, known as the *user resources*. There are two further tables, one for the *look and feel resources* and one for the *native platform resources*. When a component requests a resource, using the UIManager get() method, it will return the value from the first table which contains an entry for it. Hence user-specified resources will take priority over look and feel resources, which in turn will take priority over the native platform resources.

Hence by producing a list of the different components that are used in an artifact's interface and consulting the list produced by the *ResourceDemo* program a *setResources()* method can be produced that will pre-configure the appearance of all classes used. This approach will be effective even with any of the four standard look and feels (Metal, MS Windows, Macintosh and X/Motif) supplied by Java. Unfortunately, there is no standard mechanism in the current Java environment that allows the required resource settings to be established for all artifacts.

Summary

- ♦ Customization of an artifact to a user's individual requirements is essential to the production of high-quality products.
- ♦ Fonts can be specified by defining family, style and size.
- ♦ For portability, only the Monospaced, Serif, SansSerif and Dialog families should be used.
- ♦ The construction of artifacts should take into account whether they are executing as an applet or as an application.
- ♦ The JComboBox class provides a pull-down list capability; if it is editable it includes a text area input area.
- ♦ A JComboBox instance dispatches ItemEvents when the user changes its value and requires ItemListeners to be registered with it to receive them.
- ♦ The ItemListener interface mandates a single method called itemStateChanged().
- ♦ The JScrollPane class provides a window containing horizontal and vertical scroll bars and a viewport; it will automatically pan over any component added to the viewport.
- ♦ The addNotify() method is called after a component has been constructed but before it engages in layout negotiations, and some initialization of a component has to be left until this time.
- ♦ The GridLayout manager arranges its components into rows and columns, with each cell forced to be the same size.
- ♦ The JTextArea class provides a single line text input area and fires an ActionEvent when the user presses the `<ENTER>` key.
- ♦ Specialized classes can be extended from existing classes.
- ♦ The UIManager put() method allows user resource values to be specified that take priority over both look and feel and native resources.
- ♦ All JComponents provide accessibility support.

Exercises

2.1 Add a fourth pull-down list to the *FontViewer* artifact with entries indicating upper-case, lower-case, digits and punctuation. The central area should only display a line of glyphs corresponding to that indicated by the user.

2.2 Extend the artifact developed in Exercise 2.1 so that it displays five lines of glyphs. The central line should show, in black, the font size requested by the user; the two lines above should show, in gray, two smaller sized fonts and, likewise, the two lines below two larger sized fonts. This will require the *FontViewerTextArea* class to be re-engineered.

2.3 Remove the style menu from the *FontViewer* artifact as developed in Exercise 2.1 and instead display four lines of text with plain, bold, italic and bold italic styles.

2.4 Attempt to determine which of the four versions of *FontViewer* is preferred by users. To do this, ask a number of users to use one of the versions to choose a collection of fonts (two or three) for specified requirements (e.g. to use as the text for the front and inside of a birthday card). Obtain the user's opinion of the artifact by means of a questionnaire.

2.5 Add two further pull-down menus to the *FontViewer* artifact as developed in the chapter which allow the foreground and background color of the central area to be changed.

2.6 Adapt the *FontViewer* artifact as developed in the chapter so that it displays the first 256 characters from the font in a 16 row by 8 column GridLayout.

2.7 Revisit the *Stopwatch* artifact from Chapter 1 and add a *setResources()* method to it to improve the appearance of the interface.

2.8 Investigate the protocol of JComboBox and upgrade the editable instance used in the chapter so that it adds the font size entered by the user into the appropriate place on its drop-down list.

3

Some fundamental JFC attributes

3.1 Introduction

In Chapters 1 and 2, two central concepts concerned with the production engineering of artifacts that have a GUI were emphasized: the event generation and listening mechanism and the derivation of a software architecture from usability considerations. However, the visual appearance of the interfaces produced was not particularly engaging until resource management was introduced at the end of Chapter 2. In this chapter some fundamental attributes that further enhance the appearance of a large number of JFC component classes will be introduced.

This will be accomplished by the development of an artifact called *TabbedLabelDemo*. The interface will present a single instance of the JLabel class accompanied by a number of control panels which allow the effects of changing the value of some of its attributes to be interactively investigated. In order to implement the control panels a number of different layout management policies, JFC component classes and event classes will have to be introduced.

The intention is not only to provide a detailed introduction to the JLabel class, but, as many of the attributes which will be introduced are common to many other simple JFC components, to allow this understanding to be subsequently generalized. The engineering of the artifact in this chapter will not be as rigorous as that presented in the previous chapters, since because the example does not have any real purpose, there is no need for a full three-layer implementation.

The major artifact introduced in this chapter will require a number of different control panels. The implementations of these panels contain many repeated code fragments, and accordingly, in places, only the parts that differ significantly will be presented in detail. The Appendix contains details of how the entire source code can be obtained.

3.2 The JLabel class

The JLabel class is one of the simplest of the JFC classes; its most obvious attributes are an image used as an icon and a text label to accompany it. When a JLabel instance was used in the previous chapter it was constructed so as to display only the text label by specifying a String as its label resource using its setText() method. In this chapter other resources of the JLabel class will be introduced and described. Many of these resources are common to all, or many, of the JComponents, so an understanding of how they operate in a JLabel instance can be used with other classes.

In order to add an iconic image to a JLabel instance an instance of a class which implements the Icon interface must be specified as its icon resource, either when it is constructed or afterwards using its setIcon() method. The JLabel class has a number of constructors, allowing a combination of icon and label to be specified, along with the horizontal positioning of the icon and label within the extent of the JLabel's window.

JComponent → javax.swing.JLabel

```
JLabel()
JLabel( Icon icon)
JLabel( Icon icon, int position)
JLabel( String label)
JLabel( String label, int position)
JLabel( String label, Icon icon, int position)
```

Constructs a JLabel instance with the image, icon and label supplied; *position* can be one of StringConstants.LEFT, StringConstants.CENTER (default) or StringConstants.RIGHT.

Icon is an interface specification and the simplest class which implements it is the ImageIcon class, instances of which encapsulate a small non-editable image. The easiest way to specify the image to be displayed by an ImageIcon contained within an application is to specify the name of a GIF or JPEG file located in the same file store from which the java class file was obtained. For example, the ImageIcon constructor:

```
ImageIcon( String filename)
```

will attempt to construct an image from the file specified. The file will be loaded and the image constructed on a separate thread of control, so it may not be immediately available, or may not be available at all if the filename is invalid or the file it identifies does not contain a recognizable image. To cater for this possibility and, more importantly, to provide support for environments and users that cannot use images, alternative constructors allow a text message to be associated with the image. When the graphic cannot be shown, the text message will be shown in its place. Further details of the ImageIcon class and Icon interface can be obtained from the JFC documentation.

Object → javax.swing.ImageIcon

```
ImageIcon( String filename)
ImageIcon( String filename, String message)
ImageIcon( URL location)
ImageIcon( URL location, String message)
```

Attempts to construct an ImageIcon containing the picture contained in the *filename* indicated, stored in the same file store as the Applet was obtained from or from the URL specified. The picture will be loaded on a separate thread of control and the ImageIcon will only display *message* if the picture could not be loaded for any reason.

The following, very minimal, applet illustrates how a JLabel instance containing an image and a label can be constructed.

```
0001 // Filename FirstJLabelDemo.java.
0002 // First demonstration of how to construct a
```

```
0003 // JLabel instance.
0004 //
0005 // Written for JFC book, ch 3, see text.
0006 // Fintan Culwin, v0.1, Dec 1999.
0007
0008 package chap3;
0009
0010 import javax.swing.*;
0011 import java.awt.*;
0012
0013 public class FirstJLabelDemo extends JApplet {
0014
0015 private final static ImageIcon labelIcon =
0016                                 new ImageIcon( "shredder.gif",
0017                                                "shredder");
0018
0019    public void init() {
0020
0021    JLabel demoLabel = new JLabel( "destroy", labelIcon,
0022                                    SwingConstants.CENTER);
0023       demoLabel.setBackground( Color.white);
0024       demoLabel.setForeground( Color.black);
0025       demoLabel.setOpaque( true);
0026       this.getContentPane().add( demoLabel);
0027    } // End init.
0028
--     // Details of main() omitted
```

Following the who, what, why, where, when comments and the declaration that this class is a member of the *chap3* package, lines 0010 and 0011 provide access to the JFC and AWT facilities. Line 0013 then commences the declaration of the *FirstJLabelDemo* class by stating that it extends the JApplet class. Lines 0015 to 0017 construct a constant class-wide instance of the ImageIcon class containing the image from the "*shredder.gif*" file, with the alternative message "*shredder*". As a class-wide instance the image will commence loading as soon as the class is loaded, before any instances of it are constructed. This technique also allows the single image to be shared between all instances of the class that may require it and, as it is declared as a constant, they cannot corrupt it. The advantages of this approach are minimal in this case, as only a single instance of the class will be used; however, it may be advantageous in other circumstances.

The *FirstJLabelDemo init()* action is then declared, on lines 0019 to 0027, and proceeds by constructing an instance of the JLabel class, called *demoLabel*, containing the text label "*destroy*" and the icon from *labelIcon*, both of which are horizontally centered within the extent of the label. The JLabel instance is then configured by having an opaque white background and black foreground specified. By default a JFC component will (probably) have a transparent gray background with pale blue foreground, which is not particularly aesthetic for this requirement.

Once constructed, the JLabel instance is added to the JApplet's contentPane, which will ensure that it becomes visible as the applet is launched. The appearance of this applet is shown in the left-hand image in Figure 3.1, the right-hand image shows its appearance when the iconic image cannot be obtained and the alternative text is used.

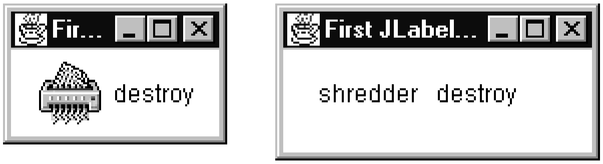


Figure 3.1 A simple JLabel instance.

Figure 3.1 shows that, by default, the text label is positioned to the right and centered within the height of the icon. The location of the text relative to the icon is determined by the values of two attributes of the JLabel class called horizontalTextPosition and verticalTextPosition, which will be explored in the following section.

> *The icon is suggesting the complete destruction of a resource, using the metaphor of a document shredder. Deletion of a document in a desktop windowing system does not actually delete the file but just moves it to a temporary storage area, a wastebasket or a recycle bin, from where it can be retrieved. Even emptying the temporary storage area, equivalent to using a delete command from the command line, does not destroy the information in the file but only the filename entry in the directory structure. It is still possible for the contents of the file to be retrieved, as many have discovered to their cost. This icon might be associated with an action which physically removes the information from the storage medium, although here the metaphor might not be completely accurate, as it is possible for a shredded document to be taped back together, again a problem which some have discovered. These considerations illustrate that an iconic representation of an action within a computer interface is a metaphor for the real-world understanding of the corresponding action and not an exact analog. It is likely that all of the most effective metaphors have already been discovered, and hence providing an iconic representation of an action will need additional support if the user is to be expected to understand it.*

The technique for constructing an ImageIcon instance from a filename, as given above, may generate a security exception when the artifact is executed as an applet. The reason for this is that most systems will be configured so as to prevent an applet, which may have arrived over the Web from an unknown host, from having any access to local resources. In these circumstances the file containing the image will have to be loaded across the Web from the host which supplied the applet. The amended code is as follows.

```
package chap3;

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


public class FirstJLabelApplet extends JApplet {

private ImageIcon labelIcon = null;

```

```
0021     public void init() {
0022
0023     JLabel demoLabel = null;
0024
0025        if ( labelIcon == null) {
0026           try {
0027              URL url = null;
0028                 this.getAppletContext();
0029                 url = new URL( this.getCodeBase(),
0030                                "shredder.gif");
0031                 labelIcon = new ImageIcon( url, "shredder");
0032                 demoLabel = new JLabel( "destroy", labelIcon,
0043                                        SwingConstants.CENTER);
0034           } catch ( MalformedURLException exception) {
0035                 demoLabel = new JLabel( "destroy",
0036                                        SwingConstants.CENTER);
0037           } catch ( NullPointerException exception) {
0038              labelIcon = new ImageIcon( "shredder.gif",
0039                                         "shredder");
0040              demoLabel = new JLabel( "destroy", labelIcon,
0041                                        SwingConstants.CENTER);
0042           } // End try/catch
0043        } // End if.
0044
0045        demoLabel.setBackground( Color.white);
0046        demoLabel.setForeground( Color.black);
0047        demoLabel.setOpaque( true);
0048        this.getContentPane().add( demoLabel);
0049     } // End init.
```

The first significant difference is the inclusion of the java.net package of classes, on line 0014, which will be required in order to have easy access to the facilities for using Internet resources. Within the *init()* method a check is made to ensure that the labelIcon has not already been constructed and, if not, the **try/catch** structure starting on line 0026 is executed. The first step, on line 0028, is an attempt to obtain an appletContext for **this** applet. This attempt will fail, throwing a NullPointerException, if the applet is executing as an application, and the appropriate exception handler, on lines 0037 to 0041, constructs the ImageIcon from a file on the local file store as before.

If no exception is thrown on line 0028 then the applet is executing within a browser and lines 0029 and 0030 prepare a URL (Uniform Resource Locator) instance to access the `shredder.gif` file at the Web location from where the applet was obtained. URLs and other java.net facilities will be considered in detail in Chapter 8. If for some reason it is not possible to obtain the URL connection a MalformedURLException will be thrown and the appropriate exception handler, on lines 0035 and 0036, construct the *demoLabel* as a JLabel instance containing only the text label "*destroy*". Otherwise, line 0031 constructs the *labelIcon* ImageIcon, passing *url* as the first argument (indicating that it should obtain its image across the Web) and lines 0032 and 0033 construct *demoLabel* using the icon and the text.

The effect of this **try/catch** structure is that *demoLabel* is guaranteed to be initialized, possibly containing only the text label if it is not possible for the image to be obtained. The remaining parts of the *init()* method configure and display the JLabel as before.

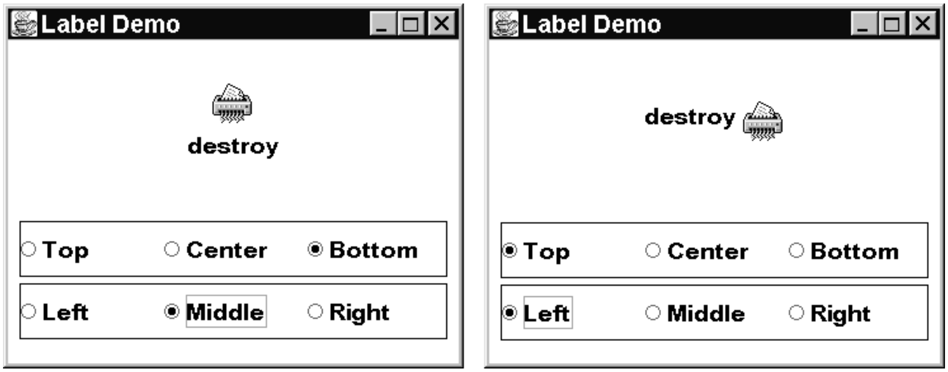


Figure 3.2 The *LabelDemo* applet.

3.3 Relative positioning of the icon text: introducing JRadioButtons

In order to explore the attributes that control the relative positioning of the icon and text in a JLabel instance, and in order to allow the JRadioButton class to be introduced, an artifact to allow them to be interactively explored will be designed and developed. The artifact, called *LabelDemo*, with suitable resources configured as described in the previous chapter, is illustrated in Figure 3.2.

The horizontalTextPosition attribute determines the horizontal location of the text relative to the icon and can have one of the values SwingConstants.LEFT, SwingConstants.CENTER (identified as "*middle*" in Figure 3.2.) or SwingConstants.RIGHT. The verticalTextPosition attribute determines the vertical location of the text relative to the icon and can have one of the values SwingConstants.TOP, SwingConstants.CENTER or SwingConstants.BOTTOM. As each of these attributes can be specified independently of the other, this gives a total of nine combinations of horizontal and vertical spacing. Only two of these are illustrated in Figure 3.2.

horizontalTextPosition & verticalTextPosition methods

```
public void setHorizontalTextPosition( int manifest)
public int  getHorizontalTextPosition()
public void setVerticalTextPosition( int manifest)
public int  getVerticalTextPosition()
```

Attributes introduced by the JLabel class, and independently by the AbstractButton class, which control the positioning of the text message relative to the image icon. Manifest values for horizontalTextPosition are SwingConstants.LEFT, SwingConstants.CENTER (default) or SwingConstants.RIGHT. Manifest values for verticalTextPosition are SwingConstants.TOP, SwingConstants.CENTER (default) or SwingConstants.BOTTOM.

The upper line of radio buttons in the artifact control the vertical positioning of the text and operate independently of the lower row of buttons, which control the horizontal

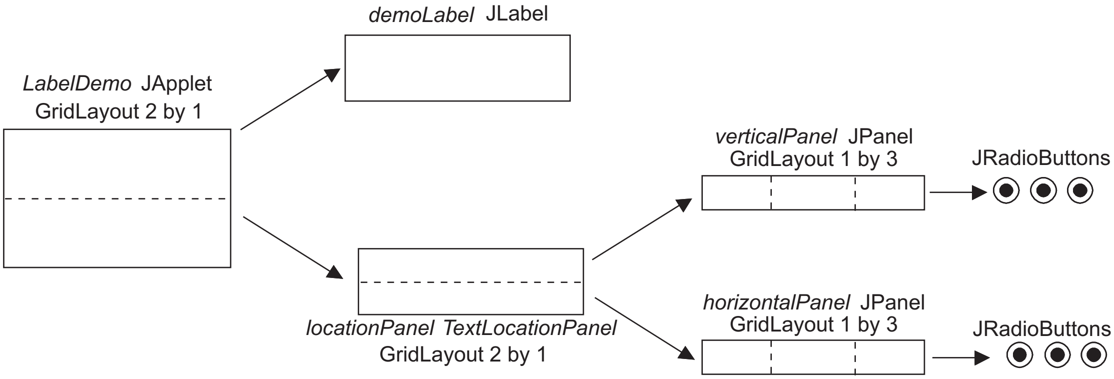


Figure 3.3 *LabelDemo* layout diagram.

positioning. To emphasize this distinction to the user a plain border has been provided around each of the two functional button groups.

This is illustrating the user interface design principle known as *grouping*. Each of the borders is indicating to the user that the three enclosed controls are connected together in some way and are distinct from the other three controls. Sets of radio buttons, in particular, should always be grouped together and never distributed across an interface.

The geometry relationships of the components that comprise this interface are illustrated in Figure 3.3. The JApplet window supplied by the *LabelDemo* class has a two row by one column GridLayout management policy. A GridLayout manager divides its space into a rectangular matrix of cells. Unlike a BorderLayout manager, but just like a FlowLayout manager, a GridLayout manager forces all of its instance children to be the same size.

The relative positioning of the Components that are added to a Container which has a GridLayout management policy is decided by the order in which they are added. The first Component is added into the top left cell, with subsequent Components added to adjacent cells to the right. When the first row is full the next Component is added in the leftmost location of the next row, and this pattern continues until all cells of the Container have been occupied.

The upper cell of the applet window contains a single JLabel instance called *demoLabel*. The lower cell contains an instance of the *TextLocationPanel*, called *locationPanel*, which also has a two row by one column GridLayout. The upper cell of the *locationPanel* contains a JPanel instance called *verticalPanel* and has a one row by three column GridLayout manager; each of its three cells contains a JRadioButton instance. The lower cell of the *locationPanel* also contains a JPanel instance, called *horizontalPanel*, with a one row by three column GridLayout manager; the remaining three JRadioButtons are mounted upon it.

This layout management arrangement will ensure that the upper half of the applet window is used to display the JLabel and the lower half the controls. The lower half is again divided horizontally exactly in half for the two rows of radio buttons. Furthermore, because both of the panels upon which the radio buttons are mounted must have exactly the same width and as they both have a GridLayout manager that will divide them vertically exactly into thirds, the six radio buttons will be vertically presented as three aligned columns.

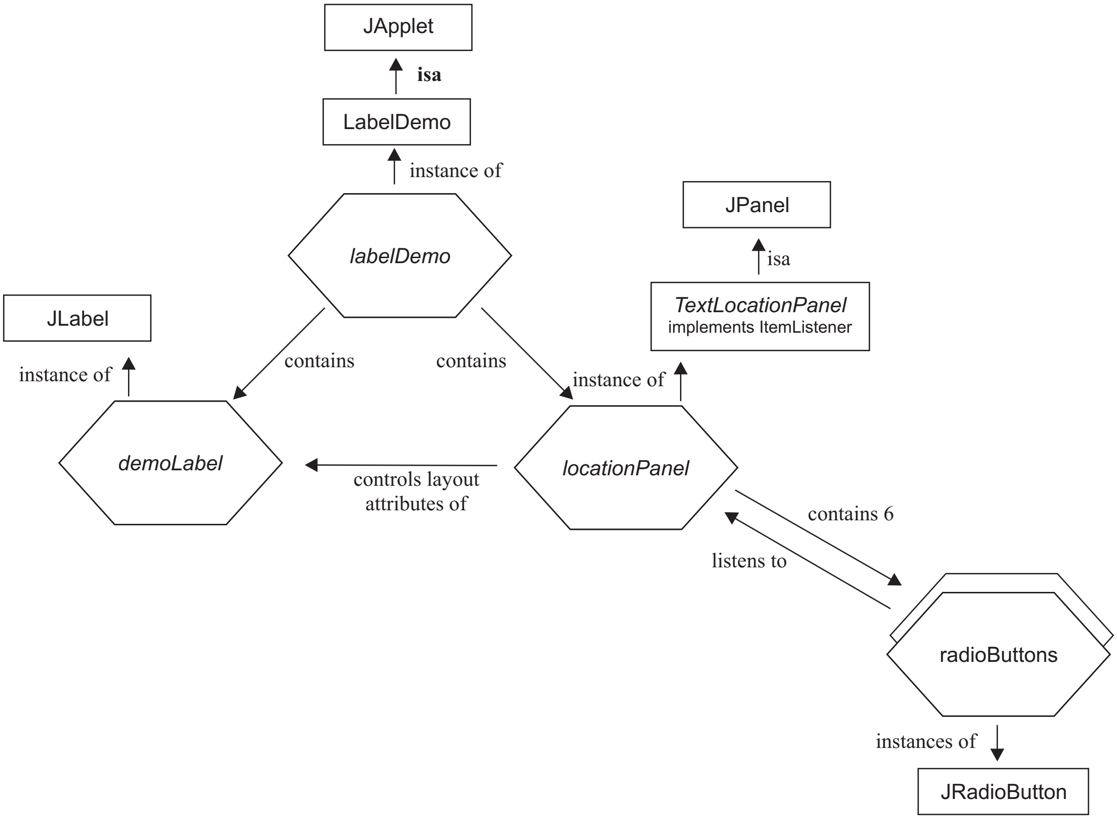


Figure 3.4 *LabelDemo* artifact: instance diagram.

An instance diagram for the *LabelDemo* artifact is given in Figure 3.4. It shows that the *LabelDemo* class supplies the *labelDemo* applet, containing an instance of the JLabel class, called *demoLabel*, whose layout attributes are controlled by an instance of the *TextLocationPanel* class, called *locationPanel*, also contained in the *applet*. The *locationPanel* contains six instances of the JRadioButton class and, as they dispatch ItemEvents when they are clicked, the *TextLocationPanel* class implements the ItemListener interface in order to be able to listen to them. As the purpose of this interface is to demonstrate the use of the JRadioButton class and the location attributes of the JLabel class, a separate translation class has not been provided.

The implementation of the *LabelDemo* class is as follows.

```
package chap3;

import javax.swing.*;

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

public class LabelDemo extends JApplet {

private final static ImageIcon labelIcon =
0018                              new ImageIcon( "shredder.gif",
0019                                             "shredder");
0020
0021    public void init() {
0022
0023    JLabel demoLabel = new JLabel( "destroy", labelIcon,
0024                                   SwingConstants.CENTER);
0025    TextLocationPanel locationPanel =
0026                            new TextLocationPanel( demoLabel);
0027
0028       this.setResources();
0029       this.getContentPane().setLayout( new GridLayout( 2, 1);
0030       this.getContentPane().add( demoLabel);
0031       this.getContentPane().add( locationPanel);
0032    } // End init.
0033
--   // details of setResources() and  main() omitted.
```

The declaration of this class commences by stating that it belongs to the *chap3* package of classes and continues with the **import**ation of the required swing and AWT classes. This time it includes the awt.event classes, as this artifact will have to respond to the user's interactions with it.

The *init()* action commences by constructing the *demoLabel* JLabel instance in an identical manner to the construction, and subsequent configuration, of the JLabel instance in the *FirstLabelDemo* artifact, as previously described. It then continues by passing the identity of this JLabel instance to the *TextLocationPanel* constructor in order that it can know which instance it is to control. Having constructed the two instances that it contains it calls its own *setResources()* method to configure its appearance. The contents of the *setResources()* method will be given later in the chapter.

On line 0029 it establishes a two row by one column GridLayout policy, without any spacing between the added components, for the Applet's contentPane, and then adds the instances to it. The four arguments to the GridLayout constructor, on line 0030, are the number of rows, the number of columns and the horizontal and vertical separation (in pixels) between the components which are added to it. The order in which instance children are added to a Container which has a GridLayout manager determines where they appear in relation to each other. In this example there are only two cells in the matrix, so the JLabel instance is added in the upper cell and the TextLocationPanel instance below it, as shown in Figure 3.2.

GridLayout constructors

Object → GridLayout

```
public GridLayout( int rows, int columns, int hgap, int vgap)
public GridLayout( int rows, int columns)
```

Constructs a GridLayout instance which will lay out its Container's children in equal-sized areas, using the number of rows and columns indicated. The location of the children is decided by the sequence in which they are added, starting at the top left and continuing in a right, down manner. Horizontal and vertical spacing between the children is determined by *hgap* and *vgap*.

The first part of the implementation of the *TextLocationPanel* class, also contained in the *chap3* package, is as follows.

```
0016 class TextLocationPanel extends JPanel implements ItemListener {
0017
0018 private JLabel controllingThis = null;
0019
0020     protected TextLocationPanel( JLabel toControl) {
0021
0022     JPanel verticalPanel   = new JPanel();
0023     JPanel horizontalPanel = new JPanel();
0024
0025     ButtonGroup verticalGroup   = new ButtonGroup();
0026     ButtonGroup horizontalGroup = new ButtonGroup();
0027
0028     JRadioButton topRadio    = new JRadioButton( "Top");
0029     JRadioButton centerRadio = new JRadioButton( "Center");
0030     JRadioButton bottomRadio =
0031                          new JRadioButton( "Bottom", true);
0032
0033     JRadioButton leftRadio   = new JRadioButton( "Left");
0034     JRadioButton middleRadio =
0035                           new JRadioButton( "Middle", true);
0036     JRadioButton rightRadio  = new JRadioButton( "Right");
```

The declaration of this class is not stated as **public**, so it can only be seen and used by other classes in the same package. It states that the *TextLocationPanel* class extends the JPanel class and implements the ItemListener interface, exactly as illustrated in its instance diagram. It contains a single instance attribute, declared on line 0018, which is the identity of the JLabel instance whose attributes are to be manipulated in reponse to the user's interactions.

The constructor, whose declaration starts on line 0020, requires a single argument of the JLabel class which provides the identity of the label to be controlled, as was described when the constructor was called on line 0027 of the *LabelDemo init()* method above. The constructor commences by declaring and constructing the two JPanel instances that will be mounted upon it, as illustrated in the layout management diagram, shown in Figure 3.3.

The constructor continues, on lines 0025 and 0026, with the declaration and construction of two instances of the ButtonGroup class, called *verticalGroup* and *horizontalGroup*. Instances of the ButtonGroup class determine which collections of JRadioButtons will operate as a set of radio buttons, allowing only one of them to be selected at a time and automatically deselecting the others when it is selected. As there are two independent sets of radio buttons in this interface, two instances of the ButtonGroup class will be required.

Object → javax.swing.ButtonGroup

```
public ButtonGroup()
public void add( AbstractButton toAdd);
public void remove( AbstractButton toRemove);
```

Constructs a ButtonGroup instance and maintains a set of AbstractButtons (including JRadioButtons) which it controls. Will ensure that only one button in the group can be selected at any instant by deselecting the existing selected button.

The constructor continues with the declaration and construction of the six JRadioButtons, on lines 0028 to 0036. The identifier for each button reflects the text label that will be associated with each of them on the interface and this is stated as the text argument to each of the constructors. Two of the constructors, for the *Bottom* button in the vertical buttons and for the *Middle* button in the horizontal group, have a second **boolean** argument (**true**) which ensures that they will be selected when the user first sees the interface. The constructor continues as follows.

JComponent → AbstractButton → JToggleButton → javax.swing.JRadioButton

```
public JRadioButton()
public JRadioButton( Icon icon)
public JRadioButton( Icon  icon, boolean selected)
public JRadioButton( String text)
public JRadioButton( String text, boolean selected)
public JRadioButton( String text, Icon icon)
public JRadioButton( String text, Icon icon, boolean selected)
```

Constructs a JRadioButton instance with an *icon* and/or *text* and possibly *selected*.

```
0038        this.setLayout( new GridLayout( 2, 1, 5, 5));
0039
0040        controllingThis = toControl;
0041        controllingThis.setVerticalTextPosition(
0042                                    SwingConstants.BOTTOM);
0043        controllingThis.setHorizontalTextPosition(
0044                                    SwingConstants.CENTER);
0045
0046        topRadio.addItemListener(    this);
0047        centerRadio.addItemListener( this);
0048        bottomRadio.addItemListener( this);
0049        leftRadio.addItemListener(   this);
0050        middleRadio.addItemListener( this);
0051        rightRadio.addItemListener(  this);
```

Line 0038 establishes a two row by one column GridLayout policy as shown on the layout diagram in Figure 3.3. Line 0040 then stores the identity of the JLabel button which is to be controlled, supplied in the *toControl* argument to the constructor, in the instance attribute *controllingThis*. The initial configuration of the radio buttons suggested that the text label would be positioned centrally below the icon. Lines 0041 to 0044 use the appropriate state-setting methods and arguments to ensure that this will be the case when the interface is first made visible to the user, as shown in the left-hand image of Figure 3.2. The vertical positioning of the text label is determined by the value of its verticalTextPosition attribute, and the call of the setVerticalTextPosition() method, with the argument StringConstants.BOTTOM, causes the text to be positioned below the icon. In a similar manner the horizontalTextPosition attribute determines the horizontal location of the text label and the call of setHorizontalTextPosition() method, with the argument StringConstants.CENTER, causes the text to be positioned in line with the icon.

This part of the constructor concludes by establishing the itemListener resource of each radio button by using its addItemListener() method. The addItemListener() method is comparable to the JButton addActionListener() method, described in Chapter 1, establishing the identity of the listener object which will receive the ItemEvents generated by the JRadioButtons when the user selects them. In this example the *TextLocationPanel* instance is listening to itself, so the argument supplied is **this**. The constructor continues as follows.

```
0053        topRadio.setHorizontalAlignment(    SwingConstants.LEFT);
0054        centerRadio.setHorizontalAlignment( SwingConstants.LEFT);
0055        bottomRadio.setHorizontalAlignment( SwingConstants.LEFT);
0056        leftRadio.setHorizontalAlignment(   SwingConstants.LEFT);
0057        middleRadio.setHorizontalAlignment( SwingConstants.LEFT);
0058        rightRadio.setHorizontalAlignment(  SwingConstants.LEFT);
0059
0060        verticalGroup.add( topRadio);
0061        verticalGroup.add( centerRadio);
0062        verticalGroup.add( bottomRadio);
0063
0064        horizontalGroup.add( leftRadio);
0065        horizontalGroup.add( middleRadio);
0066        horizontalGroup.add( rightRadio);
```

JRadioButtons share many attributes with the JLabel class, one of which is the horizontalAlignment attribute, which determines where within its area it is to display itself. In order to achieve the visual layout shown in Figure 3.2 it is necessary that the radio buttons are horizontally left aligned. This is accomplished, in lines 0053 to 0058, by calling each button's setHorizontalAlignment() method in turn.

Lines 0060 to 0062 then use the add() method of the *verticalGroup* ButtonGroup instance to ensure that the upper three buttons on the interface will operate as a single group of radio buttons. Lines 0064 to 0066 then do the same for the lower three buttons, adding them to the *horizontalGroup*.

At this stage the six radio buttons and the two ButtonGroups have been constructed and configured. The remaining part of the constructor, which follows, is concerned with configuring the two intermediate panels and then assembling the interface using the instance relationships shown in Figure 3.3.

```
0068       verticalPanel.setBorder(
0069                      new LineBorder( Color.black, 1));
0070       verticalPanel.setLayout(
0071                      new GridLayout( 1, 3, 5, 5));
0072
0073       verticalPanel.add( verticalTopRadio);
0074       verticalPanel.add( verticalCenterRadio);
0075       verticalPanel.add( verticalBottomRadio);
0076
0077       horizontalPanel.setBorder(
0078                       new LineBorder( Color.black, 1));
0079       horizontalPanel.setLayout(
0080                         new GridLayout( 1, 3, 5, 5));
0081
0082       horizontalPanel.add( horizontalLeftRadio);
0083       horizontalPanel.add( horizontalCenterRadio);
0084       horizontalPanel.add( horizontalRightRadio);
0085
0086        this.add( verticalPanel);
0087        this.add( horizontalPanel);
0088     } // End TextLocationPanel constructor;
```

The fragment commences on line 0068 and 0069 by establishing a plain, single line, black border for the *verticalPanel*. A detailed description of the border facilities supplied by the swing.border package, which has been imported by this class, will be given later in this chapter. Lines 0070 and 0071 then establish a one row by three column GridLayout manager for the *verticalPanel*, before the three vertical JRadioButtons are added to it on lines 0073 to 0075. Lines 0077 to 0084 are essentially identical, but concerned with the horizontal radio buttons. The constructor concludes, on lines 0086 and 0087, by adding the *verticalPanel* and *horizontalPanel* to **this** *TextLocationPanel* currently being constructed.

When the user selects one of the JRadioButtons on the interface an ItemEvent is generated and dispatched to its registered ItemListener. This listener object must announce that it implements the ItemListener interface and, in order for this to be allowed by Java, it must supply an *itemStateChanged()* method. This is comparable to objects which announce that they implement the ActionListener interface having to supply an *actionPerformed()* method. The dispatching of an ItemEvent from a JRadioButton to its listener is effected by the *itemStateChanged()* method in its listener object being called with the ItemEvent passed as its argument.

An ItemEvent has a number of attributes. The most significant in this example is the item attribute, which can be obtained using the ItemEvent's getItem() method. This will return the identity of the JRadioButton which generated the event and, having obtained its identity, the text which it is displaying can be obtained with its getText() method. Once the label of the radio button is known the appropriate actions can be taken, as follows.

```
0091     public void itemStateChanged( ItemEvent event) {
0092
0093     String location =
0094                     ((JRadioButton) event.getItem()).getText();
0095
0096        if ( location.equals( "Top")) {
0097           controllingThis.setVerticalTextPosition(
0098                                           SwingConstants.TOP);
0099        } else if (location.equals( "Center")) {
0100            controllingThis.setVerticalTextPosition(
0101                                        SwingConstants.CENTER);
0102        } else if (location.equals( "Bottom")) {
0103           controllingThis.setVerticalTextPosition(
0104                                        SwingConstants.BOTTOM);
0105
0106        } else if (location.equals( "Left")) {
0107           controllingThis.setHorizontalTextPosition(
0108                                          SwingConstants.LEFT);
0109         } else if ( location.equals( "Middle")) {
0110            controllingThis.setHorizontalTextPosition(
0111                                              SwingConstants.CENTER);
0112         } else if ( location.equals( "Right")) {
0113            controllingThis.setHorizontalTextPosition(
0114                                               SwingConstants.RIGHT);
0115         } // End if.
0116      } // End itemStateChanged.
0117 } // End class TextLocationPanel;
```

Line 0091 is the required prototype of an itemStateChanged() method which must be supplied in order for a class to implement the ItemStateChanged interface. On lines 0093 and 0094 the getItem() method of the *event* passed as argument is used to obtain the identity of the Component which generated the event. This Component, in this example, is known to be an instance of the JRadioButton class which is cast to this class once it is retrieved. The final term in the expression calls the getText() method of the retrieved JRadioButton to obtain the text of the radio button which was passed as a String instance, which is finally stored in the *location* local variable.

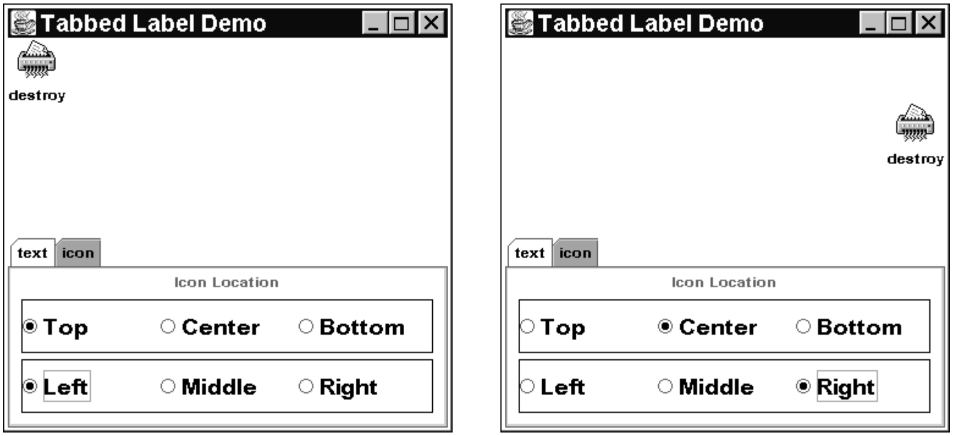
The rest of the method is a six-way selection containing a branch for each possible radio button. Within each branch either the setVerticalTextPosition() or setHorizontalTextPosition() method of the JLabel instance passed to the constructor and stored in the *controllingThis* instance variable is called, passing an appropriate value from the SwingConstants package.

The effect of this method call is to change one of the attributes that determine the spatial relationship between the icon and its text. The JLabel instance responds by immediately changing its visual appearance to comply with the changed attribute. The right-hand image in Figure 3.2 shows that the verticalTextPosition attribute has been changed from BOTTOM to TOP and that its horizontalTextPosition attribute has been changed from CENTER (middle) to LEFT.

For example, if the user were to press the button labeled *Middle* an itemEvent would be generated and passed to the *itemStateChange()* method. The first step of this method would be to retrieve the text of the button, *Middle*, and hence the part of the **if** selection structure which calls the button's setHorizontalTextPosition() method with SwingConstants.CENTER as the argument would be executed. The visible effect of changing the horizontalTextPosition attribute is that the button would redraw itself with the text label *destroy* positioned immediately above the icon. At the same time, because all of the lower buttons are in the same ButtonGroup, the button labeled *Left* would appear deselected and the button labeled *Middle* would become selected.

3.4 Positioning of the icon: introducing JTabbedPanes

In addition to the horizontalTextPosition and verticalTextPosition attributes, which determine the relative location of the text label and the icon; two additional attributes, horizontalAlignment and verticalAlignment, independently determine the location of the icon and text within the extent of the JLabel's window. In this section the effects of these attributes will be introduced using an artifact called *TabbedLabelDemo*, which is developed from the *LabelDemo* artifact just described. The appearance of this first version of the *TabbedLabelDemo*'s interface and the effect of these two attributes is shown in Figure 3.5.



Tabbed Label Demo

destroy

text | icon

Icon Location

● Top ○ Center ○ Bottom

● Left ○ Middle ○ Right

Tabbed Label Demo

destroy

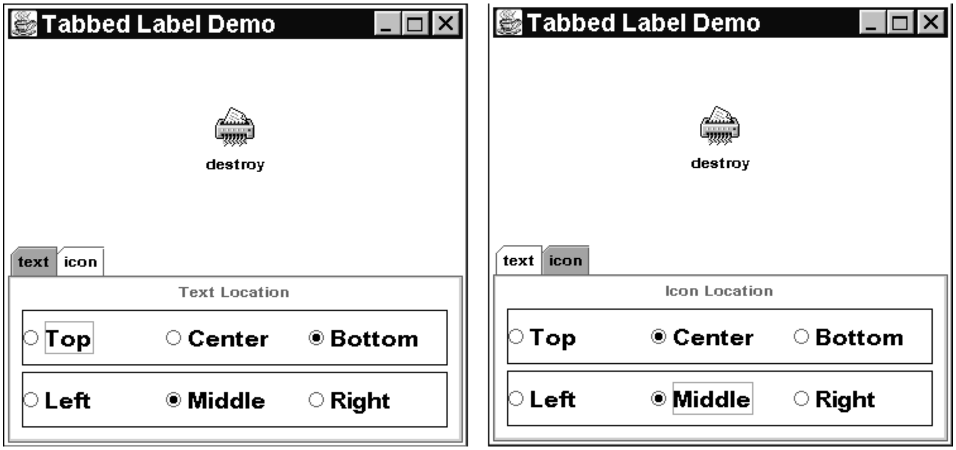
text | icon

Icon Location

○ Top ● Center ○ Bottom

○ Left ○ Middle ● Right

Figure 3.5 *TabbedLabelDemo* showing icon positioning.



Tabbed Label Demo

destroy

text | icon

Text Location

○ Top ○ Center ● Bottom

○ Left ● Middle ○ Right

Tabbed Label Demo

destroy

text | icon

Icon Location

○ Top ● Center ○ Bottom

○ Left ● Middle ○ Right

Figure 3.6 *TabbedLabelDemo*, showing both Panels.

In the left-hand image the radio buttons on the panel labeled *Icon Location* have been selected so as to set the value of verticalAlignment to SwingConstants.TOP and the horizontalAlignment to SwingConstants.LEFT. In the right-hand image these attributes have been set to CENTER and RIGHT respectively. The JLabel instance shown above the tabbed pane has responded appropriately.

Figure 3.5 also indicates that a JTabbedPane has been used to provide the layout management of the lower part of the interface. A tabbed pane consists of a number of tabs, in this example two labeled *text* and *icon*, each of which is associated with a Component which only becomes visible when its tab is selected. In the *TabbedLabelDemo* artifact there are two extended JPanels associated with the two tabs. An instance of the *TextLocationPanel* in the previous section is associated with the *text* tab. An instance of the *IconPositionPanel*, which is very similar to the *TextLocationPanel*, is associated with the *icon* tab. Figure 3.6 illustrates the two appearance of the panels.

Figure 3.7 is an instance diagram for the *TabbedLabelDemo* artifact. It shows that the *applet* is an instance of the *TabbedLabelDemo* class and that it contains an instance

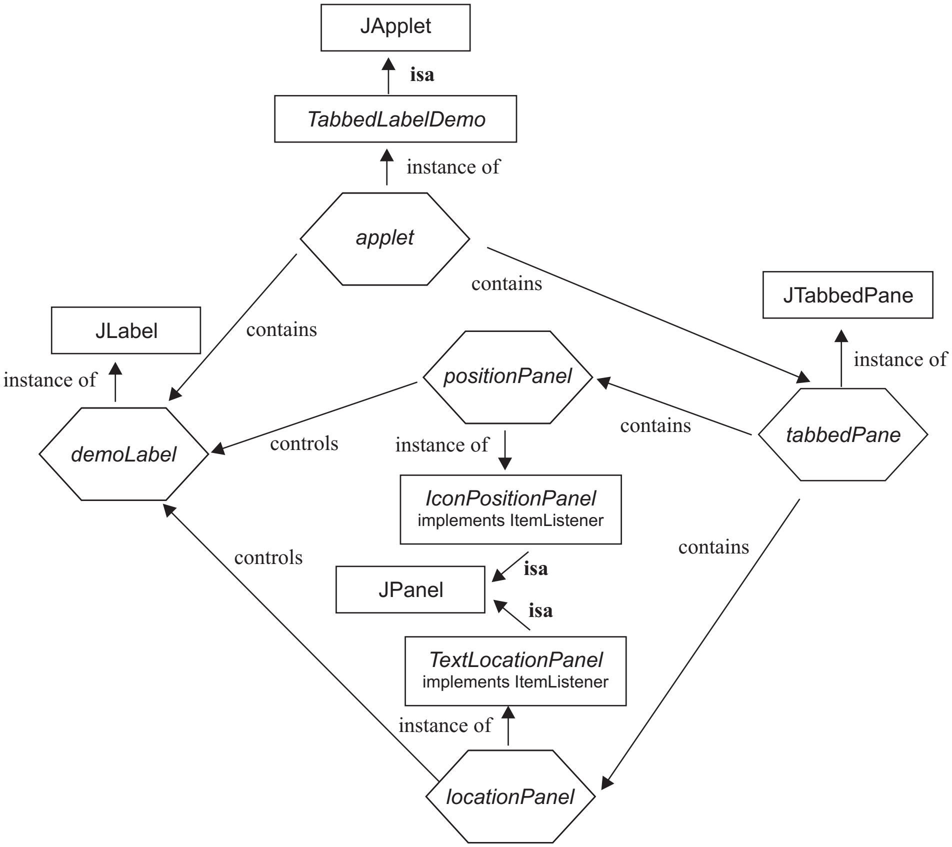


Figure 3.7 *TabbedLabelDemo* instance diagram.

of the JLabel class called *demoLabel* and an instance of the JTabbedPane class called *tabbedPane*. The *tabbedPane* contains two components, an instance of the *TextLocationPanel* class, identified as *locationPanel*, and an instance of the *IconLocationPanel* class, called *positionPanel*. Both of these components control the value of attributes in the *demoLabel* instance and both of their classes are extended from the JPanel class.

The only difference in the *TextLocationPanel* as used in this artifact is the provision of a title at the top, intended to reinforce to the user the different function of the two panels, which might not always be immediately obvious from the selected tab. This is accomplished by installing a TitledBorder into it. A TitledBorder is a compound border which, in this example, contains a EmptyBorder and a title. The construction and installation of this border is accomplished within the constructor of the *TextLocationPanel* as follows.

```
this.setBorder( new TitledBorder( new EmptyBorder( 5, 5, 5, 5),
                                  "Text Location",
                                  TitledBorder.CENTER,
                                  TitledBorder.TOP));
```

The setBorder() method is being used to install a border into the *TextLocationPanel*, **this**, which is currently being constructed. The first argument to the TitledBorder constructor is the Border which is to have the title inserted into it. This is an EmptyBorder, i.e. one that will reserve space for itself but will not draw anything within that space. The second argument is the title to be used. The third and fourth arguments position the title, using manifest values from the TitledBorder class; other possible values that can be used to position the title will be introduced later in this chapter when borders are discussed in detail.

The construction of the *IconPositionPanel* is essentially identical to the construction of the *TextPositionPanel*, as previously described. The two rows of JRadioButtons are again assembled as two independent groups, each group mounted on its own intermediate panel which has a simple one pixel line border. The upper row controls the JButton instance's verticalAlignment attribute by calling its setVerticalAlignment() method with one of the arguments SwingConstants.TOP, SwingConstants.CENTER or SwingConstants.BOTTOM, as appropriate. The lower group controls the horizontalAlignment attribute by calling the setHorizontalAlignment() method, with one of the arguments SwingConstants.LEFT, SwingConstants.CENTER or SwingConstants.RIGHT as appropriate.

horizontalAlignment and **verticalAlignment** methods

```
public void setHorizontalAlignment( int manifest)
public int  getHorizontalAlignment()
public void setVerticalAlignment( int manifest)
public int  getVerticalAlignment()
```

Attributes introduced by the JLabel class, and independently by the AbstractButton class, which control the positioning of the text and icon within the component's window. Manifest values for horizontalAlignment are SwingConstants.LEFT, SwingConstants.CENTER (default) or SwingConstants.RIGHT. Manifest values for verticalAlignment are SwingConstants.TOP, SwingConstants.CENTER (default) or SwingConstants.BOTTOM.

The *JTabbedDemo*'s *init()* method constructs an instance of both of these classes adding them to an instance of the JTabbedPane class, located below a JLabel instance, as follows.

```
public void init() {

JLabel demoLabel  = new JLabel( "destroy", labelIcon,
                               SwingConstants.CENTER);
JTabbedPane tabbedPane = new JTabbedPane();

  this.setResources();
  this.getContentPane().setLayout(
                               new GridLayout( 2, 1,));

  demoLabel.setOpaque( true);


  tabbedPane.addTab("text",
                    new TextLocationPanel( demoLabel));
  tabbedPane.addTab("icon",
                    new IconPositionPanel( demoLabel));

0026        tabbedPane.setSelectedIndex( 0);
0027
0028        this.getContentPane().add( demoLabel);
0029        this.getContentPane().add( tabbedPane);
0030     } // End init.
```

Lines 0010 and 0011 construct an instance of the JLabel class called *demoLabel* as before. Line 0013 then constructs an instance of the *JTabbedPane* class called *tabbedPane*, using its default constructor and *setResources()* is called on line 0014 to configure the visual appearance of the artifact. A two row by one column GridLayout is established for the JApplet's contentPane, on lines 0015 and 0016, before the *demoLabel*'s opacity is configured as before.

Lines 0021 and 0022 construct an anonymous instance of the *TextLocationPanel* and add it to the *tabbedPane*, with a tab labeled *text* associated with it. The JTabbedPane addTab() method adds a new component to the pane, the first argument is the text label to be shown on the tab and the second argument is the component which is to be shown when the user selects the tab.

JComponent → javax.swing.JTabbedPane

```
public JTabbedPane()
public JtabbedPane( int tabPosition);
```

Constructs an empty JTabbedPane layout manager with the tabs in the default (TOP) position or with position (TOP, BOTTOM, LEFT, RIGHT) as specified.

```
public void addtab( String title, Component toAdd);
public void addtab( String title, Icon icon, Component toAdd);
public void addtab( String title, Icon icon,
                    Component toAdd, String tip);
public void inserttab( String title, Icon icon,
                       Component toAdd, String tip, int index);
```

Allows tabs with associated Components and possibly with an icon and/or a tool tip to be added to the pane. When a tab is selected the associated Component becomes visible.

```
public void setSelectedIndex(int index)
public int  getSelectedIndex();
public void setSelectedComponent( Component toSelect)
public Component getSelectedComponent()
```

Allows the currently selected Component to be set or queried.

Lines 0023 and 0024 add a second tab to the *tabbedPane*, this time an anonymous instance of the *IconPositionPanel*, with the associated tab *icon*. The *init()* method continues by calling the *tabbedPane*'s setSelectedIndex() method. The argument (0) to this call indicates that the first tab added should be selected and hence the component associated with the tab will be visible when the applet is first shown to the user. The last actions of *init()* are to add the *demoLabel* into its upper GridLayout location and the *tabbedPane* into the lower location.

Following execution of the *init()* method, as the applet is started, the artifact will become visible to the user in the configuration shown in the left-hand image of Figure 3.6.

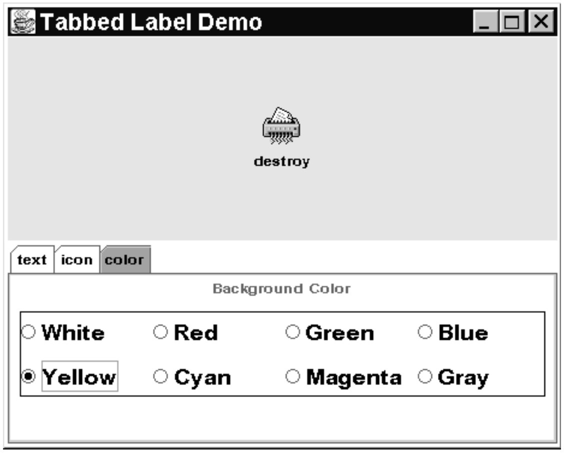


Figure 3.8 *TabbedLabelDemo* showing the color panel.

If the user then presses the *icon* tab the component associated with that tab, an instance of the *IconPositionPanel*, will be become visible, as shown in the right-hand image of Figure 3.6. Whichever of the panels is visible the user can press one of the radio buttons and the appearance of the JLabel instance will change accordingly.

3.5 The background Color attribute – standard colors

In this part of the chapter a further tab will be added to the *TabbedLabelDemo* artifact which will control the background Color attribute, allowing the user to choose between a small number of standard colors. The appearance of the *TabbedLabelDemo* artifact when the *color* tab has been selected and a different background color has been selected is shown in Figure 3.8.

The *color* tab is associated with an instance of the *ColorPanel* class whose design and implementation does not differ significantly from that of the *TextLocationPanel*. It contains eight JRadio buttons, mounted on an intermediate panel with a two row by four column GridLayout, all of which are associated with a single ButtonGroup.

Each JRadio button has the *ColorPanel* instance, which it is part of, registered as its ItemListener resource, and so will dispatch an ItemEvent to its own *itemStateChanged()* method when the user selects it. The *ColorPanel itemStateChanged()* method commences by retrieving the String associated with the JRadio button and continues with an eight-way **if/else if** structure, containing a selection for each of the standard colors. Each selection consists of a call of the *controllingThis* setBackground() method. The argument to this method is an instance of the Color class, which is supplied by the AWT for the definition of colors. In this version of the *ColorPanel* one of the manifest constants provided by the Color class is used as an argument. For example, the branch associated with the "*Yellow*" radio button is implemented as follows.

```
} else if ( theColor.equals( "Yellow")) {
   controllingThis.setBackground( Color.yellow);
} . . .
```

Object → java.awt.Color

```
public Color( float red, float green, float blue)
public Color( int red,   int green,   int blue)
public Color( int alphaRedGreenBlue)
```

Constructs a Color from floating-point red/green/blue intensities in the range 0.0 to 1.0, integer values in the range 0 to 255 (0x00 to 0xFF), or by taking *alphaRedGreenBlue* as eight-bit values from the 32 bit integer.

```
black    blue   cyan    darkGray  gray  lightGray
magenta  green  orange  pink      red   yellow     white
```

Constants supplied by the Color class.

```
public int getRed();
public int getGreen()
public int getBlue()
```

Obtains the red, green or blue component of the Color in the range 0 to 255.

```
public Color lighter();
public Color darker();
```

Constructs and returns a lighter or darker hue of the Color.

Unlike the attribute changes caused by the text and icon panels, which caused the *demoLabel* to automatically redisplay itself, changing the *background* color attribute is not immediately visible. Consequently a call of *controllingThis.repaint()* at the end of the *itemStateChanged()* method has to be included to make the change visible.

Background color and opacity

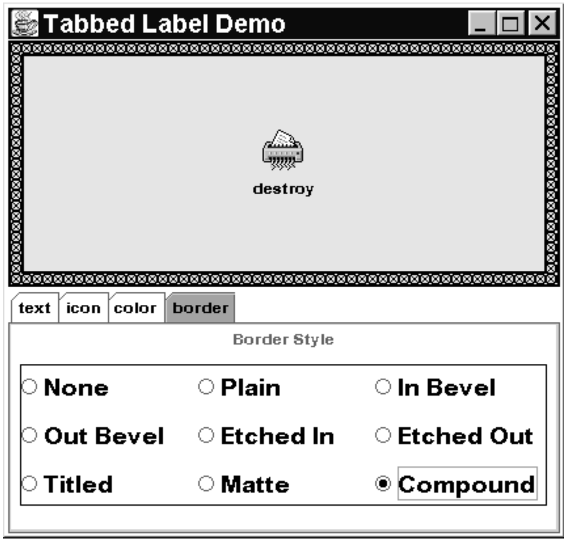
```
public void  setBackground( Color setTo)
public Color getBackground()
```

Introduced into the hierarchy by the Component class and determines the background Color of the Component's window.

```
public void    setOpaque( boolean opaqueOrNot)
public boolean isOpaque()
```

Introduced into the hierarchy by JComponent. An opaque component will be rendered in its entirety, occluding anything underneath it. A non-opaque component will not render any parts which would be displayed in the background color, allowing any components underneath to be visible.

The *init()* action of *TabbedLabelDemo* would have to be extended to create an instance of the *ColorPanel* class and add it to the *tabbedPane*, using its addTab() method, in order to produce the interface shown in Figure 3.8. The instance diagram for this version of *TabbedLabelDemo* does not differ significantly from that of the previous version given in Figure 3.7. The revised diagram would show that the *tabbedPane* contains an additional component, of the *ColorPanel* class. Like the other two components, the new component will control the attributes of the JLabel *demoLabel* instance.



Border Style

○ None	○ Plain	○ In Bevel
○ Out Bevel	○ Etched In	○ Etched Out
○ Titled	○ Matte	● Compound

Figure 3.9 *TabbedLabelDemo* showing the border panel.

This version of the *ColorPanel* allows the user to select between a small number of standard colors. In a later section of this chapter it will be extended to allow the user to interactively indicate more precisely exactly which color they require.

3.6 Borders

This extension to the *TabbedLabelDemo* adds an instance of the *BorderPanel* class to the *tabbedPane* with an associated tab labeled *border*. The appearance of this version of the interface is shown in Figure 3.9.

The plain, titled and empty border capabilities of all JFC components have already been briefly introduced. The *plain*, *in-bevel*, *out-bevel* and *etched-in* and *etched-out* borders are simple borders supplied by the LineBorder, BevelBorder and EtchedBorder JFC classes. The exaggerated appearance of these borders is illustrated in Figure 3.10. The plain border is provided as a simple outline rectangle of a solid color, darker than the component's background color. The beveled borders provide a pseudo-three-dimensional appearance. For the in-bevel it is imagined that there is a light source to the top left of the component, and that it is recessed into the screen. Accordingly the top and left of the component are shown in darker hues than the background, as they are in shadow, and the right and bottom are shown in lighter hues, as they are imagined to be more directly in the light. An out-bevel is similar, but it is imagined that it projects from the screen, reversing the pattern of hues. The etched border contains a single outlined rectangle of a darker, or lighter, hue inscribed within the limits of the component.

The *titled* border is a compound border, supplied by the TitledBorder class, which contains a border and a title inscribed within it. The *matte* border is supplied by the MatteBorder class and is implemented by the tessellation of an Icon supplied to its constructor. The *Compound* border is supplied by the CompoundBorder class, which allows different borders to be inscribed within each other.

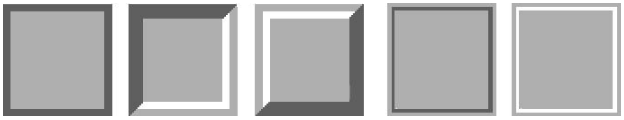


Figure 3.10 Exaggerated simple borders.

The five borders shown in Figure 3.10, and the absence of a border associated with the *None* button, are installed by the first part of the *BorderPanel*'s *itemStateChanged()* method, given below. The *BorderPanel*'s constructor does not differ significantly from the previous constructors. It creates nine JRadioButton instances and installs them on a panel with a three by three GridLayout, all of which are members of the same ButtonGroup and all of which dispatch ItemEvents to their own *itemStateChanged()* method. The *init()* action of the *TabbedLabelDemo* class has also been extended to construct an instance of the *BorderPanel* class and adds it, with the tab *border*, to the *tabbedPane*. This produces the appearance shown in Figure 3.9.

```
0084    public void itemStateChanged( ItemEvent event) {
0085
0086    String borderType = ((JRadioButton)
0087                                    event.getItem()).getText();
0088
0089      if (borderType.equals( "None")) {
0090          controllingThis.setBorder(
0091                      new EmptyBorder( new Insets(2, 2, 2, 2)));
0092
0093       } else if (borderType.equals( "Plain")) {
0094          controllingThis.setBorder(
0095                            new LineBorder( Color.black, 2));
0096
0097       } else if (borderType.equals( "In Bevel")) {
0098          controllingThis.setBorder(
0099                      new BevelBorder( BevelBorder.LOWERED));
0100
0101       } else if (borderType.equals( "Out Bevel")) {
0102          controllingThis.setBorder(
0103                       new BevelBorder( BevelBorder.RAISED));
0104
0105       } else if (borderType.equals( "Etched In")) {
0106          controllingThis.setBorder(
0107                 new EtchedBorder( EtchedBorder.LOWERED));
0108
0109       } else if (borderType.equals( "Etched Out")) {
0110          controllingThis.setBorder(
0111                 new EtchedBorder( EtchedBorder. RAISED));
```

The declaration commences, on line 0084, with the required prototype of an *itemStateChanged()* method and continues, on lines 0086 and 0087, by retrieving the String associated with the JRadioButton which generated the *event*. The first branch of the **if/else if** structure, commencing on line 0089, responds to the selection of the *None* button by installing an EmptyBorder into the JLabel instance *controllingThis*, using its setBorder() method. An EmptyBorder is a border that reserves space for itself and prevents anything else from drawing in that space. The amount of space reserved is defined by the Insets argument to the constructor, as shown on line 0091. An AWT Insets instance contains four **public int** attributes called top, left, bottom and right which define the amount of space (in pixels) to be reserved at each edge of the component. In this example all four of these insets are specified as two pixels. The difference between a **null** border and an EmptyBorder is that having no border at all, i.e. a **null** border, does not reserve any space at the edges of the component, so there may be no visible separation between components. However, an EmptyBorder will reserve the amount of space determined by Insets within the extent of component, and upon which nothing will be drawn.

A LineBorder differs from an EmptyBorder by having an equal-sized inset at each edge of the component, defined by the second argument to its constructor as shown on line 0095. The first argument to the constructor specifies the solid color that is to be used to draw the border and the second is the size of the border in pixels. In this example a two-pixel black border is installed into the JLabel, on lines 0094 and 0095, when the user selects the *Plain* button on the BorderPanel.

The next two selections, between lines 0097 to 0103, install instances of the BevelBorder class. The difference between an in-border (depressed) and an out-border (raised) is determined by the manifest argument to the constructor, as shown on lines 0099 and 0103. This constructor supplies only a two-pixel border; wider borders have to be constructed as CompoundBorders containing a number of BevelBorders. The final borders in this part of the method are the EtchedBorder instances constructed and installed between lines 0105 and 0111, using techniques similar to those used for the BevelBorders.

Object → AbstractBorder → javax.swing.border.EmptyBorder

```
public EmptyBorder( Insets insets)
public EmptyBorder( int top, int left, int bottom, int right)
```

Constructs a border, with insets as specified, which will reserve space for itself but will not draw anything in it.

Object → AbstractBorder → javax.swing.border.LineBorder

```
public LineBorder( Color color)
public LineBorder( Color color, int thickness)
```

Constructs a border shown as a simple rectangle draw in the *color* specified; *thickness* defaults to 1 unless otherwise specified.

Object → AbstractBorder → javax.swing.border.BevelBorder

```
public BevelBorder( int manifest)
public BevelBorder( int manifest, Color highlight, Color shadow)
public BevelBorder( int manifest,
                    Color outerhighlight, Color innerHighlight,
                    Color outerShadow,    Color innerShadow)
```

Constructs a pseudo-three-dimensional border with colors derived from the background color, or as specified. The first, *manifest*, argument can be BevelBorder. LOWERED or BevelBorder.RAISED.

Object → AbstractBorder → javax.swing.border.EtchedBorder

```
public EtchedBorder( int manifest)
public EtchedBorder( int manifest, Color highlight, Color shadow)
public EtchedBorder( int manifest,
                     Color outerhighlight, Color innerHighlight,
                     Color outerShadow,    Color innerShadow)
```

Constructors as for the BevelBorder, as given above.

Instances of the TitledBorder have already been introduced, when titles were added to the simple LineBorder used to surround the Panels in the *tabbedPane*. The TitledBorder installed in response to the user pressing the "*Titled*" button is constructed on lines 0113 to 0116, as follows.

```
0113       } else if ( borderType.equals( "Titled")) {
0114         controllingThis.setBorder( new TitledBorder(
0115                                 new LineBorder( Color.black, 1),
0116                                                           "Title"));
```

The arguments to this TitledBorder constructor are the border into which the title is to be inserted, in this case a single-pixel LineBorder, and the title which is to be used. By default the title is shown at the top left of the component and the horizontal extent of the characters used to render it is added to the top of the Insets. As was shown in the previous TitledBorder constructors, more complex constructors can also be used which allow the position of the title to be varied. For example, the installation of the "*Border Style*" title into the BorderPanel, as shown in Figure 3.9, was accomplished as follows.

```
0036       this.setBorder( new TitledBorder(
0037                            new LineBorder(Color.black,1),
0038                                           "Border Style",
0039                                           TitledBorder.CENTER,
0040                                           TitledBorder.TOP));
```

The third argument controls the horizontal positioning of the title and the fourth its vertical positioning. Both of these arguments are specified using manifest values from the TitledBorder class: including, LEFT, CENTER, RIGHT, TOP and BOTTOM.

Object → AbstractBorder → javax.swing.border.TitledBorder

```
public TitledBorder( Border border)
public TitledBorder( Border border, String title)
public TitledBorder( Border border, String title,
                     int titleJustification, int titlePosition)
public TitledBorder( Border border, String title,
                     int titleJustification, int titlePosition,
                     Font titleFont)
public TitledBorder( Border border, String title,
                     int titleJustification, int titlePosition,
                     Font titleFont, Color titleColor)
```

Constructs a TitledBorder inserting the *title* into the *border* supplied, possibly using the *titleFont* and *titleColor* supplied. The *titleJustification* argument may take one of the

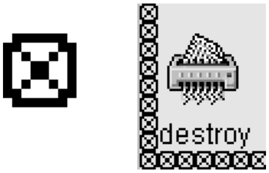


Figure 3.11 The matte icon and tessellation detail.

values LEFT (default), CENTER, RIGHT. The *titlePosition* argument may take one of the values ABOVE_TOP, TOP (default), BELOW_TOP, ABOVE_BOTTOM, BOTTOM or BELOW_BOTTOM. All of these manifest values are supplied by *TitledBorder*. State setting and enquiry actions for the title, titleColor, titleFont, titlePosition and titleJustification attributes are supplied.

A MatteBorder is constructed by tessellating an Icon, supplied to its constructor, within the insets also supplied to its constructor. The next part of the *BorderPanel itemStateChanged()* method, shown below, uses an Icon, called *matteIcon* and an eight-pixel inset at each edge. The *matteIcon* is an instance of the ImageIcon class, constructed at the start of the *BorderPanel* class, also shown below, from a GIF file called `matteX.gif`. This is the same technique as was used in the *TabbedLabelDemo* class to construct the *imageIcon*.

```
private static final ImageIcon matteIcon
                              = new ImageIcon( "matteX.gif");
      } else if ( borderType.equals( "Matte")) {
         controllingThis.setBorder(
                    new MatteBorder( 8, 8, 8, 8, matteIcon));
```

Figure 3.11 shows the `matteX.gif` file and the detailed appearance of the MatteBorder which it is tessellated within. It also illustrates, by the positioning of the icon within the extent of the component's window, the effects of the left and bottom insets. The positioning of the shredder image and the rendering of the "*destroy*" text is such that they do not occlude the matte border.

Object → AbstractBorder → javax.swing.border.MatteBorder

```
public MatteBorder( int top, int left,
                    int bottom, int right, Icon icon)
public MatteBorder( int top, int left,
                    int bottom, int right, Color color)
```

Constructs a border by tessellating the icon within the insets specified, or by filling with the color specified.

The final border supplied by the BorderPanel is an instance of the CompoundBorder class. A CompoundBorder instance contains two Borders, one or both of which can themselves be CompoundBorders, allowing multiple borders to be inscribed within each other. The border supplied by the "*Compound*" button in the BorderPanel, illustrated in Figure 3.9, consists of two two-pixel LineBorders sandwiching a matte LineBorder constructed within the *itemStateChanged()* method, as follows.

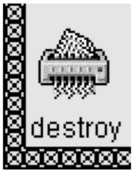


Figure 3.12 *Compound* borders as supplied by *BorderPanel*.

```
0118        } else if ( borderType.equals( "Compound")) {
0119           controllingThis.setBorder( new CompoundBorder(
0120                  new CompoundBorder(
0121                     new LineBorder( Color.black, 1),
0122                     new MatteBorder( 8, 8, 8, 8, matteIcon)),
0123                  new LineBorder( Color.black, 1)));
0124        } // End if.
0125        controllingThis.repaint();
0126     } // End itemStateChanged.
```

Lines 0119 to 0123 create and install the *Compound* border example; the arguments to the CompoundBorder constructor, on line 0120, being the CompoundBorder constructed on lines 0120 to 0122 and the LineBorder constructed on line 0123. The inner CompoundBorder (lines 0120 to 0122) consists of a LineBorder constructed on line 0121 and a MatteBorder on line 0122. The detailed appearance of this border is illustrated in Figure 3.12. As suggested in the listing, on line 0125, changing the border attribute of a component does not automatically cause it to redisplay itself with the changed border, and an explicit call of its repaint() method is required to make the change visible.

Object → AbstractBorder → javax.swing.border.CompoundBorder

```
public CompoundBorder( Border innerBorder, Border outerBorder)
```

Constructs a CompoundBorder from the two Borders supplied.

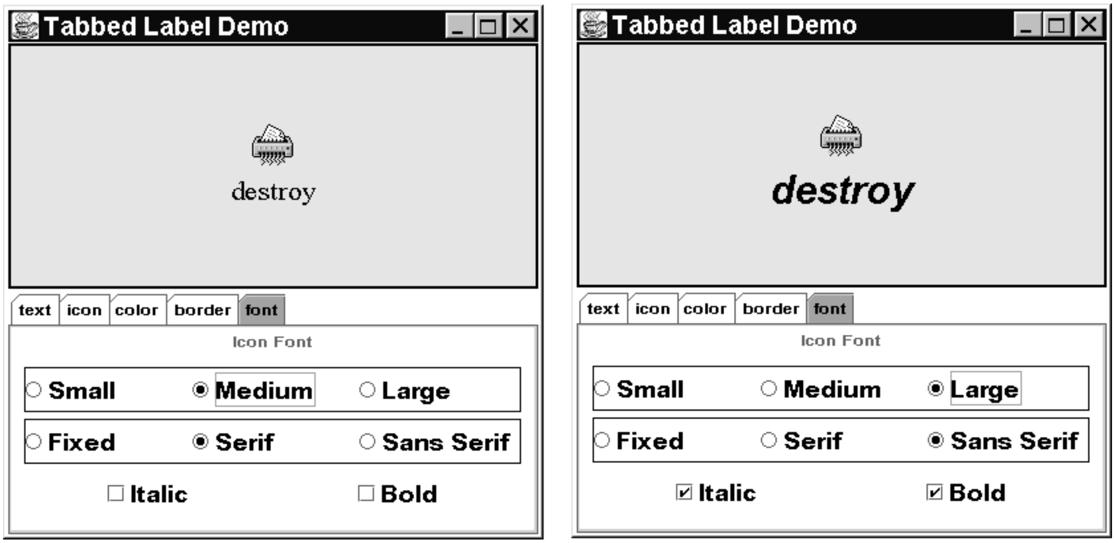
```
public Border getInsideBorder()
public Border getOutsideBorder()
```

Enquiry method for each of the contained borders.

3.7 The font resource: introducing JCheckBoxes

The next extension to *TabbedLabelDemo* adds a fifth tab labeled *font*, associated with which is an instance of the *FontPanel* class, whose appearance is illustrated in Figure 3.13.

The first row of radio buttons is a single group controlling the size of the font: the second row is a separate group controlling the font family. The third row contains two instances of the JCheckBox class controlling the font style: bold and/or italic. As the controls on the last row operate independently of each other they are not grouped by a border, reinforcing to the user the difference between them and the other controls. The images in Figure 3.13 illustrate the effects of the controls, with the right-hand image showing the "*destroy*" text label in a large, bold, italic sans serif font. The artifact makes use of the *FontStore* class from the *fontviewer* package, as described in the previous chapter, to obtain Fonts.



Tabbed Label Demo

destroy

text | icon | color | border | font

Icon Font

○ Small ● Medium ○ Large

○ Fixed ● Serif ○ Sans Serif

□ Italic □ Bold

Tabbed Label Demo

destroy

text | icon | color | border | font

Icon Font

○ Small ○ Medium ● Large

○ Fixed ○ Serif ● Sans Serif

☑ Italic ☑ Bold

Figure 3.13 *TabbedLabelDemo* showing the *FontPanel*.

The JCheckBox instances used in this panel consist of a text label and a state indicator. Unlike JRadioButtons, JCheckBoxes can be set, or unset, independently of each other, and are toggled between being set and unset by clicking upon them with the mouse. The state indicator contains a tick when the box is set, and is clear when they are unset. As with JRadioButtons, JCheckBoxes will generate an ItemEvent when their state is changed and require an ItemListener to be registered with them, to which the event will be dispatched. The first stage of the FontPanel constructor is as follows.

```
class FontPanel extends JPanel implements ItemListener {

private JLabel    controllingThis = null;
private FontStore fontStore       = null;

    protected FontPanel( JLabel toControl) {

    JPanel controlPanel = new JPanel(
                              new GridLayout( 3, 1, 5, 5));
    JPanel sizePanel    = new JPanel(
                              new GridLayout( 1, 3, 5, 5));
    JPanel familyPanel  = new JPanel(
                              new GridLayout( 1, 3, 5, 5));
    JPanel stylePanel   = new JPanel(
                              new GridLayout( 1, 2, 5, 5));

    ButtonGroup sizeGroup  = new ButtonGroup();
    ButtonGroup styleGroup = new ButtonGroup();

    JRadioButton smallRadio  = new JRadioButton( "Small",true);
    JRadioButton mediumRadio = new JRadioButton( "Medium");
    JRadioButton largeRadio  = new JRadioButton( "Large");

```

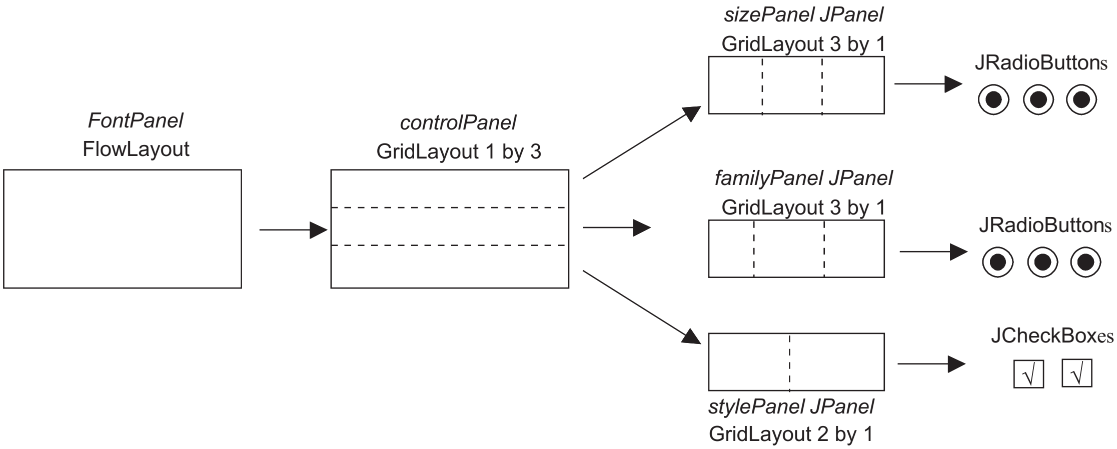


Figure 3.14 *FontPanel* layout diagram.

```
0044      JRadioButton fixedRadio = new JRadioButton( "Fixed");
0045      JRadioButton serifRadio = new JRadioButton( "Serif", true);
0046      JRadioButton sansRadio  = new JRadioButton( "Sans Serif");
0047
0048      JCheckBox italicToggle = new JCheckBox( "Italic", false);
0049      JCheckBox boldToggle   = new JCheckBox( "Bold",   false);
```

The class has two encapsulated attributes. As with the other panels in this chapter it encapsulates the identity of the JLabel whose attributes it is controlling in *controllingThis*. It also has a *FontStore* instance, called *fontStore*, to maintain knowledge of the font that is currently selected and to obtain a new Font when the user changes one of the settings.

The constructor commences, on lines 0028 to 0035 by declaring, constructing and configuring the four intermediate panels that are needed to implement the panel's appearance. The geometry relationships used in the *FontPanel* are illustrated in the layout diagram in Figure 3.14. The *FontPanel* retains its default FlowLayout and has a single *controlPanel* added to it. The *controlPanel* has a three row by one column GridLayout with each of the rows being used for one of the set of control buttons. The upper cell of the *controlPanel* contains the *sizePanel*, whose one row by three column GridLayout is used to position the three radio buttons that control the size of the font. Likewise, the *controlPanel*'s second row is used for the *familyPanel* which also has a one row by three column GridLayout whose three radio buttons control the font family. The lowest row in the *controlPanel* contains the *stylePanel*, which has a one row by two column GridLayout and contains the two check boxes that are used to control the font style.

This part of the constructor continues, on lines 0037 and 0038, by constructing the two ButtonGroups that will be required to control the two sets of three JRadioButtons constructed on lines 0040 to 0046. Finally, on lines 0048 and 0049, the two JCheckBox instances, *italicToggle* and *boldToggle*, are declared. The arguments to the JCheckBox constructors indicate the text to label them with and that when first presented to the user they should not be checked.

JComponent → AbstractButton → JToggleButton → javax.swing.JCheckBox

```
public JCheckBox()
public JCheckBox ( Icon icon)
public JCheckBox ( Icon  icon, boolean selected)
public JCheckBox ( String text)
public JCheckBox ( String text, boolean selected)
public JCheckBox ( String text, Icon icon)
public JCheckBox ( String text, Icon icon, boolean selected)
```

Constructs a JCheckBox instance with an *icon* and/or *text* and possibly *selected*.

The constructor continues as follows.

```
0050         this.setBorder( new TitledBorder(
0051                                     new EmptyBorder( 5,5,5,5),
0052                                     "Icon Font",
0054                                     TitledBorder.CENTER,
0055                                     TitledBorder.TOP));
0056
0057         fontStore = new FontStore();
0058         fontStore.setSelectedFamily( "Serif");
0059         fontStore.setSelectedSize(   "14");
0060         fontStore.setSelectedStyle(  "Plain");
0061         controllingThis = toControl;
0062         controllingThis.setFont( fontStore.getSelectedFont());
```

Lines 0050 to 0055 install the TitledBorder into **this** FontPanel. The *fontStore* instance is constructed and configured, on lines 0058 to 0060, so as to be congruent with the default settings on the panel. After the argument *toControl* has been stored in *controllingThis*, the *fontStore getSelectedFont()* method is used to supply a Font argument to the *controllingThis* setFont() method. The effect of this fragment is to make sure that, when the panel first becomes visible to the user, the font used by the JLabel instance, the settings on the *FontPanel* and the state of the *fontStore* are all congruent.

The constructor continues by registering the *FontPanel* being constructed as the itemListener resource of all eight user controls, setting their horizontalAlignment attributes and establishing the ButtonGroups, as follows in outline only.

```
0064         smallRadio.addItemListener(  this);
--         // Omitted five radio buttons addItemListener.
0072         italicToggle.addItemListener( this);
0073         boldToggle.addItemListener(   this);
0074
0075         smallRadio.setHorizontalAlignment( SwingConstants.LEFT);
--         // Omitted five radio buttons setHorizontalAlignment.
0083         italicToggle.setHorizontalAlignment(
0084                                         SwingConstants.CENTER);
0085         boldToggle.setHorizontalAlignment(
0086                                         SwingConstants.CENTER);
0085
0086         sizeGroup.add( smallRadio);
--         // Omitted other two radio buttons sizeGroup.add().
0090         familyGroup.add( fixedRadio);
```

```
--         // Omitted other two radio buttons familyGroup.add().
```

The remaining part of the constructor implements the geometry relationships shown in the layout diagram in Figure 3.14, as follows.

```
        sizePanel.setBorder( new LineBorder( Color.black, 1));
        sizePanel.add( smallRadio);
        sizePanel.add( mediumRadio);
        sizePanel.add( largeRadio);

        familyPanel.setBorder( new LineBorder( Color.black, 1));
        familyPanel.add( fixedRadio);
        familyPanel.add( serifRadio);
        familyPanel.add( sansRadio);

        stylePanel.add( italicToggle);
        stylePanel.add( boldToggle);

        controlPanel.add( sizePanel);
        controlPanel.add( familyPanel);
        controlPanel.add( stylePanel);

        this.add( controlPanel);
    } // End FontPanel constructor.
```

The effect of this constructor is to create an instance of the *FontPanel* which will appear to the user as shown in Figure 3.13, assuming that an instance has been constructed and added to the *tabbedPane* in the *TabbedLabelDemo init()* action, and the user selects its *font* tab. Whenever the user selects one of the radio buttons or selects, or deselects, one of the check boxes, an ItemEvent will be constructed and passed as an argument to the FontPanel's *itemStateChanged()* method, which commences as follows.

```
     public void itemStateChanged( ItemEvent event) {

     String theItem = ((JToggleButton)
                             event.getItem()).getText();

        if ( theItem.equals( "Fixed")) {
           fontStore.setSelectedFamily( "Monospaced");
        } else if ( theItem.equals( "Serif")) {
           fontStore.setSelectedFamily( "Serif");
        } else if ( theItem.equals( "Sans Serif")) {
           fontStore.setSelectedFamily( "SansSerif");

        } else if ( theItem.equals( "Small"))  {
           fontStore.setSelectedSize( "14");
        } else if ( theItem.equals( "Medium")) {
           fontStore.setSelectedSize( "20");
        } else if ( theItem.equals( "Large"))  {
           fontStore.setSelectedSize( "30");
        } else {
```

This method may be called as a consequence of the user pressing a JRadioButton or pressing a JCheckBox button. Both of these classes are subclasses of the JToggleButton class, which supplies the getText() method, as previously used. Accordingly, on lines 0117 and 0118, the String label of whichever button was pressed is obtained after casting the component item of the *event* into a JToggleButton.

This first part of the *itemStateChanged()* method is concerned with responding to the events dispatched when the user selects one of the JRadioButtons. Accordingly, it is implemented as a six way sequential selection with one branch for each possible event source, identified by the value of the *theItem* local variable. Within each branch either the *fontStore setSelectedFamily()* or *setSelectedSize()* method is called, with an appropriate String argument.

The seventh branch of the sequential **if** structure is concerned with responding to presses of the JCheckBoxes. It, and the remainder of the *FontPanel* class, is implemented as follows.

```
0133             if ( event.getStateChange()
0134                         == ItemEvent.SELECTED) {
0135                if ( theItem.equals( "Bold")) {
0136                   bold   = true;
0137                } else {
0138                   italic = true;
0139                } // end if.
0140             } else {
0141                if ( theItem.equals( "Bold")) {
0142                   bold   = false;
0143                } else {
0144                   italic = false;
0145                } // end if.
0146             } // End if.
0147
0148             if ( !bold && !italic ) {
0149                fontStore.setSelectedStyle( "Plain");
0150             } else if ( bold && !italic )  {
0151                fontStore.setSelectedStyle( "Bold");
0152             } else if ( !bold && italic )  {
0153                fontStore.setSelectedStyle( "Italic");
0154             } else if ( bold && italic )   {
0155                fontStore.setSelectedStyle( "Bold/Italic");
0156             } // End if.
0157         } // End if.
0158
0159         controllingThis.setFont(
0160                                fontStore.getSelectedFont());
0161         controllingThis.repaint();
0162      } // End itemStateChanged.
0163
0164  private boolean bold   = false;
0165  private boolean italic = false;
0166
0167  } // End class FontPanel.
```

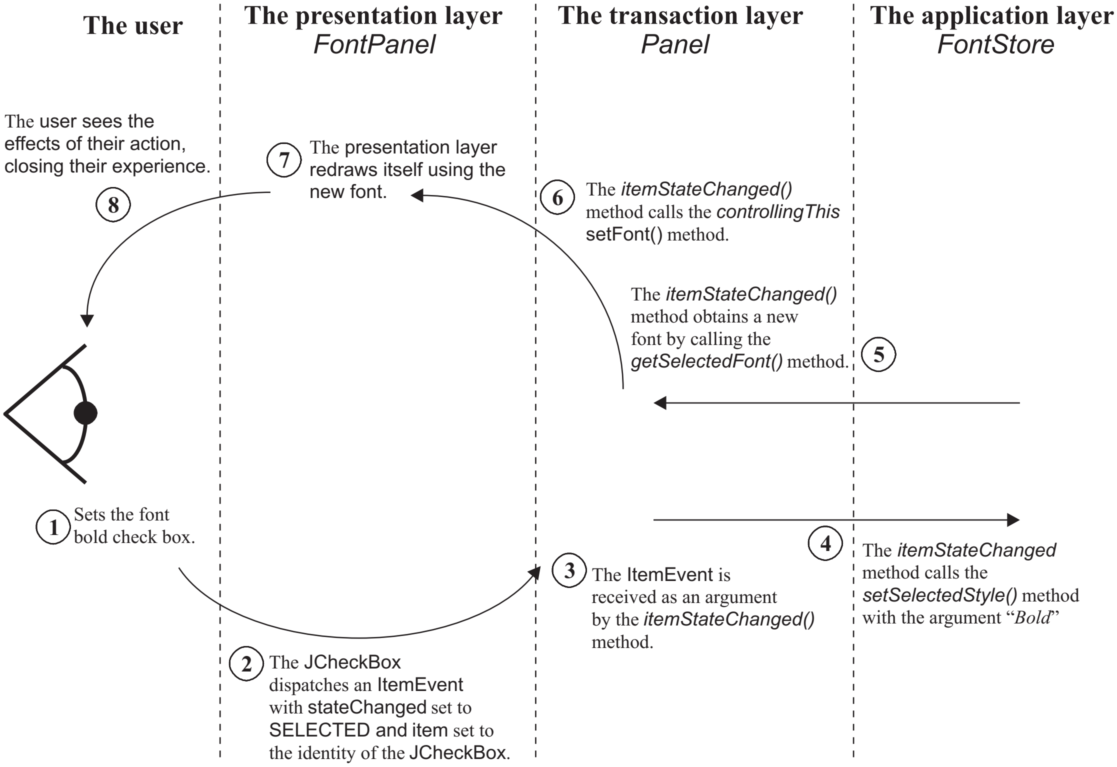
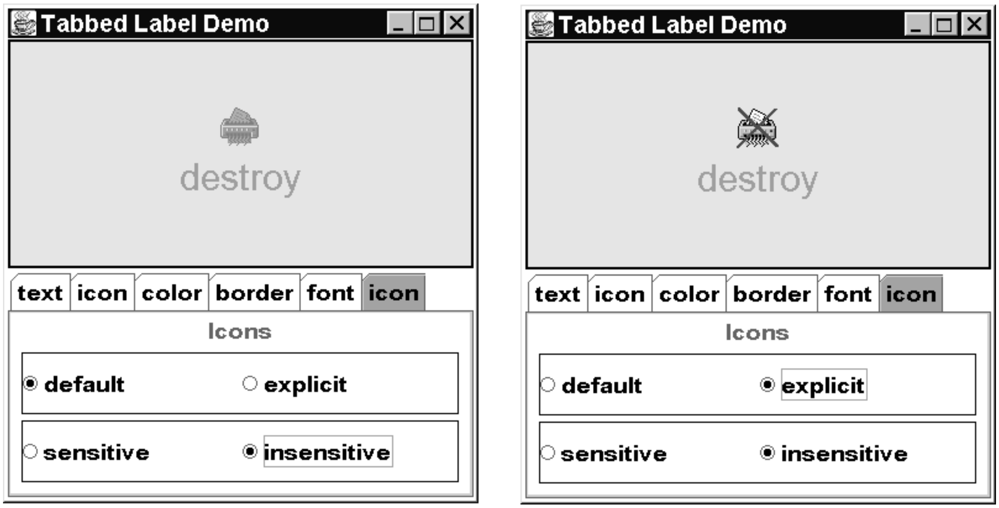

The user
The presentation layer
FontPanel
The transaction layer
Panel
The application layer
FontStore
The user sees the effects of their action, closing their experience.
8
7
The presentation layer redraws itself using the new font.
6
The itemStateChanged() method calls the controllingThis setFont() method.
The itemStateChanged() method obtains a new font by calling the getSelectedFont() method.
5
1
Sets the font bold check box.
4
The itemStateChanged method calls the setSelectedStyle() method with the argument "Bold"
3
The ItemEvent is received as an argument by the itemStateChanged() method.
2
The JCheckBox dispatches an ItemEvent with stateChanged set to SELECTED and item set to the identity of the JCheckBox.

Figure 3.15 Object interactions diagram when Bold is checked.

This part of the class has a requirement to maintain knowledge of the state of the two JCheckBoxes on the user interface. It accomplishes this by maintaining two further **boolean** instance attributes, called *bold* and *italic*, declared on lines 0164 and 0165. These two attributes will be manipulated directly, i.e. not by the use of **private** state setting and inquiry methods. This implementation decision was taken as the extent of the code which has an interest in them is strictly limited to this part of the implementation and, as they are both primitive, the additional complexity of maintaining the methods was not felt worthwhile.

When the user selects a JCheckBox the stateChange attribute of the *event* generated will have the value ItemEvent.SELECTED. Likewise, it will have the value ItemEvent.DE_SELECTED when the user deselects a JCheckBox. The **if** structure on lines 0133 to 0146 uses the stateChanged attribute of the *event* and the label of the check box, stored in *theItem*, to maintain the values of *bold* and *italic* in step with the state of the two JCheckBoxes on the interface. Lines 0148 to 0156 then call the *fontStore setSelectedStyle()* attribute with a String argument appropriate to the current state of the two **boolean** attributes.

The end of the seven-way selection structure is on line 0157, and as one of the attributes of the Font must have been changed by the user, the JLabel's appearance needs to be changed to reflect this. Lines 0159 and 0160 call the *controllingThis* setFont() method, passing as an argument the Font returned by the *fontStore getSelectedFont()* method. The



● default ○ explicit
○ sensitive ● insensitive

○ default ● explicit
○ sensitive ● insensitive

Figure 3.16 *TabbedLabelDemo* showing the *IconPanel*.

final line of the method, line 0161, ensures that this change is made visible to the user by calling the *controllingThis* repaint() method.

Figure 3.15 contains an object interaction diagram which describes the sequence of methods which are called, starting when the user checks the "*Bold*" button and finishing when the text label in the JLabel window is emboldened, closing the user's experience.

3.8 Alternative icons and component sensitivity

The next extension to *TabbedLabelDemo* adds a sixth tab labeled *icon*, associated with which is an instance of the *IconPanel* class, whose appearance is illustrated in Figure 3.16.

The *IconPanel* allows the user to indicate the component sensitivity, controlled by the **boolean** enabled attribute. Components which are disabled are unable to interact with the user and are displayed in a different manner. In this example, if it were real, it might indicate that the shredding operation is, for some reason, unavailable. In the left-hand image of Figure 3.16 the default insensitive appearance of the JLabel is illustrated. In order to indicate that the resource is not available the iconic image has been converted to gray scale, has had its brightness reduced and the label associated with it is rendered in a gray outlined font. In the right-hand illustration an explicit alternative icon, again intended to indicate that the resource is not available, has been installed; the insensitive text label is rendered as before.

The two rows of radio buttons operate as two separate groups. The user can indicate, using the lower row, that the JLabel instance is to be shown in its sensitive or insensitive state. The upper row can be used, independently, to indicate whether the default or alternative icon is to be shown when it is insensitive.

The overall construction of the *ImageIcon* class does not differ greatly from other classes, such as the *TextLocationPanel* class, which contain two groups of radio buttons. Outside the scope of any of the classes' methods an ImageIcon, called *insensitiveIcon*, is constructed, as follows, to contain the alternative shredder image, shown in the right-hand image of Figure 3.16. The details of this icon construction technique were described in section 3.2 of this chapter.

```
0020  private final static ImageIcon disabledIcon =
0021                             new ImageIcon( "noShredder.gif");
```

Within the *init()* method, after the identity of the JLabel instance has been stored in the *controllingThis* instance attribute, it is configured so as to conform with the default selections of the radio buttons. This configuration is such that the JLabel is shown in an enabled state with the default disabled icon available; configuration is accomplished as follows.

```
0040         controllingThis = toControl;
0041         controllingThis.setDisabledIcon( null);
0042         controllingThis.setEnabled( true);
```

JComponent → javax.swing.JLabel

```
public void setDisabledIcon( Icon setTo)
public Icon getDisabledIcon()
```

Installs, or obtains the identity of, the icon which will de displayed when the label is disabled. If this resource is **null**, then a grayed-out version of the default icon, if any, will be shown.

The JLabel setDisabledIcon() method allows an alternative image to be specified, which will be shown when the component is disabled. If the argument to this action is **null**, as above, the JLabel will display a grayed-out version of the installed enabled icon when it is disabled. The setEnabled() action is introduced into the hierarchy by the AWT Component class, and so is inherited by all AWT and JFC components. If the argument to this action is **true** then the component is enabled and the user will be able to interact with it as normal. If the argument is **false** the component is disabled, the user will be unable to interact with it and it will be shown in a different manner to indicate this.

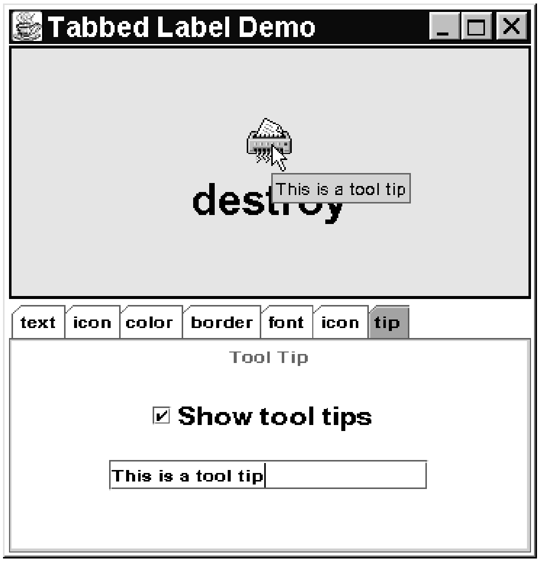
Object → java.awt.Component

```
public void setEnabled( boolean yesOrNo)
public boolean isEnabled()
```

Sets, or obtains, the sensitivity of a component. A disabled component cannot be interacted with and is displayed in an alternative manner, usually (and conventionally) grayed out to indicate this.

The *IconPanel itemStateChanged*() method, shown below, commences, on lines 0070 to 0071, by retrieving the text label of the radio button which generated the ItemEvent, as before.

```
0068      public void itemStateChanged( ItemEvent event) {
0069
0070      String theState = ((JRadioButton)
0071                                 event.getItem()).getText();
0072
0073         if ( theState.equals( "default")) {
0074            controllingThis.setDisabledIcon( null);
0075         } else if ( theState.equals( "explicit")) {
0076            controllingThis.setDisabledIcon( disabledIcon);
0077
```



This is a tool tip

☑ Show tool tips

This is a tool tip

Figure 3.17 *TabbedLabelDemo* showing the *TipPanel*.

```
0078          } else if ( theState.equals( "sensitive")) {
0079             controllingThis.setEnabled( true);
0080          } else if ( theState.equals( "insensitive")) {
0081             controllingThis.setEnabled( false);
0082          } // End if.
0083          controllingThis.repaint();
0084       } // End itemStateChanged.
```

The first two branches of the **if/else if** structure are connected to the upper row of radio buttons, labeled *default* and *explicit*. The *default* branch calls the *controllingThis* setDisabledIcon() method with the argument **null** and the *explicit* branch uses the same method to install the *disabledIcon*. The two remaining branches are connected to the sensitive and insensitive buttons and both call the *controllingThis* setEnabled() method with the arguments **true** and **false** respectively. The changes to the JLabel instance do not result in it redrawing itself to show the effects of these attribute changes, so on line 0082 its repaint() method is called.

The consequence of these actions is that if the *sensitive* or *insensitive* buttons are pressed the JLabel will always redisplay itself to show the changed icon. If the *default* or *explicit* buttons are pressed the image will change only if the *insensitive* button is selected, otherwise the effect of the change will only become apparent when it becomes insensitive.

3.9 ToolTip resources: introducing the JTextField class

This extension to *TabbedLabelDemo* adds a seventh tab labeled *tip*, associated with which is an instance of the *TipPanel* class, whose appearance is illustrated in Figure 3.17.

A *ToolTip* is a small window containing a short message which is shown to the user when the mouse pointer rests for a second or two over a component. In the image in the tool tip a box containing the text “*This is a tool tip*” can be seen. If this artifact were executed with a Macintosh *look and feel* the tool tip would take the form of a 'bubble help'

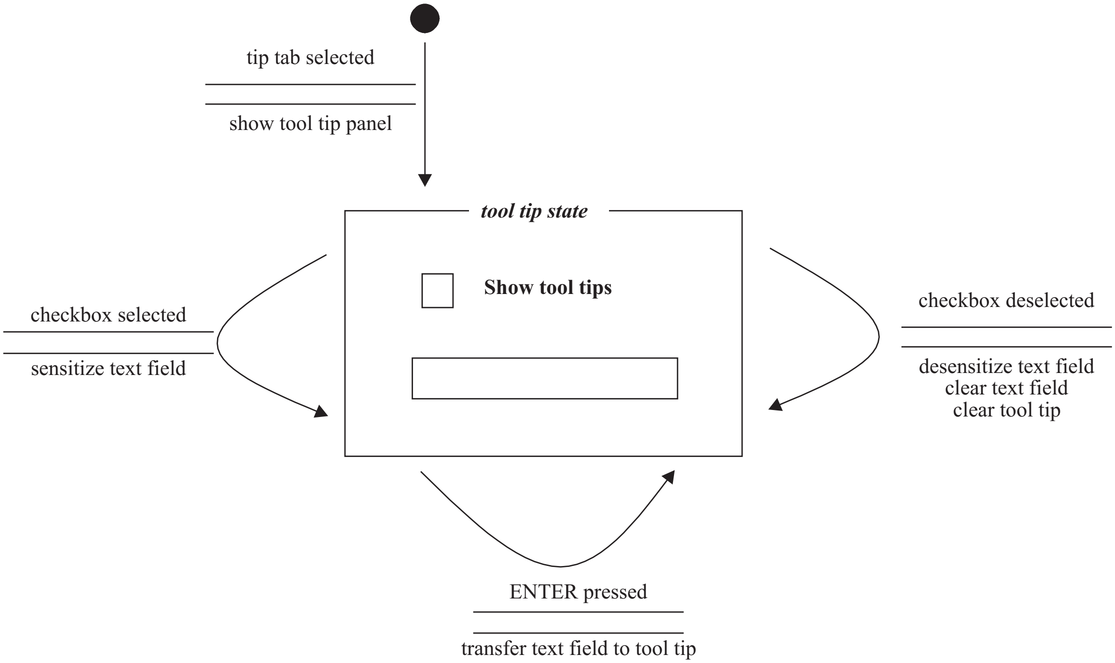


Figure 3.18 *TipPanel* state transition diagram.

indicator. The precise meaning of look and feel and the support provided for different look and feels will be presented later in this chapter.

The check box at the top of the *TipPanel* determines whether the tip is to be shown or not. The JTextField at the bottom of the panel allows the user to indicate what message is to be shown as the tip. A JTextField component supplies a single-line text input area into which the user can type a short message; it generates an ActionEvent, whose actionCommand attribute contains the text from the text field, when the user presses the ENTER key. There is no sensible reason why a user should be empowered to change a tool tip in this way; this example is being introduced to indicate that the toolTip resource of a swing component can be changed should it be required.

For example, if the shredding operation is available the tip could explain that the resource would be permanently destroyed. However, if the shredding operation were not available, then, in addition to using a disabled icon, the tool tip could be changed to explain why not.

A tool tip can only be entered into the JTextField if tool tips are enabled and is only installed into the JLabel instance when the user presses the ENTER key to complete the text. This behavior is specified in the state transition diagram given in Figure 3.18.

To obtain the visual appearance shown in Figure 3.17 the *TipPanel* contains a *controlPanel* which has a two row by one column GridLayout. In the upper cell a JPanel, with default FlowLayout, contains the JCheckBox and in the lower cell another JPanel, also with default FlowLayout, contains the JTextField. These geometry relationships are installed in the *init()* action as follows.

```
0019  class TipPanel extends    JPanel
0020                 implements ItemListener, ActionListener {
```

```
0021
0022  private JLabel     controllingThis = null;
0023  private JTextField tipField        = null;
0024
0025      protected TipPanel( JLabel toControl) {
0026
0027      JPanel    controlPanel = new JPanel(
0028                                       new GridLayout( 2, 1));
0029      JPanel    upperPanel = new JPanel();
0030      JPanel    lowerPanel = new JPanel();
0031      JCheckBox tipBox     = new JCheckBox( "Show tool tips");
0032
0033         tipField = new JTextField(  20);
0034         tipField.addActionListener( this);
0035
0036         tipBox.addItemListener(     this);
0037
0038         this.setBorder( new TitledBorder(
0039                                    new EmptyBorder( 5,5,5,5),
0040                                    "Tool Tip",
0041                                    TitledBorder.CENTER,
0042                                    TitledBorder.TOP));
0043
0044         controllingThis = toControl;
0045
0046         upperPanel.add( tipBox);
0047         lowerPanel.add( tipField);
0048
0049         controlPanel.add( upperPanel);
0050         controlPanel.add( lowerPanel);
0051
0052         this.add( controlPanel);
0053     } // End TipPanel constructor.
```

The check box will generate ItemEvents when it is used and the text field will generate ActionEvents; consequently this class must implement both the ItemListener and ActionListener interfaces in order to be able to listen to all possible events. Following the class declaration, and the declaration of the *controllingThis* JLabel instance attribute on line 0022, an instance of the JTextField is declared on line 0023. The constructor commences by constructing the three intermediate JPanels needed for the required layout and, on line 0031, an instance of the JCheckBox class, called *tipBox*.

The *toolTip* TextField instance is constructed, on line 0033, using a constructor indicating that it should be capable of displaying about 20 characters. It has the *TipPanel* currently being constructed registered as its actionListener resource on line 0034. The *TipPanel* is also registered as the *tipBoxes*' itemListener resource, on line 0036. Following the operations on lines 0038 to 0044 that install the *TipPanel*'s title and record the identity of the JLabel *toControl*, the interface is assembled on lines 0046 to 0052.

JComponent → JTextComponent → javax.swing.JTextField

```
public JTextField()
public JTextField( int columns)
public JTextField( String text)
public JTextField( String text, int columns)
```

Constructors allowing the initial *text* String to be specified and/or the number of *columns*.

```
public void   setColumns( int columns)
public int    getColumns()
public void   setText( String text)
public String getText ()
```

State setting and inquiry methods for the two attributes used in the constructors. The setText() and getText() methods are inherited from JTextComponent and there are many other attributes and methods.

The implementation of the two listener methods is as follows.

```
0053      public void itemStateChanged( ItemEvent event) {
0054
0055         if ( event.getStateChange() == ItemEvent.SELECTED) {
0056            tipField.setEditable( true);
0057         } else {
0058            controllingThis.setToolTipText( null);
0059            tipField.setText( "");
0060            tipField.setEditable( false);
0061         } // End if.
0062      } // End itemStateChanged.
0063
0064
0065      public void actionPerformed( ActionEvent event) {
0066         controllingThis.setToolTipText(
0067                                    event.getActionCommand());
0068      }// End actionPerformed.
0069  } // End class TipPanel.
```

The *itemStateChanged()* method is implemented as a two-way selection, each branch associated with the check box being selected or deselected as determined by the stateChange attribute of the ItemEvent received. There is a single action associated with selecting the check box, as indicated on its STD. Line 0055 sensitizes the *tipField* by calling its setEditable() method with the argument **true**. If the check box is deselected the sequence of three actions on lines 0058 to 0060 is executed. On line 0058 any existing tool tip in the JLabel instance *controllingThis* is cleared by calling its setToolTipText() method with a **null** argument. Line 0059 clears any text from the *tipField* by calling its setText() method with an empty String ("") as its argument; and line 0060 desensitizes it by calling setEditable() with the argument **false**.

An ActionEvent can only be dispatched to the *actionPerformed()* method if the *tipField* is sensitive. It is implemented on lines 0066 and 0067, by calling the *controllingThis* setToolTipText() method to install the new tool tip text. The actionCommand attribute of an ActionEvent generated from a TextField will contain the contents of the TextField. Accordingly, the getActionCommand() inquiry method of the *event* argument is used to supply the argument to setToolTipText(). The effect is to install the text from the text field into the JLabel as its tool tip resource.

javax.swing.JComponent

```
public void   setToolTipText( String toThis)
public String getToolTipText()
```

Sets, or obtains, the text to be shown if the mouse pointer rests over the component. Nothing will be shown if this attribute is set to **null**.

3.10 Custom colors – introducing the JSlider class

The next extension to the *TabbedLabelDemo* artifact adds an instance of the *CustomColorPanel*, associated with a tab labeled *custom*, which allows the user to interactively indicate the precise background color for the JLabel[1]. This panel complements the functionality of the existing *ColorPanel* component, which only allows the colors which are pre-supplied by the AWT Color class to be used. The appearance of this panel is illustrated in Figure 3.19.

The interface offers three instances of the JSlider class, labeled *red*, *green* and *blue*, controlling the three primary attributes of a color. The sliders' left extents indicate a

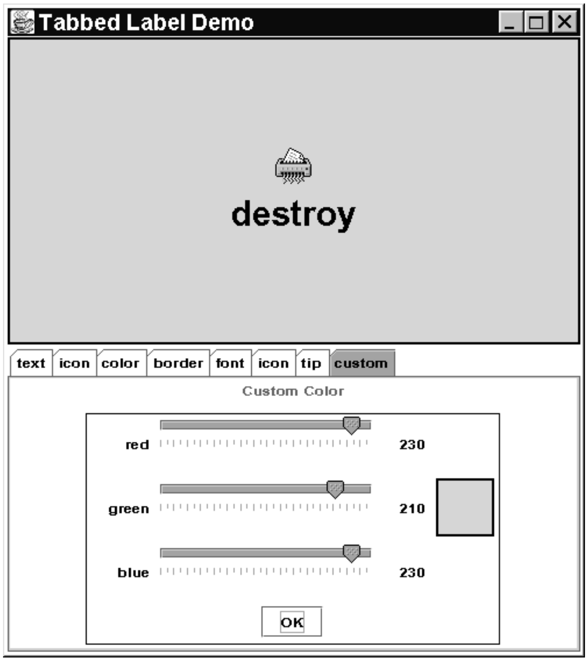


Figure 3.19 *TabbedLabelDemo* showing the *CustomColorPanel*.

[1] The JFC supplies a class called JColorChooser that also supplies this functionality. However, the intention here is to illustrate the use of JSliders and complex layout management. It might also be the case that instances of the JColorChooser might not be totally suitable for a particular requirement, in which case this design and implementation might be more amenable to changes.

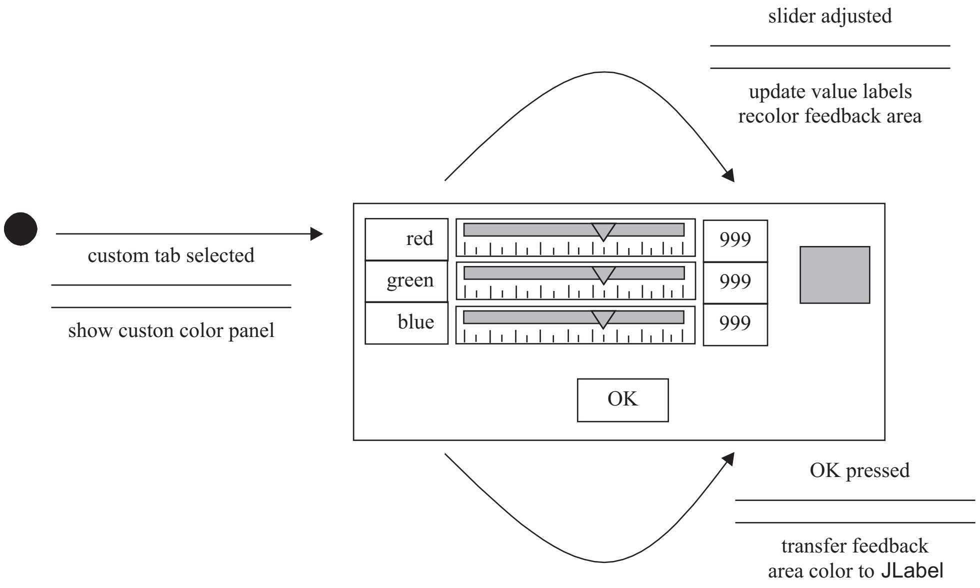


Figure 3.20 *CustomColorPanel* state transition diagram.

minimal contribution of that primary color and the right extents a maximal contribution. The relative contributions are indicated by the numeric labels to the right of the slider and can range between 0 (0x00) and 255 (0xFF). To the right of the three numeric labels there is a feedback area where the appearance of the color indicated by the sliders is shown to the user. The button labeled *OK* at the bottom of the interface will transfer the color from the feedback area to the background resource of the JLabel instance.

The user interacts with this interface by dragging, or otherwise interacting with the sliders, until the color they require is showing in the feedback area and then by pressing the *OK* button to set the background attribute of the JLabel. This behavior is illustrated in the STD diagram in Figure 3.20.

This panel is the most complex interface that will be presented in this chapter and consequently has complex geometry relationships, as illustrated in Figure 3.21. The *CustomColorPanel* itself, shown in the center of the diagram, has a BorderLayout with three of its possible five areas occupied. In its *South* area a JPanel instance called *okPanel*, contains the *okButton* which is an instance of the JButton class.

The *CustomColorPanel*'s *East* location contains a Box instance, called *feedbackPanel*. A Box is a lightweight container that has a BoxLayout manager installed. BoxLayout management policy is similar to FlowLayout management policy but affords more control over the relative positioning of its children. In this example it has a single child, a JPanel instance called *feedbackArea*, with glue above and below it. In this arrangement, the layout manager will attempt to afford the *feedbackArea* its preferred size and any remaining vertical space will be given equally to both glue areas. Hence the *feedbackArea* will always be vertically centered within the available space.

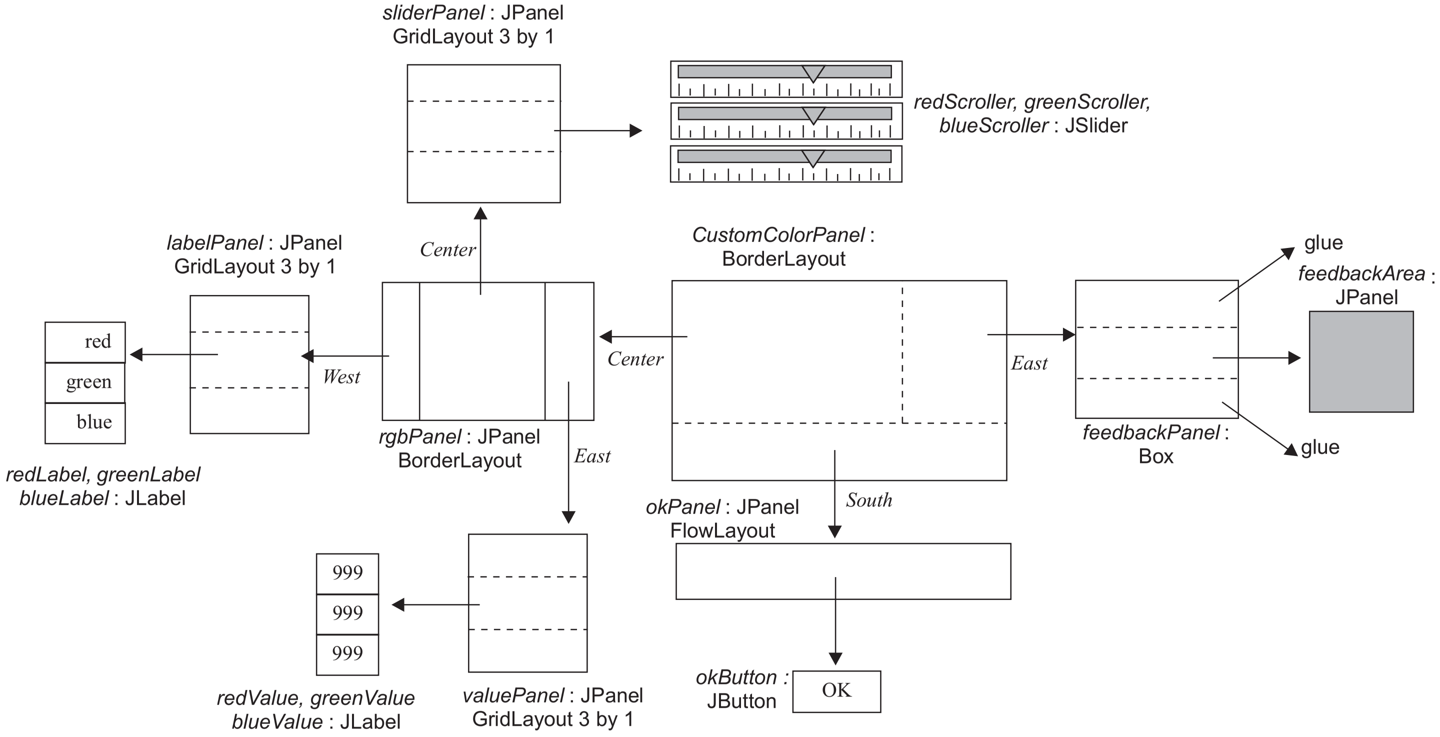


Figure 3.21 *CustomColorPanel* layout diagram.

Object → Component → Container → javax.swing.Box

```
public Box( int orientation)
```

Constructs a Box with a horizontal, BoxLayout.X_Axis, or vertical, BoxLayout.Y_Axis, layout.

```
public static Component createRigidArea( Dimension size)
public static Component createHorizontalStrut( int width)
public static Component createVerticalStrut(   int height)
public static Component createHorizontalGlue(  int width)
public static Component createVerticalGlue(    int height)
public static Component createGlue()
```

Convenience methods that create lighweight components to populate a BoxLayout manager. A rigidArea is always the size specified; horizontalStruts and verticalStruts maintain the width or height specified. A glue component will be afforded an equal share of any remaining space.

The *Center* location of the *CustomColorPanel* contains a JPanel instance called *rgbPanel* that also has a BorderLayout policy. The *West* location of the *rgbPanel* contains a JPanel, called *labelPanel*, with a three row by one column GridLayout policy containing the three JLabel instances which provide names for the sliders. In the *rgbPanel*'s *East* location another JPanel, called *valuePanel*, with a three row by one column GridLayout policy, contains the three JLabel instances which indicate the current numeric value of each of the sliders. In the *Center* location of the *rgbPanel* a further instance of the JPanel, called *sliderPanel*, with a three row by one column GridLayout policy, contains the three JSlider instances. The effect of these geometry relationships is to lay out the various components as illustrated in Figure 3.19.

The *CustomColorPanel*'s constructor is responsible for constructing and laying out the components and is necessarily similarly complex; the class declaration commences as follows.

```
class CustomColorPanel extends    JPanel
                       implements ChangeListener,
                                  ActionListener {

private JLabel  controllingThis = null;

private JSlider redSlider       = null;
private JSlider greenSlider     = null;
private JSlider blueSlider      = null;

private JLabel  redValue        = null;
private JLabel  greenValue      = null;
private JLabel  blueValue       = null;

private JPanel  feedbackArea     = null;
```

Instances of the *CustomColorPanel* class have to listen to the ChangeEvents generated by the JSliders and also to the ActionEvents generated by the JButton. Accordingly this class has to implement both the ChangeListener and ActionListener interfaces. The first part of the class declares all of the components whose identities are required by the listener methods as **private** instance attributes of the class. The first part of the constructor is as follows.

```
0034      protected CustomColorPanel( JLabel toControl) {
0035
0037      JButton okButton     = null;
0036      JPanel  controlPanel = new JPanel( new BorderLayout());
0038
0039      JPanel labelPanel  = new JPanel(
0040                                new GridLayout( 3, 1, 0, 5));
0041      JPanel sliderPanel = new JPanel(
0042                                new GridLayout( 3, 1, 0, 5));
0043      JPanel valuePanel  = new JPanel(
0044                                new GridLayout( 3, 1, 0, 5));
0045      JPanel rgbPanel    = new JPanel(
0046                                new BorderLayout());
0047
0048      JPanel okPanel       = new JPanel();
0049
0050      Box    feedbackPanel = new Box( BoxLayout.X_AXIS);
0051                                new GridLayout( 3, 1, 0, 5));
0052
0053      JLabel redLabel   = new JLabel( "red",
0054                                         SwingConstants.RIGHT);
0055      JLabel greenLabel = new JLabel( "green",
0056                                         SwingConstants.RIGHT);
0057      JLabel blueLabel  = new JLabel( "blue",
0058                                         SwingConstants.RIGHT);
```

The identity of the *okButton* is only required while the panel is being initialized and so is declared as a local attribute of the method on line 0035. Lines 0037 to 0052 then construct the eight intermediate JPanel instances indicated on the geometry diagram in Figure 3.21 and install their layout managers. This fragment concludes by constructing the three JLabel instances which will be installed into the *labelPanel*, before continuing as follows.

```
0061         redSlider = new JSlider( 0, 255, 100);
0062         redSlider.setMajorTickSpacing( 16);
0063         redSlider.setMinorTickSpacing( 8);
0064         redSlider.setPaintTicks( true);
0065         redSlider.addChangeListener( this);
0066
0067         greenSlider = new JSlider( 0, 255, 100);
0068         greenSlider.setMajorTickSpacing( 16);
0069         greenSlider.setMinorTickSpacing( 8);
0070         greenSlider.setPaintTicks( true);
0071         greenSlider.addChangeListener( this);
0072
0073         blueSlider = new JSlider( 0, 255, 100);
0074         blueSlider.setMajorTickSpacing( 16);
0075         blueSlider.setMinorTickSpacing( 8);
0076         blueSlider.setPaintTicks( true);
0077         blueSlider.addChangeListener( this);
```

JComponent → javax.swing.JSlider

```
public JSlider()
public JSlider( int orientation)
public JSlider( int minimum, int maximum)
public JSlider( int minimum, int maximum, int value)
public JSlider( int orientation, int minimum,
                    int maximum, int value)
```

Constructs a JSlider instance with *orientation*, either JSlider.HORIZONTAL (default) or JSlider.VERTICAL, whose left-hand (top) location is set to indicate *minimum* (default 0) and right-hand (lower) location to indicate *maximum* (default 100). The slider is positioned at *value* relative to these two values (default 50).

```
public void setOrientation( int setTo)
public int  getOrientation()
public void setValue( int setTo)
public int  getValue()
public void setMinimum( int setTo)
public int  getMinimum()
public void setMaximum( int setTo)
public int  getMaximum()
public void setExtent( int setTo)
public int  getExtent ()
```

State setting and inquiry actions for the four attributes described above. The extent attribute determines the range of values covered by the slider's control.

```
public void    setPaintTrack( boolean yesOrNo)
public boolean getPaintTrack()
public void    setMajorTickSpacing( int setTo)
public int     getMajorTickSpacing ()
public void    setMinorTickSpacing( int setTo)
public int     getMinorTickSpacing ()
public void    setPaintTicks( boolean yesOrNo)
public boolean getPaintTicks()
public void    setSnapToTicks( boolean yesOrNo)
public boolean getSnapToTicks()
```

Methods to control the visual appearance and behavior of the slider. The paintTrack attribute determines if the track is drawn (default **true**). The majorTickSpacing and minorTickSpacing attributes determine the intervals between the long and short tick marks with paintTicks (default **false**) determining whether they are drawn. The snapToTicks attribute determines whether the value is constrained to values determined by the tick marks. It is also possible to supply text labels using the paintLabels attributes; for details see the JFC documentation.

This part of the method constructs the three JSlider instances, the three arguments to the constructors indicating their minimum value (0), maximum value (255) and the initial value (100). Major (longer) vertical tick marks are placed 16 units apart and minor (shorter) vertical tickmarks every 8 units, and the paintTicks attribute is set on in order that the tick marks are shown, as illustrated in Figure 3.19. The conclusion of the construction and configuration of each slider involves the registration of the instance of the *CustomColorPanel*, **this**, being initialized as the scroll bar's ChangeListener attribute.

```
0079         redValue   = new JLabel( "  100  ",
0080                                  SwingConstants.CENTER);
0081         greenValue = new JLabel( "  100  ",
0082                                  SwingConstants.CENTER);
0083         blueValue  = new JLabel( "  100  ",
0084                                  SwingConstants.CENTER);
0085
0086         okButton = new JButton( "OK");
0087         okButton.addActionListener( this);
0088         okPanel.add( okButton);
0089
0090         feedbackArea = new JPanel();
0091         feedbackArea.setPreferredSize( new Dimension( 50, 50));
0092         feedbackArea.setOpaque( true);
0093         feedbackArea.setBackground(
0094                              new Color( 100, 100, 100));
0095         feedbackArea.setBorder(
0096                               new LineBorder( Color.black, 2));
0097
0098         feedbackPanel.add( Box.createGlue());
0099         feedbackPanel.add( feedbackArea);
0100         feedbackPanel.add( Box.createGlue());
```

This part of the constructor commences by constructing the three JLabel instances which will be mounted upon the *valuePanel*. Each is constructed displaying the value 100, with additional spaces to the left and the right in order to ensure that the component will be wide enough to accommodate all possible values. It continues, on lines 0086 to 0089, by constructing the *okButton*, registering its *ActionListener* and adding it to the *okPanel*.

The fragment concludes, on lines 0091 to 0100, by constructing an instance of the JPanel class, called *feedbackArea*: establishing a size for it with its setPreferredSize(), setting it opaque, initializing its background color to conform to the initial values of the three sliders (100) and giving it a plain border. On line 0099 it is added to the *feedbackPanel* after some *glue* has been installed above it and before some *glue* is installed below it. A JPanel instance is being used for the *feedbackArea*, as all that is required is an area which can have its background color attribute specified and which will display itself using that color. The most suitable component class to use for this requirement happens to be the JPanel class, as it is one of the simplest of the JFC classes. The simplest JFC class, the JComponent class, cannot be used, as it is declared **abstract**. The constructor concludes as follows.

```
0101         this.setBorder( new TitledBorder(
0102                                 new EmptyBorder( 5, 5, 5, 5),
0103                                     "Custom Color",
0104                                     TitledBorder.CENTER,
0105                                     TitledBorder.TOP));
0106
0107        controllingThis = toControl;
0108
0109        labelPanel.add( redLabel);
0110        labelPanel.add( greenLabel);
```

```
0111          labelPanel.add( blueLabel);
0112
0113          sliderPanel.add( redSlider);
0114          sliderPanel.add( greenSlider);
0115          sliderPanel.add( blueSlider);
0116
0117          valuePanel.add( redValue);
0118          valuePanel.add( greenValue);
0119          valuePanel.add( blueValue);
0120
0121          rgbPanel.add( labelPanel,    BorderLayout.WEST);
0122          rgbPanel.add( sliderPanel, BorderLayout.CENTER);
0123          rgbPanel.add( valuePanel,    BorderLayout.EAST);
0124
0125          controlPanel.setBorder( new LineBorder( Color.black, 1));
0126          controlPanel.add( rgbPanel,      BorderLayout.CENTER);
0127          controlPanel.add( feedbackPanel, BorderLayout.EAST);
0128          controlPanel.add( okPanel,       BorderLayout.SOUTH);
0129
0130          this.add( controlPanel);
0131       } // End CustomColorPanel constructor.
```

On lines 0101 to 0105 a TitledBorder is installed into **this** *CustomColorPanel* currently being initialized and line 0106 stores the identity of the JLabel instance *toControl*. The remaining actions assemble the interface by adding each component to its parent container, using the relationships indicated in the geometry diagram in Figure 3.21.

javax.swing.event.ChangeListener interface

```
public void stateChanged( ChangeEvent event)
```

Method to be called by the event source when the listener needs to be notified that a change has occurured in a component.

Every time the user adjusts one of the sliders an ChangeEvent instance will be constructed and the stateChanged() method of its registered ChangeListener called, passing the event as an argument. Each slider has had the *CustomControlPanel* of which it is a component part registered as its listener, so the *stateChanged()* method, which follows, will be called each time.

```
0142       public void stateChanged( ChangeEvent event){
0143
0144       int red   = redSlider.getValue();
0145       int green = greenSlider.getValue();
0146       int blue  = blueSlider.getValue();
0147
0148          feedbackArea.setBackground(
0149                                  new Color( red, green, blue));
0150
0151          redValue.setText(   (new Integer( red)).toString());
0152          greenValue.setText( (new Integer( green)).toString());
0153          blueValue.setText(  (new Integer( blue)).toString());
0154       } // End stateChanged.
```

The method commences, on lines 0144 to 0146, by obtaining the value indicated by each slider, using their getValue() actions, and storing the values in three local **int** variables. Lines 0148 and 0149 then use these three values to construct an instance of the AWT Color class and use this as an argument to the *feedbackArea*'s setBackground() method. The consequence of this will be that the feedbackArea will change color to reflect the values of the three scroll bars. The final stage of this method converts the three **int** values to String representations and installs them into the appropriate JLabel instance using its setText() method. The consequence of this part of the method will be that the three numeric labels to the right of the sliders will change to reflect the changed value of one of them. It would have been possible to identify exactly which slider generated the event and change only one value, but this seems to add additional complexity for little benefit.

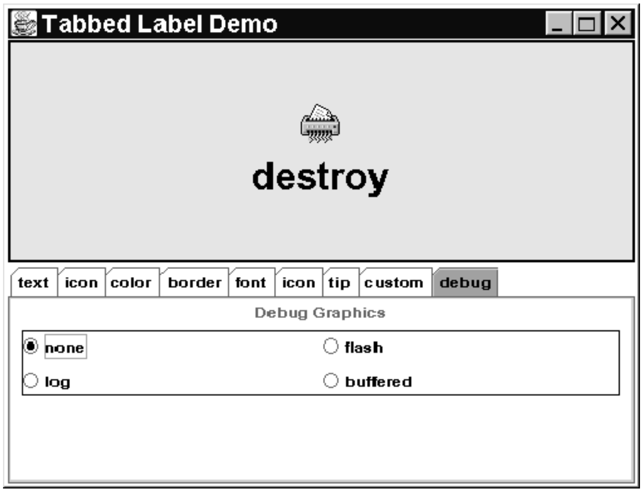
The *CustomColorPanel* class declaration concludes with the *actionPerformed()* method, which will be called when the user presses the *okButton*, as follows.

```
0158     public void actionPerformed( ActionEvent event) {
0159        controllingThis.setBackground(
0160                         feedbackArea.getBackground());
0161        controllingThis.repaint();
0162     }// End actionPerformed.
0163  } // End class CustomColorPanel.
```

This method commences, on lines 0159 to 0160, by obtaining the background attribute of the *feedbackArea* using its *getBackground()* method and using this value as the argument to the *setBackground()* method of *controllingThis()*. Changing the background attribute of a JLabel does not cause it to redisplay itself. Consequently, a call of the *controllingThis* repaint() method, on line 0161, is required to make the change visible to the user.

3.11 DebugGraphics support

The next extension to the *TabbedLabelDemo* artifact adds an instance of the *DebugPanel* which allows the DebugGraphics support supplied by the JFC to be demonstrated. The appearance of this panel is illustrated in Figure 3.22. The construction of this panel,



Debug Graphics

● none ○ flash
○ log ○ buffered

Figure 3.22 *TabbedLabelDemo* showing the *DebugPanel*.

involving a single set of radio buttons in a two column by two row layout, does not differ significantly from other similar panels such as the *BorderPanel*, and will not be presented.

The debug graphics support allows the rendering of JFC components to be investigated in various ways, which can prove useful when debugging an artifact. The first level of support, controlled by the *flash* button on this interface, slows down the drawing of the components and flashes the area of the component about to be drawn on before each part is drawn. For example, if the *Matte* icon button is selected on the *Border* panel and debug options are set to *flash*, the JLabel component will flash each time an instance of the *matteIcon* is placed on the screen to form a part of the border. This allows it to be seen that border is rendered, starting from the top left, clockwise around the component.

The *log* option causes every drawing action to be reported upon on the output stream. If the *TabbedLabelDemo* artifact is being executed as an application this output will, most probably, appear on the console window from where the application was started. If the artifact is being executed as an applet within a browser, the output will, most probably, be captured and available for inspection somehow. For example, if the label has been configured with a yellow background, a line border and a large sans serif bold font, the output produced when it is rendered would be as follows.

```
Graphics(0-1) Enabling debug
Graphics(0-1) Setting new clipRect:
java.awt.Rectangle[x=0,y=0,width=363,height=160]
Graphics(0-1) Setting color: java.awt.Color[r=0,g=0,b=0]
Graphics(0-1) Setting font:
java.awt.Font[family=Sans-serif,name=Sans-serif,style=bold,
size=26]
Graphics(1-1) Setting color: java.awt.Color[r=255,g=255,b=0]
Graphics(1-1) Filling rect:
java.awt.Rectangle[x=0,y=0,width=363,height=160]
Graphics(1-1) Drawing image: sun.awt.windows.WImage@1cc955
at: java.awt.Point[x=164,y=46]
Graphics(1-1) Setting color: java.awt.Color[r=0,g=0,b=0]
Graphics(1-1) Drawing string: "destroy" at:
java.awt.Point[x=134,y=107]
Graphics(0-1) Setting color: java.awt.Color[r=0,g=0,b=0]
Graphics(0-1) Drawing rect:
java.awt.Rectangle[x=0,y=0,width=362,height=159]
Graphics(0-1) Drawing rect:
java.awt.Rectangle[x=1,y=1,width=360,height=157]
Graphics(0-1) Setting color: java.awt.Color[r=0,g=0,b=0]
```

The *buffered* option is only effective when a component is being double buffered and makes the off-screen buffer visible upon the screen in its own frame.

javax.swing.JComponent, debugGraphics attribute

```
public void setDebugGraphicsOptions( int optionMask)
public int  getDebugGraphicsOptions()
```

Sets, or obtains, the debug graphics support for the component and any children. DebugGraphics.NONE_OPTION is the default and causes components to be rendered as normal. Other values, which can be bitwise ored together to turn on mutiple options are: DebugGraphics.FLASH_OPTION, which causes the component to flash before every drawing action; DebugGraphics.LOG_OPTION, which outputs onto the default output stream a report on every drawing action; and DebugGraphics.BUFFERED_OPTION, which shows the off-screen graphics buffer in a separate window. The FLASH and LOG options are only effective if double buffering is turned off and the BUFFERED option is only effective when it is turned on.

These operations are controlled by the setDebugGraphicsOptions() method introduced into the hierarchy by the JComponent class and so available to all JFC components. If the action is used on a container instance then the debug options apply to all components contained within it. The *DebugPanel* constructor contains nothing which has not already been shown in the previous sections. The implementation of its *itemStateChanged()* method is as follows.

```
public void itemStateChanged( ItemEvent event) {

   String debugChoice = ((JRadioButton)
                         event.getItem()).getText();
   RepaintManager manager =
           RepaintManager.currentManager( controllingThis);

      if ( debugChoice.equals( "none")) {
         manager.setDoubleBufferingEnabled( true);
         controllingThis.setDebugGraphicsOptions(
                            DebugGraphics.NONE_OPTION);
      } else if ( debugChoice.equals( "flash")) {
          manager.setDoubleBufferingEnabled( false);
          DebugGraphics.setFlashTime( 125);
          DebugGraphics.setFlashCount( 3);
          controllingThis.setDebugGraphicsOptions(
                            DebugGraphics.FLASH_OPTION);
      } else if ( debugChoice.equals( "log")) {
          manager.setDoubleBufferingEnabled( false);
          controllingThis.setDebugGraphicsOptions(
                            DebugGraphics.LOG_OPTION);
      } else if ( debugChoice.equals( "buffered")) {
          manager.setDoubleBufferingEnabled( true);
          controllingThis.setDebugGraphicsOptions(
                            DebugGraphics.BUFFERED_OPTION);
      } // End if.
   } // End itemStateChanged.
```

The four branches of the selection in this method each call the *controllingThis* setDebugGraphicsOptions() method with an appropriate argument: DebugGraphics.NONE_OPTION, DebugGraphics.FLASH_OPTION, DebugGraphics.LOG_OPTION or DebugGraphics.BUFFERED_OPTION. These manifest values are masks for the options and so can be combined together, using the or operator (|), to turn on more than one

```
com.sun.java.swing.JFrame[frame0,0,0,156x66,layout=java.awt.BorderLayout,resizable,title=First JLabel Demo]
  com.sun.java.swing.JRootPane[,4,29,148x33,layout=com.sun.java.swing.JRootPane$RootLayout]
    com.sun.java.swing.JPanel[null.glassPane,0,0,148x33,hidden,layout=java.awt.FlowLayout]
    com.sun.java.swing.JLayeredPane[null.layeredPane,0,0,148x33]
      com.sun.java.swing.JPanel[null.contentPane,0,0,148x33,layout=com.sun.java.swing.JRootPane$1]
        chap3.FirstJLabelDemo[panel0,0,0,148x33,layout=java.awt.BorderLayout]
          com.sun.java.swing.JRootPane[,0,0,148x33,layout=com.sun.java.swing.JRootPane$RootLayout]
            com.sun.java.swing.JPanel[null.glassPane,0,0,148x33,hidden,layout=java.awt.FlowLayout]
            com.sun.java.swing.JLayeredPane[null.layeredPane,0,0,148x33]
              com.sun.java.swing.JPanel[null.contentPane,0,0,148x33,layout=com.sun.java.swing.JRootPane$1]
                com.sun.java.swing.JLabel[,0,0,148x33]
```

Figure 3.23 *FirstJLabelDemo* window hierarchy dump.

option at a time. For the *flash* option the DebugGraphics class-wide attributes that determine the number of flashes and the rate of flashing (in milliseconds) are also set.

It is necessary with the FLASH and LOG options for double buffering of the painting options to be turned off and for the BUFFERED option for it to be turned on. This can be accomplished by obtaining the identity of the RepaintManager for the *controllingThis* JLabel instance and calling its setDoubleBufferingEnabled() method, as appropriate. Double buffering involves first drawing the image on an invisible off-screen image and then copying the image from the off-screen image to the on-screen window.

The JFC components have one other facility that is useful when debugging an artifact. If the `CTRL`, `SHIFT` and `F1` keys are all pressed simultaneously then a text dump of the window hierarchy will be sent to the standard output stream. Figure 3.23 illustrates the output produced when a dump of the *FirstJLabelDemo*, as illustrated in the left-hand image in Figure 3.1, was obtained.

These facilities can be useful if an interface, or a part of the interface, is not appearing as expected. However, like most debugging tools, they produce a vast amount of information which can occlude the actual fault. The most effective way to use them is to form a theory about exactly what might not be happening, or what might be happening incorrectly, and use the facilities to attempt to prove or disprove the theory.

3.12 The IconPositionPanel revisited – the IconicIconPositionPanel

In this extension the icon position panel from Section 3.4 will be revisited and revised in order to make use of iconic, rather than textual, communication. The appearance of the revised panel, an instance of the *IconicIconPositionPanel*, with the same positioning of the icon and text as in Figure 3.5, is given in Figure 3.24.

The images in Figure 3.24 show that the radio button markers have not been displayed as a part of each JRadioButton; instead, two different icons are being used. One is in full color (although this can only be seen as darker shades of gray when reproduced as a non-color image) indicating that the radio button is currently selected. The alternative image, indicating that the radio button is not currently selected, is a gray scale icon (seen as lighter shades of gray when reproduced).

Just as the ButtonGroup took care of setting and resetting the radio button markers when the user selected an option in the original implementation, in this implementation it will take care of displaying the option that the user has selected using the selected (color) icon and ensure that all other buttons in the group are displayed with their deselected (gray) icon.

A JRadioButton can have three different icons associated with it: the default icon that is shown when it is not currently selected, the pressed icon which is shown between the

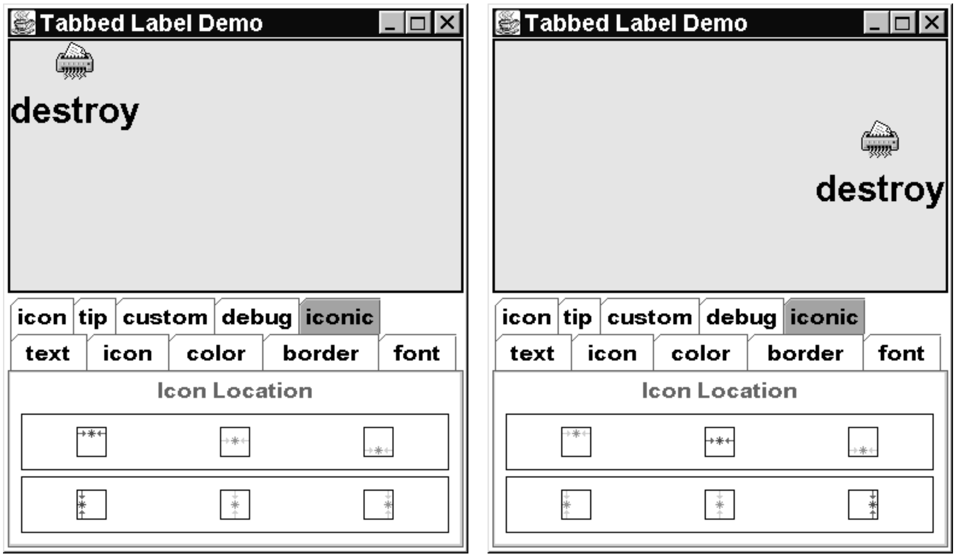


Figure 3.24 *TabbedLabelDemo* showing the *IconicIconPositionPanel*.

time the user presses the mouse button on top of the button and releases it, and the selected icon which is shown when the button is selected. If the default and selected icon attributes are specified for an entire ButtonGroup of JRadioButtons then the JFC infers that the radio button markers are not required and does not show them.

In this example the pressed icon is specified to be the same as the selected icon so that, as the user interacts with the buttons, they will get a positive feedback that pressing the mouse button down has had an effect. If this were not done then a change of button state would not become visible until the mouse button were released, which would give no closure of initiation of the interaction.

To prepare for the implementation of the *IconicIconPositionPanel* class a total of 12 icon images were prepared and given systematic names. For example, the pair of icons which might appear at the bottom right of the two images in Figure 3.23 were given the names `verright.gif` (vertical arrows at the right) and `verrightg.gif` (gray vertical arrows at the right).

One other difference between the visual design of Figures 3.5 and 3.24 is also apparent. Figure 3.24 shows that if the TabbedPane does not have sufficient width to display all of the tabs in a single row it will lay them out using more than one row. The tabs which were added first will be shown on the lowest row, closest to the area that they are controlling.

The fragment of the *IconicIconPositionPanel* which constructs and configures the topRadio button, shown as the upper left button in Figure 3.24, is as follows. The constructor specifies only its default (gray) icon (`hortopg.gif`), without any text label. Once constructed, on lines 0037 and 0038, the selectedIcon and pressedIcon resources of the JRadioButton are both specified as the active (color) icon (`hortop.gif`). As it is known that all the icons will be the same size, on line 0040 the image is horizontally centred within its available space; rather than being left aligned as in the original implementation.

```
0035        JRadioButton topRadio =
0036              new JRadioButton( new ImageIcon(  "hortopg.gif"));
0037         topRadio.setSelectedIcon( new ImageIcon( "hortop.gif"));
0038         topRadio.setPressedIcon(  new ImageIcon( "hortop.gif"));
0039         topRadio.setHorizontalAlignment( SwingConstants.CENTER);
0040         topRadio.setHorizontalAlignment( SwingConstants.CENTER);
0041         topRadio.setName( "Top");
```

The final step in the construction and configuration of the *topRadio* button, on line 0041, is to give it a *name*. In the previous implementation the *itemStateChanged()* method was able to decide which of the JRadioButtons had been selected by examining the text resource of the *item* attribute of the ItemEvent. Hence if the user selected the radio button labelled *Middle*, the item attribute would be the JRadioButton itself and its text attribute would be *Middle*. However, as these JRadioButtons do not have text labels associated with them this technique will no longer work. Instead, the *name* attribute of the JRadioButton, introduced into the class hierarchy by the AWT Component class and hence available to all AWT and JFC components, is being used instead.

The remaining parts of the *IconicIconPositionPanel* constructor contain no surprises and will not be presented. The steps omitted include constructing and configuring the five remaining JRadioButton instances, configuring the panels, registering the panel as the ItemListener for all six radio buttons, registering the buttons with their ButtonGroups and finally assembling the components with the required layout relationships.

The only other method in the *IconicIconPositionPanel* class is *itemStateChanged()* which has to be supplied as this class promises to implement the ItemListener interface, and must do so in order to be able to listen to the ItemEvents generated by the JRadioButtons. The first part of its implementation is as follows.

```
0122         public void itemStateChanged( ItemEvent event) {
0123
0124         String location = ((Component) event.getItem()).getName();
0125
0126            if ( location.equals( "Left")) {
0127               controllingThis.setHorizontalAlignment(
0128                                                 SwingConstants.LEFT);
```

The equivalent fragment of the *IconPositionPanel*'s *itemStateChanged()* method was as follows.

```
0122         public void itemStateChanged( ItemEvent event) {
0123
0124         String location = ((JRadioButton)
0125                                       event.getItem()).getText();
0125
0126            if ( location.equals( "Left")) {
0127               controllingThis.setHorizontalAlignment(
0128                                                SwingConstants.LEFT);
```

The only significant difference is that the revised implementation casts the item attribute of the ItemEvent to AWT Component and then retrieves its name attribute. The original implementation cast it to JFC JRadioButton and then retrieved its *text* attribute. This, as explained above, is necessary, as the radio buttons in this implementation do not have any *text* resource. This is also a preferable technique as the text attribute should change according to the linguistic locale that the artifact is being used within, as will be demonstrated in the following chapters. Consequently the *text* label cannot be relied upon not to change, whereas the *name* attribute is purely internal and so can be relied upon not to change.

JComponent → javax.swing.AbstractButton, image resources

```
public void setIcon(Icon defaultIcon)
public Icon getIcon()
```

Establishes or obtains the identity of the *defaultIcon*, this icon will be displayed in all circumstances unless one or more of the more specialized icons listed below have been specified.

```
public void setDisabledIcon(Icon disabledIcon)
public Icon getDisabledIcon()
public void setPressedIcon(Icon pressedIcon)
public Icon getPressedIcon()
public void setSelectedIcon(Icon selectedIcon)
public Icon getSelectedIcon()
public void setRolloverIcon(Icon rolloverIcon)
public Icon getRolloverIcon()
public void setRolloverSelectedIcon(Icon rolloverSelectedIcon)
public Icon getRolloverSelectedIcon()
public void setDisabledSelectedIcon(Icon disabledSelectedIcon)
public Icon getDisabledSelectedIcon()
```

The disabled icon is shown if the button is not sensitive to the user's interactions (following a call of setEnabled(**false**)). The pressed icon is displayed between the mouse button being depressed, while the pointer is over the button, and released. The selected icon is shown by toggle buttons and radio buttons when they are selected by the user. A rollover occurs while the mouse pointer transits across the button.

3.13 The *LookAndFeel* (L&F) panel

In this final extension to the *TabbedLabelDemo* artifact a panel to control the artifact's *look and feel* will be developed and presented. A *look and feel* (L&F) determines the overall visual appearance of the components contained in a set of widgets and the way in which the user interacts with them.

The three most established look and feels are those supplied by Apple computers for the Macintosh series of computers, by Microsoft for their series of operating systems and X/Motif supplied by the Open Software Foundation (OSF) for use with Unix and the X Windowing system. Other look and feels may be required for other requirements, for example the L&F used by personal organizers such as the Psion series of products. The ability of Java artifacts to have an L&F plugged in at run time implies that the same product could be supplied for use both on different desktop computers and on handheld devices, adapting its L&F as appropriate.

When Java was first developed it was thought that users of a particular computer would insist upon artifacts using the native platform look and feel. Accordingly, the JFC was designed to allow different look and feels to be plugged into the same Java artifact at run time, allowing it to adopt the native L&F. The success of products such as Netscape showed that users would accept a cross-platform L&F and the JFC has a default

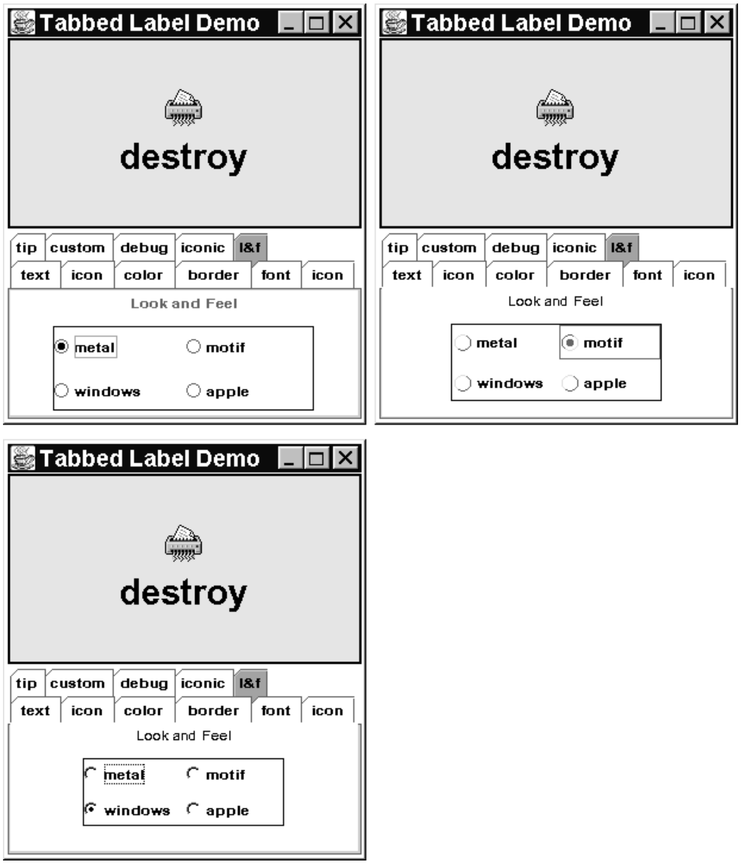


Figure 3.25 *TabbedLabelDemo* showing the *LookAndFeelPanel*.

cross-platform L&F, known as *Metal*. Despite this it still retains and supports the ability for different look and feels to be installed. It is also possible for a software house to define and supply its own L&F in order to make its products distinctive. However, the techniques for achieving this are outside the scope of this book and only the four look and feels supported by the JFC distribution will be considered here.

The appearance of the *LookAndFeelPanel* is illustrated in Figure 3.26. The upper left image shows the default *Metal* L&F supplied by the JFC. The upper right shows the *Motif* L&F and the lower left the MS *Windows* L&F. These three look and feels are visually very similar. One of the most obvious of the subtle differences between them is the appearance of the radio button markers. The Motif and MS Windows look and feels were both developed so as to be visually similar to one initially developed by IBM called the *C*ommon *D*esktop *E*nvironment (CDE). The metal L&F was produced to be visually comparable with these two, but sufficiently distinct to be recognizable. Only the *Apple* L&F, which is not shown, is dramatically different from the other three, reflecting the independent development of it by Apple. The differences between the images in Figure 3.25 are not particularly dramatic, Figure 3.26 shows the detailed appearance of a slider from the *CustomColorPanel* which shows how different the three appearances can be.

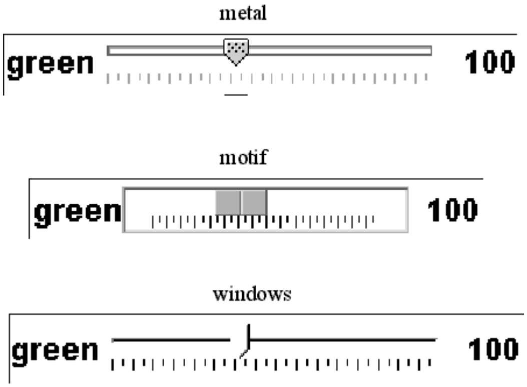


Figure 3.26 *CustomColorPanel* sliders with different look and feels.

One major difference between this panel and the previous panels is that the changes controlled by the panel are applied to the artifact as a whole and not just to the JLabel instance. Accordingly, the constructor requires the identity of the applet it is contained within, not the identity of the JLabel instance it is *toControl*. The declaration of the *LookAndFeelPanel*'s attribute and constructor declaration is as follows.

```
class LookAndFeelPanel extends JPanel
                    implements ItemListener {

private Component topLevel  = null;
private Applet    theApplet = null;


    protected LookAndFeelPanel( Applet applet) {
```

The identity of the applet supplied to the constructor will be stored in the *theApplet* instance attribute for use by the *itemStateChanged()* method, as will be described below, along with the use made of the other attribute, *topLevel*. The consequential change in the call of the constructor from the *TabbedLabelDemo*'s *init()* method is as follows.

```
  tabbedPane.addTab("iconic",
                  new IconicIconPositionPanel( demoLabel));
  tabbedPane.addTab("l&f",
                  new LookAndFeelPanel( this));
```

The remainder of the implementation of the *LookAndFeelPanel*'s constructor does not differ significantly from the other panels, such as the *ColorPanel*, that contain a single group of radio buttons. All of the JRadioButton instances on panel have **this** *LookAndFeelPanel* instance registered as their itemListener resource and so all will dispatch ItemEvents to its *itemStateChanged()* method, the first part of which is implemented as follows.

```
    public void itemStateChanged( ItemEvent event) {

    String lafChoice = ((JRadioButton)
                          event.getItem()).getText();
```

```
0076         if ( topLevel == null) {
0077            try {
0078                theApplet.getAppletContext();
0079                topLevel = theApplet;
0080            } catch ( NullPointerException exception) {
0081                topLevel = theApplet;
0082                while ( ! ( topLevel instanceof JFrame)) {
0083                   topLevel = topLevel.getParent();
0084                } // End while.
0085            } // End try catch.
0086         } // End if.
```

When this method is called for the first time the *topLevel* attribute will still have its default, **null**, value and the code between lines 0076 and 0086 will be executed. A change of look and feel should be effected upon the top-level component of the artifact, causing it to be propagated downwards throughout the whole hierarchy. For an artifact executing as an applet within a browser this will be the *applet* itself. For an artifact executing as an application this will be the JFrame that it is contained within. The effect of the **if** structure is to record the appropriate identity in the *topLevel* attribute, using techniques that were fully explained in Chapter 2. The remaining part of the *itemStateChanged()* method is as follows.

```
0089         try {
0090            if ( lafChoice.equals( "metal")) {
0091                UIManager.setLookAndFeel(
0092               "javax.swing.plaf.metal.MetalLookAndFeel");
0093            } else if ( lafChoice.equals( "motif")) {
0094                UIManager.setLookAndFeel(
0095               "javax.swing.plaf.motif.MotifLookAndFeel");
0096            } else if ( lafChoice.equals( "windows")) {
0097                UIManager.setLookAndFeel(
0098           "javax.swing.plaf.windows.WindowsLookAndFeel");
0099            } else if ( lafChoice.equals( "apple")) {
0100                UIManager.setLookAndFeel(
0101                   "javax.swing.plaf.mac.MacLookAndFeel");
0102            } // End if.
0103            SwingUtilities.updateComponentTreeUI( topLevel);
0104         } catch (Exception exception) {
0105             System.err.println( "L&F not supported!");
0106         } // end try catch
0107      } // End itemStateChanged.
```

The first part of this fragment is a four-way selection calling the UIManager's setLookAndFeel() method with an appropriate argument identifying the package containing the L&F details. This attempt to install a new L&F may throw an exception, in which case a message is output onto the standard error stream and the program continued with the existing L&F. Otherwise the SwingUtilities updateComponentTree() method is called, passing *topLevel* as its argument. This method will effect the L&F change on all the components in the window hierarchy below that passed as its argument.

3.14 The *TabbedLabelDemo* in retrospect

To conclude this chapter the relationships between the various AWT and JFC component classes that have been used will be commented upon. A class hierarchy diagram of these

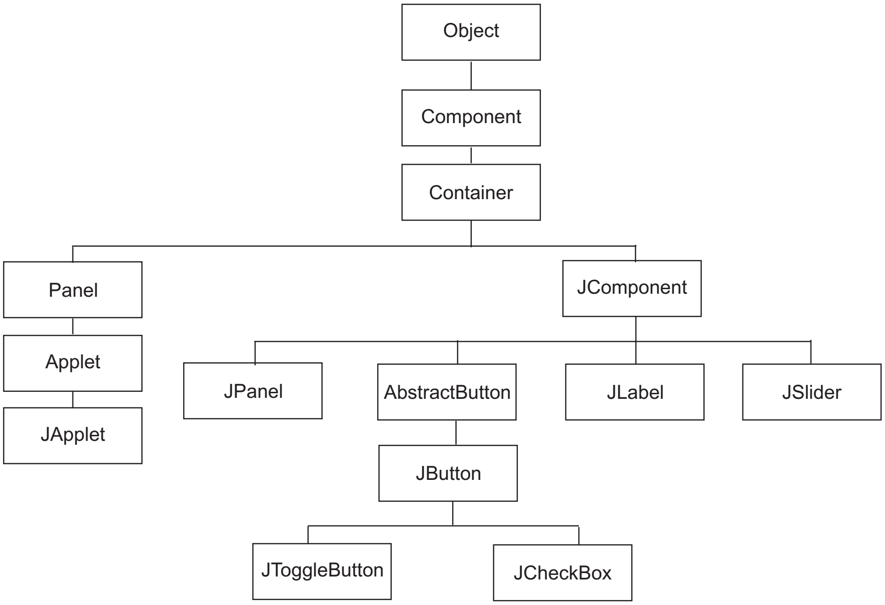


Figure 3.27 Partial AWT and JFC class hierarchy.

is given in Figure 3.27. The diagram shows that all JFC classes, are descended from the AWT Container class and hence all inherit the capability to have instance children and to manage their layout. However, many of the classes, for example JLabel, seem to make little or no use of this capability.

All of the JFC classes, apart from JApplet, are descended from JComponent and hence inherit all the resources that it introduces into the hierarchy. For example, the ability to have a border or support tool tips is introduced by JComponent, so the details of these facilities given in this chapter will be applicable to any other JFC component. Some other attributes, for example foreground and background colors, are introduced by the AWT Component class, and so are even more fundamental and applicable, as are the details of the AWT Color class.

Other resources are introduced at different places below JComponent. For example, the horizontal and vertical alignment and positioning attributes, which were introduced at the very start of this chapter, are independently introduced by both the JLabel class and the AbstractButton class. Hence the knowledge of how to use these attributes is applicable only to classes in that part of the hierarchy.

The set of icon resources introduced in this chapter, the default, disabled, selected, pressed and rollover icon attributes, are supplied by the AbstractButton class. Consequently the knowledge of how to use these attributes can be applied to all three of the button classes which can appear on user interfaces.

Likewise, although not shown in Figure 3.27, the examples of event handling, including ActionEvents, ItemEvents and ChangeEvents, which have been introduced so far, are applicable to any event source and listener combination that may be met in any situation. Furthermore the fundamental registration, dispatch and listening relationships will be applicable to any other kinds of events that are subsequently introduced. This applicability is also the case for the layout mechanisms so far introduced, including FlowLayout, GridLayout, BorderLayout and JTabbedPane.

Table 3.1 Resource declarations and settings for the TabbedLabelDemo artifact.

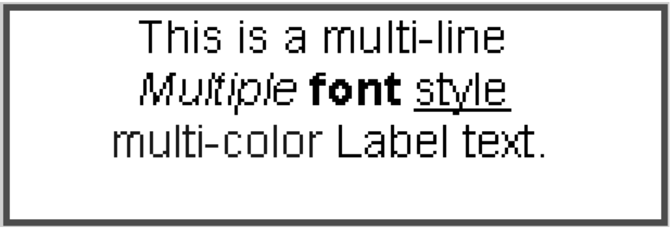
Resource class	Identifier	Constructor value
`ColorUIResource`	`defaultBackground`	`ColorUIResource( Color.white);`
`ColorUIResource`	`defaultForeground`	`ColorUIResource( Color.black);`
`FontUIResource`	`FontUIResource`	`FontUIResource( Font( "Dialog", Font.BOLD, 18));`
`FontUIResource`	`smallerFont`	`FontUIResource( Font( "Dialog", Font.BOLD, 12));`

Resource name	Resource value
`Button.background`	`defaultBackground`
`Button.foreground`	`defaultForeground`
`Button.font`	`smallerFont`
`CheckBox.background`	`defaultBackground`
`CheckBox.foreground`	`defaultForeground`
`CheckBox.font`	`defaultFont`
`Label.background`	`defaultBackground`
`Label.foreground`	`defaultForeground`
`Label.font`	`defaultFont`
`Panel.background`	`defaultBackground`
`Panel.foreground`	`defaultForeground`
`RadioButton.background`	`defaultBackground`
`RadioButton.foreground`	`defaultForeground`
`RadioButton.font`	`defaultFont`
`Slider.background`	`defaultBackground`
`TextField.font`	`smallerFont`
`TabbedPane.font`	`smallerFont`
`TabbedPane.foreground`	`defaultForeground`
`TabbedPane.background`	`defaultBackground`

Hence, although the artifact developed in this chapter, unlike the one presented in Chapters 1 and 2, does not seem to have any practical use, it has provided a mechanism by which a large number of fundamental considerations can be introduced and explained. For the sake of completeness the resource settings effected by the *TabbedLabelDemo*'s *setResources()* method are given in Table 3.1. The first part of the table indicates the classes, identifiers and values supplied to the constructors of the four required resources. The second part of the table gives an alphabetical list of the resource names and values which are installed by the *setResources()* method.

Many of these simple components that support a text attribute are capable of interpreting and rendering a subset of *H*yper*T*ext *M*arkup *L*anguage (HTML). HTML will be introduced in detail in Chapter 8, it is a plain text format which contains tags embedded between chevrons (< and >). When the text is rendered the tags are interpreted to determine how it is to be presented, allowing multiple line phrases with different fonts, font sizes, colors and qualities to be shown. Figure 3.28 shows the appearance of a JLabel instance whose HTML text resource is also shown.

In the next chapter an introduction will be made to the techniques by which an interactive JFC Component can be extended to fulfill a requirement which none of the existing components is capable of fulfilling.



This is a multi-line
Multiple **font** style
multi-color Label text.

```
demoLabel.setText( "<html><centre>This is a multi-line<br>" +
                   "<i>Multiple</i> <b>font</b> <u>style</u><br>" +
                   "<font color=blue>multi-color</font> " +
                   "Label text.</centre></html>");
```

Figure 3.28 JLabel instance with HTML text resource.

Summary

- The ImageIcon interface class can be used to supply icons to components.
- When executing as an applet, artifacts must obtain images (and other resources) via URL connections.
- Grouping of components should be used to indicate operational functionality.
- A ButtonGroup instance will identify a set of JRadioButtons that should behave as a single set.
- JRadioButtons and JCheckBoxes dispatch ItemEvents and consequently require an ItemListener instance to receive them.
- The ItemListener interface mandates a single method called itemStateChanged().
- An ItemEvent has itemSelectable, item, id and a manifest (SELECTED or DESELECTED) attributes.
- A tabbedPane layout manager displays only one of its children at a time, as indicated by the selected tab.
- The swing.border package supplies a number of border classes, which can be installed into any JFC component.
- Different icons can be supplied to some JFC components and will be automatically used to indicate the component's state.
- All JFC components support a toolTip attribute, which will be displayed when the mouse pointer rests over the component.
- JTextFields allow a single line of text to be typed in and fire an ActionEvent when the user presses the ENTER key.
- JSliders can be used to indicate a numeric input and fire a ChangeEvent when the user moves its pointer.
- The ChangeListener interface mandates a single method called itemStateChanged().
- A ChangeEvent supports only a source and id attribute.
- The operations used to paint JFC components can be investigated, in various ways, by setting their debugGraphics attribute.
- JFC components support different look and feels, which can be changed at run-time.

Exercises

3.1 Revisit the *Stopwatch* artifact from Chapter 1 and install enabled and disabled icons into its buttons. Further improvements can be made by the use of borders and tool tips.

3.2 Use the JFC documentation to investigate the JColorChooser class and install an instance of it into a revised version of the ColorPanel class.

3.3 In the *TabbedLabelDemo* artifact, as presented in this chapter, there are two panels that control the background color of the JLabel instance. Hence it is possible for a disparity to occur between the settings on the control panels and the color displayed. A JTabbedPane fires a ChangeEvent every time the user selects one of the tabs. Add a ChangeListener to the tabbed pane and use it to implement a mechanism whereby the appearance of the JLabel instance is always in accord with the color panel that is displayed.

3.4 Reimplement the *fontPanel* using a JSlider instance to indicate the font size.

3.5 Design and implement a specialized panel that allows the positioning of a TitledBorder's title to be interactively explored.

3.6 Design and implement a series of panels that allow an artifact's font and color resources to be interactively explored. Add a suitable control panel to the Stopwatch artifact from Chapter 1 and use to investigate different settings. Hint: the SwingUtilities updateComponentTree() method will have to be used to make any changes visible.

4

Specialized components – the numeric input hierarchy

4.1 Introduction

The *TabbedLabelDemo* artifact in Chapter 3 briefly introduced an instance of the JTextField class as a single-line text input widget that generated an ActionEvent when the user pressed the ENTER key within it. The first part of this chapter will extend the introduction of the JTextField class by using it as the base class of a specialized text field class called *NumericTextField*. Instances of this class will only allow the user to enter numeric values; it also supplies a method to obtain the value entered as a **double**.

In order to accomplish this a different technique for the extension of existing components will have to be introduced. In Chapter 2, when the *FontViewerTextArea* class was introduced as an extension of the JComponent class, it was not necessary for the class to support any interactive behavior with the user and hence did not need to be concerned with event handling. In the *NumericTextField* class it will be necessary for the extended class to be aware, at all times, of exactly what the user is doing while interacting with it. In order to accomplish this it will have to intercept and handle events sent to the component before any registered listeners are notified of the user's actions.

Once the *NumericTextField* has been developed and demonstrated it will be further extended to supply an integer-only text input field called *IntegerTextField*, and then this will be used as a constituent part of a *SpinBox* component. A *SpinBox* is an integer input area combined with two button-like controls that allow the value to be incremented and decremented.

4.2 Internal event handling

In all the examples introduced so far event handling has made use of the event source/listener mechanism. This mechanism is appropriate when the requirement is to make use of a component's essential behavior. For example, the essential behavior of a JButton is to be pushed, and it sends ActionEvents to its listeners to inform them when this happens. Likewise, the essential behavior of a JSlider is to be moved and it sends ChangeEvents to its listeners when this happens. However, both JButtons and JSliders also have internal behavior: when a button is pressed it will give some indication that it has been primed and will activate when it is released. A slider redraws the position of its pointer as it is dragged, but only fires events to its listeners when dragging stops. Both of these are examples of the essential internal behavior, which are required by every button or slider instance and hence are part of the definition of the class. The consequences of

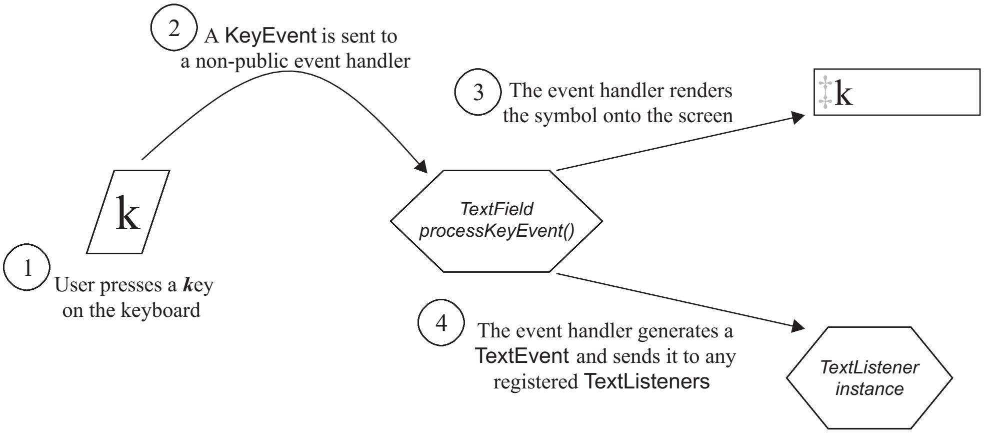

2 A KeyEvent is sent to a non-public event handler
3 The event handler renders the symbol onto the screen
k
k
TextField processKeyEvent()
1 User presses a *k*ey on the keyboard
4 The event handler generates a TextEvent and sends it to any registered TextListeners
TextListener instance

Figure 4.1 (Simplified) event handling in a JTextField.

pressing a button or dragging a slider are external to the component and will be different for every instance used.

The external, consequential, behavior of a component should be implemented by making use of the event source/event listener mechanism. The internal, essential, behavior of a component is an integral part of its class definition and should, where possible, make use of a different mechanism. This mechanism, which will be explained in detail below, intercepts events generated by the user's behavior, such as mouse movements or keyboard key presses, and implements the required essential behavior before, possibly, sending events on to any registered listeners.

Figure 4.1 shows a simplification of the processes involved in JTextField event handling when the user presses a key. The pressing of the key will cause the Java run-time system to generate a KeyEvent containing details of exactly which key was pressed. This event will be dispatched to a non-public event-handling method called processKeyEvent(). This method will first cause the symbol associated with the key to be rendered onto the screen at the location of the text field's insertion cursor. It will then generate a TextEvent, indicating that the text within the text field has changed, and send it to any registered TextListeners. If the user presses the ENTER key a similar pattern of actions will occur, but there will be no effect upon the appearance of the text field and an ActionEvent will be sent to any registered listeners. By default the actionCommand attribute of the event will contain the contents of the text field.

The rendering of the symbols onto the screen and the dispatch of events to listeners are essential internal behaviors of a JTextField and consequently are implemented privately within the class. The consequences of users entering text into the field, or indicating that they have finished entering text by pressing ENTER, depend upon exactly what an instance is being used for within an artifact and rely upon external processing by the event listener mechanism.

One of the simplifications in this diagram is that an inputMethod object sits between the keyboard and the component. Its task is to collapse multiple key presses into a single Unicode character. For example in some romanized alphabets characters such as those which have diacritical marks (e.g. ë, î, ð) or those that are diphthongs (e.g. 'æ') are produced from a standard keyboard by a sequence of key presses. When input in

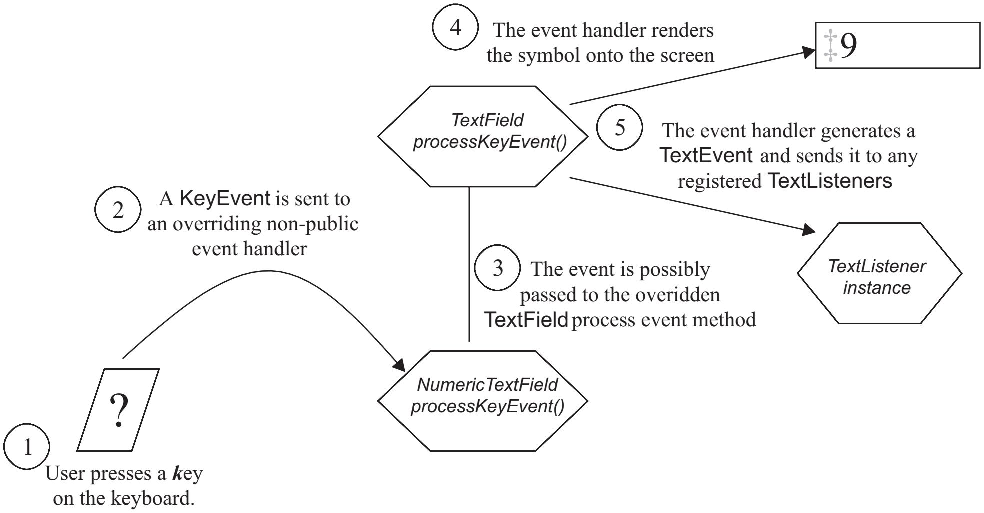

4 The event handler renders the symbol onto the screen
9
TextField processKeyEvent()
5 The event handler generates a TextEvent and sends it to any registered TextListeners
2 A KeyEvent is sent to an overriding non-public event handler
3 The event is possibly passed to the overidden TextField process event method
TextListener instance
NumericTextField processKeyEvent()
?
1 User presses a *k*ey on the keyboard.

Figure 4.2 (Simplified) event handling in a NumericTextField.

ideographic languages such as Japanese, Chinese or Korean is required, the inputMethod object has a complex task and is an input dialog in its own right. However, for simple input requirements the inputMethod object can be more of a hindrance than a help.

In order to implement a *NumericTextField* the essential behavior of a JTextField will have to be changed, and a simplified overview of the changes is given in Figure 4.2.

In this design the KeyEvent is passed to the overriding *NumericTextField processKeyEvent()* method to decide whether the key press should be responded to or ignored. For example, the digit keys (‘0' to ‘9') should always be responded to and the alphabetic keys should always be ignored. If the key is to be responded to, the event is passed on to the inherited TextField processKeyEvent() method and processed as before. If it is to be ignored, nothing need happen.

4.3 *NumericTextField*, STD and class design

A State Transition Diagram for the required behavior of a *NumericTextField* is given in Figure 4.3. As it only has a single state, labeled as *numeric text field*, it suggests that it should be easy for a user to operate it. However, as two of the six transitions have conditions associated with them it can also be predicted that some users will have difficulty with these transitions.

The transitions show that in all cases any presses of the digit keys should be processed as normal; likewise, any use of the editing keys (backspace and delete) or navigation keys (left arrow, right arrow, home and end). The period key (decimal point ‘.’) should only be processed if there is not already a decimal point in the field. The sign keys (plus ‘+’ and minus ‘–’) should only be processed if the insertion cursor is at the start of the input area and if there is not already a sign character at that location. The only other transition shows that the ENTER key should cause an ActionEvent to be dispatched to any registered listeners. All other keys, as they are not shown as event labels on any transitions, will be ignored.

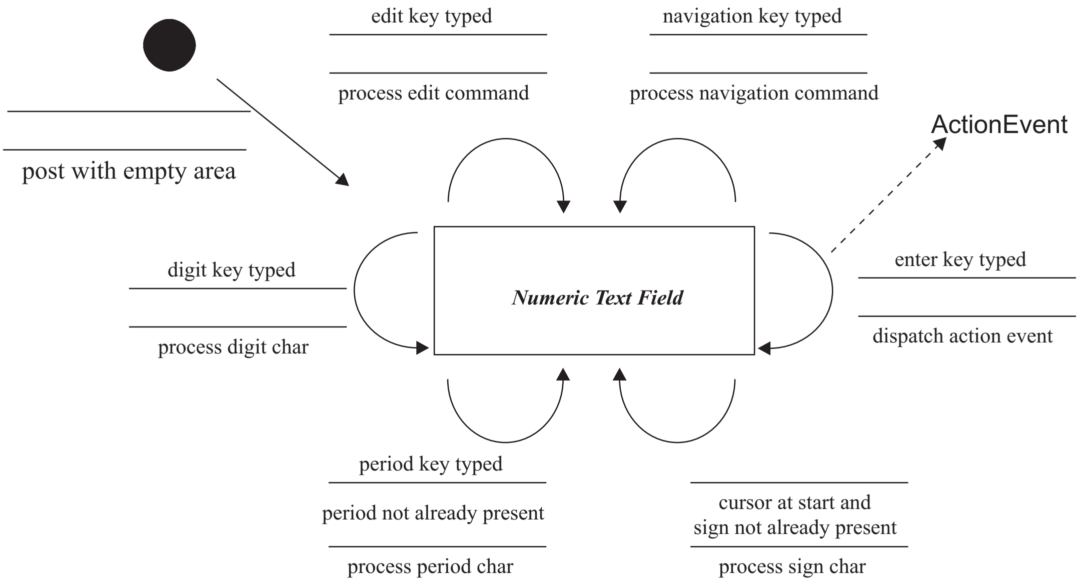


Figure 4.3 *NumericTextField* state transition diagram.

This design is illustrating the user interface design principle that prevention of 'errors' is better than having to correct them later. With this design it is not possible for the user to enter anything other than a sequence of characters that can be interpreted as a floating-point number. An alternative possibility would be to supply a simple text field and instruct the user to enter a floating-point number. It can be predicted that at some stage a user would not attend to the instruction and enter an invalid sequence requiring this 'error' to be corrected in some way.

The class diagram for the *NumericTextField* class is given in Figure 4.4 and shows that it is a member of the *numericinput* package of classes and that it extends the JTextField class, overriding its default constructor. It supplies two inquiry methods to obtain the numeric value from the field, called *getValue()* and *getFloatValue()*. The value is primarily considered as a **double** value and the *getFloatValue()* method is supplied as a convenience for users of the class. Both methods return the value entered into the text field by the user, or zero if it is empty. The *setValue()* method takes a **double** argument, which by automatic upward casting can be supplied as a **float** value, and installs it into the area.

The remaining public method is *setText()*, which overrides the setText() method inherited from JTextField. If this method were not overridden it would provide a method that could compromise the integrity of the component by allowing any text string to be placed in the area. The overriding method will only place the String supplied as an argument into the area if it can be interpreted as a floating-point value; otherwise it does not change the contents of the component.

The remaining method, *processKeyEvent()*, must be declared as a **protected** method as it is concerned with the internal handling of events, in contrast to event-listening methods, such as actionPerformed(), which are concerned with the **public** handling of events.

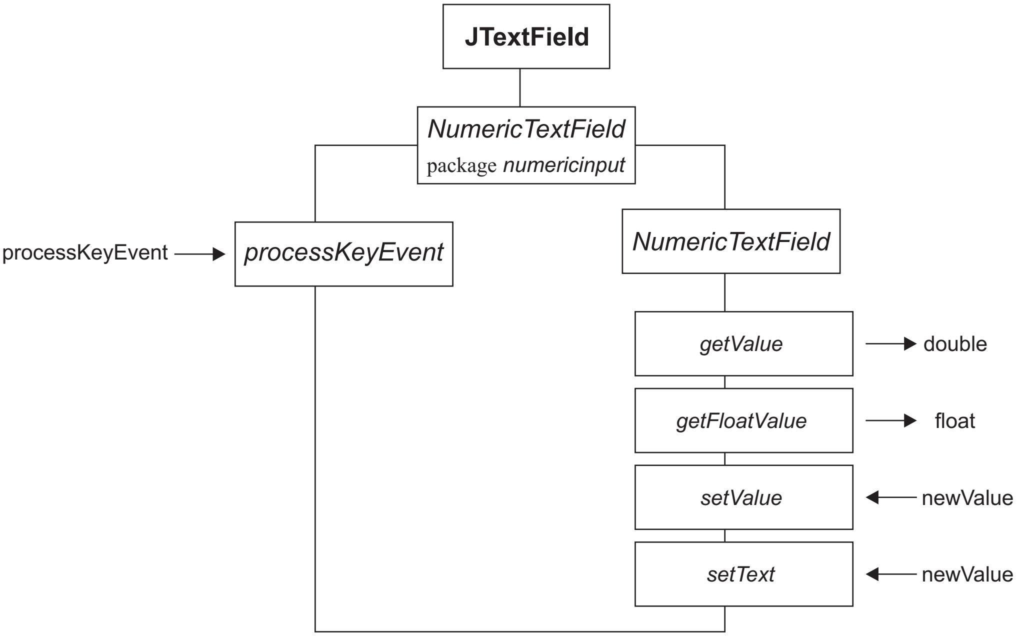


Figure 4.4 NumericTextField class diagram.

4.4 *NumericTextField* implementation

The first part of the implementation of the *NumericTextField* class, as far as the end of the constructor, is as follows.

```
0006  package numericinput;
0007
0008  import java.awt.event.*;
0009  import javax.swing.*;
0010
0011  public class NumericTextField extends JTextField {
0012
0013     public NumericTextField() {
0014        super();
0015        this.setInputMethod( false);
0016        this.enableEvents( AWTEvent.KEY_EVENT_MASK);
0017     } // End NumericTextField constructor.
```

Following the importation of the AWT event and swing packages, the declaration of the constructor commences with a call of the **super**, TextField, default constructor. Following this, on line 0015, the inputMethod attribute is set **false** to indicate that events from the keyboard should be sent directly to the component and not intercepted to determine whether they might be parts of a composite character, as explained above.

The remaining action of the constructor is to enable key events upon itself. By default the Java run-time environment does not pass events to a component unless it indicates that it is interested in them. One way to indicate an interest is to register a listener for a particular class (or classes) of events. So when, in Chapter 3, the JTextField in the

TabbedLabelDemo class registered an ActionListener with addActionListener(), one consequence of this was to enable ActionEvents upon itself. This ensured that any ActionEvents generated in response to the user's actions were passed to the component in order to be dispatched to its listeners.

When a component is being extended, as in this case, as opposed to simply being used, the appropriate classes of events must be explicitly enabled. This is accomplished by calling the enableEvent() method, which is introduced into the hierarchy by the AWT Component class. The argument to enableEvent() is a mask constructed from the manifest values provided by the AWTEvent class. In this case only events associated with the keyboard need to be enabled, and the appropriate mask is identified as KEY_EVENT_MASK. If for some reason this class had also to respond to mouse button presses then the MOUSE_EVENT_MASK would have to be combined with the KEY_EVENT_MASK using the bitwise or operator (|), as in KEY_EVENT_MASK | MOUSE_EVENT_MASK.

AWT Component explicit event processing

```
protected final void enableEvents(  long manifest)
protected final void disableEvents( long manifest)
```

Indicates to the Java run-time environment that this Component is interested in receiving, or in no longer receiving, particular classes of events. The argument is a bitwise (|) combination of masks supplied by the AWTEvent class including: ACTION_EVENT_MASK, ADJUSTMENT_EVENT_MASK, COMPONENT_EVENT_MASK, CONTAINER_EVENT_MASK, FOCUS_EVENT_MASK, ITEM_EVENT_MASK, KEY_EVENT_MASK, MOUSE_EVENT_MASK, MOUSE_MOTION_EVENT_MASK, TEXT_EVENT_MASK and WINDOW_EVENT_MASK. Events are enabled or disabled automatically when listeners are registered or removed. This method is used when events have to be explicitly processed in an extended component.

```
protected void processEvent( AWTEvent event)
```

The basic explicit event-processing method. When events are enabled they are first sent to this method which dispatches them to the appropriate specialized event-processing method, as follows:

```
protected void processComponentEvent( ComponentEvent event)
protected void processFocusEvent( FocusEvent event)
protected void processKeyEvent( KeyEvent event)
protected void processMouseEvent( MouseEvent event)
protected void processMouseMotionEvent( MouseEvent event)
```

Specialized event-processing methods, by default called from processEvent().

The effect of enabling key events upon a *NumericTextField* is to ensure that events generated by the user as they type are delivered to the instance. This is accomplished by passing KeyEvents to the instance's *processKeyEvent()* method. This method is also introduced into the hierarchy by the AWT Component class and the *processKeyEvent()* supplied by *NumericTextField* class overrides it.

Java will generate a KeyEvent whose ID attribute contains KEY_PRESSED when the user presses a key and one whose ID contains KEY_RELEASED when it is released. It will also generate a third KeyEvent whose ID attribute contains KEY_TYPED when a combination of key down followed by key up is detected and the key has a character associated with it. For example, if the user presses the ‘a’ key on the keyboard then all three KeyEvents will be generated, but a key such as SHIFT will only generate KEY_PRESSED and KEY_RELEASED events.

In this example no special consideration is needed to differentiate between these three different types of KeyEvent, and all three will be processed identically. The first part of the implementation of the *NumericTextField*’s *processKeyEvent()* method is as follows.

```
0019     protected void processKeyEvent( KeyEvent event) {
0020
0021     char    thisChar    = event.getKeyChar();
0022     int     thisCode    = event.getKeyCode();
0023     String  theText     = new String( this.getText());
0024
0025     boolean doIt        = false;
```

The method signature, on line 0019, indicates that an instance of the KeyEvent class will be passed as an argument to it and the first two steps extract the keyChar and keyCode attributes from the *event* into local variables. The keyChar attribute identifies the key pressed as a character, for example if the ‘A’ key were pressed while the SHIFT key was held down the keyChar attribute would contain the **char**acter value ‘A’.

Not all keys on the keyboard, for example the HOME key or the LEFT ARROW key have characters associated with them. To detect these keys the KeyEvent class declares a large number of manifest *v*irtual *k*ey constants, such as VK_HOME or VK_LEFT. There are also manifest values for the character keys, such as VK_A for the ‘A’ key; but there is no distinction made in keyCodes between lower- and upper-case characters. The keyCode attribute of KeyEvents contains the virtual key manifest value of the key used, as opposed to the character that it generates.

Object → EventObject → AWTEvent →
ComponentEvent → InputEvent → java.awt.event.KeyEvent

```
public KeyEvent( Component source, int id,      long when,
                 int modifiers,    int keyCode, int keyChar)
```

Constructs a KeyEvent originating from *source* for the reason expressed in *id* (PRESSED, RELEASED, or TYPED) at time *when*. The *modifiers* attribute is a bitwise combination of the masks supplied by the InputEvent class (SHIFT_MASK, ALT_MASK, CTRL_MASK and META_MASK). The *keyCode* attribute is a manifest value from the KeyEvent class (VK_SHIFT, VK_A, etc.). The *keyChar* attribute is the **char** value of the key and should be set to VK_UNDEFINED for keys that do not have an associated character.

```
public char getKeyChar()
public int  getKeyCode()
```

Enquiry methods supplied by KeyEvent.

```
public int     getModifiers()
public int     getWhen()
public boolean isAltDown()
public boolean isControlDown()
public boolean isShiftDown()
public boolean isMetaDown()
```

Enquiry methods supplied by InputEvent.

```
public void consume()
public int  isConsumed()
```

Originally introduced by AWTEvent and overridden in InputEvent. An event is initially not *consumed*. When its consume() method is called no further processing of the event will take place.

The first part of the *processKeyEvent()* method, on line 0023, obtains the text currently entered by calling the inherited TextField getText() method. The final local declaration is of a **boolean** flag called *doIt*. If the value of this flag remains **false** then it will indicate that the KeyEvent is not to be processed further; otherwise the event will be passed to the TextField processKeyEvent() method for further processing.

The next part of the *NumericTextField*'s *processKeyEvent()* method implements the essential logic from the STD, as follows.

```
0027        if ( ((thisChar >= '0') && (thisChar <= '9')) ){
0028           doIt = true;
0029
0030        } else if ( thisChar == '.') {
0031           if ( theText.indexOf( '.') == -1) {
0032              doIt = true;
0033           } // End if.
0034
0035        } else if ( thisChar == '+' ||
0036                    thisChar == '-' ){
0037            if ( theText.length() == 0 ) {
0038               doIt = true;
0039            } else if ( this.getCaretPosition() == 0   &&
0040                        theText.charAt( 0)      != '+' &&
0041                        theText.charAt( 0)      != '-' ){
0042              doIt = true;
0043           } // End if.
0044
0045        } else if ( thisCode == KeyEvent.VK_BACK_SPACE ||
0046                    thisCode == KeyEvent.VK_DELETE     ||
0047                    thisCode == KeyEvent.VK_END        ||
0048                    thisCode == KeyEvent.VK_HOME       ||
0049                    thisCode == KeyEvent.VK_LEFT       ||
0050                    thisCode == KeyEvent.VK_RIGHT      ||
0051                    thisCode == KeyEvent.VK_ENTER      ){
0052           doIt = true;
0053        } // end if.
```

The first part of the logic, on lines 0027 and 0028, confirms that the event should be further processed if the *event*'s keyChar attribute indicates that the user has pressed one of the digit ('0' to '9') keys. The next part of the logic, on lines 0030 to 0033, uses a compound condition to consider decimal points ('.'); the processing of the decimal point is confirmed only if line 0031 indicates that there is not a decimal point already in *theText*.

The next part of the logic, on lines 0035 to 0043, processes the sign (‘+’ and ‘–’) keys. These keys are always processed if, as indicated on line 0037, *theText* is empty. Otherwise a complex check, on lines 0039 to 0041, will confirm the action only if the insertion cursor, known as the *caret*, is at the start of the text and that the first position of *theText* does not already contain a sign character. The final part of the logic, on lines 0045 to 0053, processes the recognized editing (backspace and delete), navigation (end, home, left arrow and right arrow) and activation (enter) keys.

Having decided whether the event is or is not to be further processed, the final part of the *processKeyEvent()* method effects this as follows.

```
0055        if ( doIt) {
0056           super.processKeyEvent( event);
0057        } else {
0058           event.consume();
0059        } // End if doit.
0060     } // end processKeyEvent.
```

If the *doIt* flag indicates that further processing is appropriate then line 0056 passes on the *event*, received as an argument, to the **super**, TextField, processKeyEvent() method. The consequence of this is that the key press will be processed as normal and the state and appearance of the component will change as appropriate. Otherwise, further processing of the event is not appropriate and line 0058 calls the consume() method to set the *event*'s consume attribute, which will prevent any further processing of it by Java.

The *NumericTextField*'s implementation continues with the *getValue()* and *getFloatValue()* methods, as follows.

```
0067     public double getValue() {
0068        if ( this.getText().length() == 0 ) {
0069           this.setText( "0.0");
0070           return 0.0;
0071        } else {
0072           return ( Double.valueOf(
0073                          this.getText())).doubleValue();
0074        } // End if.
0075     } // End getValue.
0077
0078     public float getFloatValue() {
0079        return (float) this.getValue();
0080     } // End getFloatValue.
```

If the *NumericTextField* is empty a design decision has been taken to **return** the value 0.0 from the component and update the visible contents to indicate this. An alternative decision would have been to throw an exception at this point. As zero seems to be a very sensible default value for an empty *NumericTextField* this would seem to be an appropriate decision. A more sophisticated design might have included an attribute to decide between the two alternatives, with a default behavior to return zero.

The first part of *getValue()* on lines 0068 to 0070 returns zero after setting the text in the area to “0.0” if the area is empty. Otherwise, lines 0072 to 0073 convert the text contained in the area into a **double** value and return it. The *getFloatValue()* method is implemented as a call of *getValue()*, downcasting the value obtained from a **double** to a **float**. If the **double** value is outside the range of possible **float** values this cast will not throw an exception and a decision has been taken not to throw an exception from the method in accord with the behavior of the Java language. The remaining two methods are implemented as follows.

```
0080     public void setValue( double newValue) {
0081        super.setText( ( new Double( newValue)).toString());
0082     } // End setValue.
0083
0084
0085     public void setText( String newValue) {
0086
0087     double value = 0.0;
0088
0089        try {
0090          value = (new Double( newValue)).doubleValue();
0091          super.setText( newValue);
0092        } catch ( NumberFormatException exception) {
0093          // Do nothing.
0094        } // End try/catch.
0095     } // End setText.
0096  } // End class NumericTextField.
```

The *setValue()* method installs the **double** value passed as an argument into the text area by calling the inherited *setText()* method, after converting it, on line 0081, to a String representation. The *setText()* method attempts, on line 0090, to convert the String supplied as an argument into a **double** value. If this attempt does not throw an exception the String supplied as an argument is installed into the input area with a call of the **super**, JTextField, setText() method. If the String supplied to *setText()* cannot be interpreted as a floating-point value the attempt to convert it will throw a NumberFormatException which, when it is caught by the handler on line 0093, does nothing.

With this implementation there is no indication to the user that any key presses that cannot be used to compose a floating-point value have not been attended to. A small addition to the code, as follows, would cause a beep to be emitted by the terminal in these situations.

```
0055        if ( doIt) {
0056           super.processKeyEvent( event);
0057        } else {
0058           if ( event.getID() == KeyEvent.KEY_TYPED) &&
0059                thisChar != KeyEvent.CHAR_UNDEFINED) ){
0060              this.getToolkit().beep();
0061           } // End if.
0062           event.consume();
0063        } // End if doit.
0064     } // end processKeyEvent.
```

The condition on line 0058 and 0059 ensures that only a single bleep is caused when a key is pressed and released and prevents presses of the keys not associated with a particular character to cause a beep to be emitted. On most systems the keys on the keyboard will send a series of events if they are held down; this can be useful, for example, to a user who wishes to fill in a line with space characters. However, if the user were entering a CONTROL key combination then as the CONTROL key is held down a sequence of events would be dispatched before the combination key was pressed, and this would cause a series of beeps to be emitted. As the CONTROL key is not associated with a particular typed character the keyChar attribute of the KeyEvents that it dispatches will have the value CHAR_UNDEFINED. Additionally, when a key which has a valid character associated with it is used it generates three KeyEvents (pressed, released and typed), the first part of the condition, on line 0058, restricts bleeping to the key typed event.

If a beep is to be emitted line 0060 obtains the toolkit associated with the component. This is effectively a mechanism for accessing the actual hardware that the artifact is running upon. However, the beep() method, which causes the speaker to emit a single beep, is one of the very few Toolkit methods that should be used directly.

> *The use of beeps is contentious as many UI designers (including the author) hold the view that they are redundant and may cause noise pollution that will necessarily stress the user. Others, however, hold the view that they are a useful way to alert the user to a problem, assuming that the environment they are working in is quiet enough for them to hear it! The swing implementation of JTextField uses beeps in some situations (for example attempting to paste text into the field when the clipboard is empty), so there is an additional argument that NumericTextFields should also do so when appropriate.*

As with all other components this class should be mounted within a demonstration harness and shown to be operating correctly before it is relied upon and used as a part of a more complex interface. A suitable harness will be presented after the *IntegerTextField* class has been designed and implemented.

4.5 The *IntegerTextField*: design and implementation

The class diagram for the *IntegerTextField* class is given in Figure 4.5. One major difference between the *IntegerTextField* class and the *NumericTextField* class is that this class supports two attributes, *minimumValue* and *maximumValue*, which delineate the range of values that the component will allow the user to enter. The *IntegerTextField* class is extended from the *NumericTextField* class and also manages its own internal event handling, by means of a protected *processKeyEvent()* method.

The default constructor will allow the full range of **long** values to be entered. The first alternative constructor requires a maximum value to be specified and will allow values between zero and the value supplied to be entered. The second alternative constructor allows a range of values between its *minimum* and *maximum* arguments to be entered. The two instance inquiry methods, *getMinimum()* and *getMaximum()*, provide a mechanism for the limits of the acceptable range to be obtained.

The remaining methods are comparable to the corresponding methods from the *NumericTextField* class. The *getLong()* method will return the value from the input area, or provide it with either the value zero or the *minimumValue*, depending on which is the higher, if it is currently empty. The *getInt()* method is supplied as a convenience, downcasting the **long** value without throwing an exception if accuracy is lost. The *setValue()* and *setText()* methods allow the value shown by the component to be specified; if the string supplied to the *setText()* method cannot be interpreted as a **long** value the state of the component is not changed and no exception is thrown.

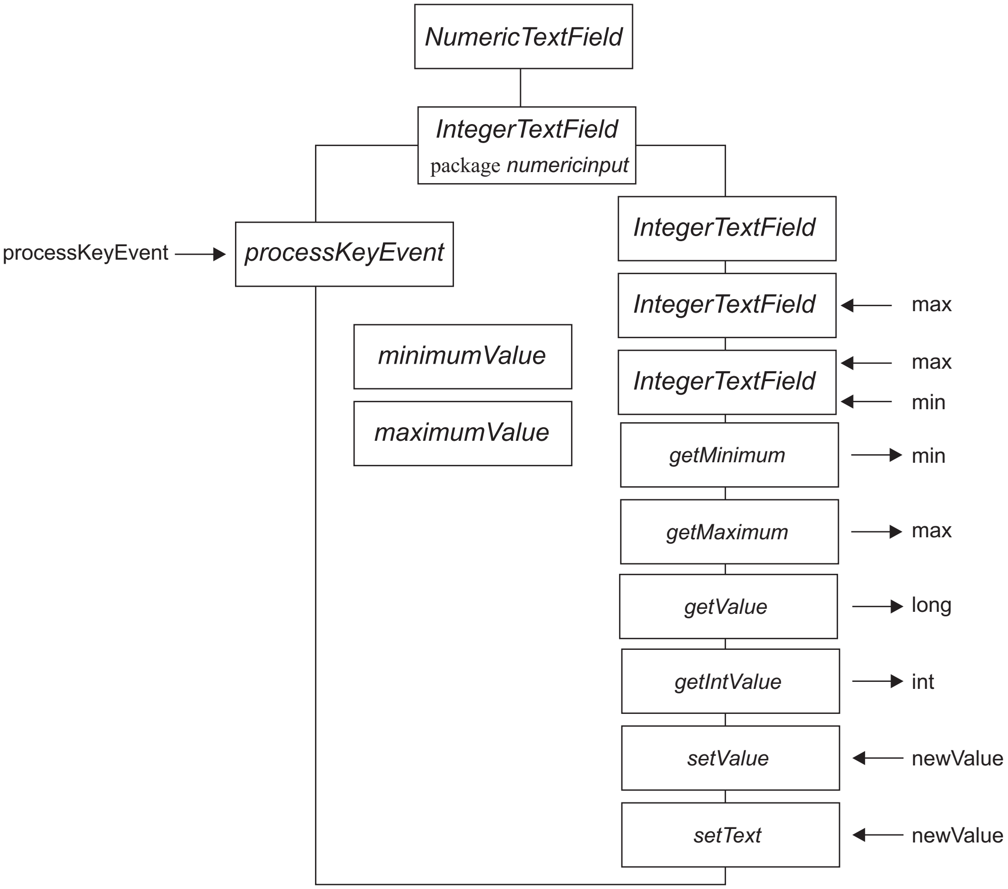


Figure 4.5 *IntegerTextField* class diagram.

The behavior of an *IntegerTextField* instance differs from that of a *NumericTextField* instance; the revised behavior is shown in Figure 4.6.

There are three significant differences between this STD and that for the *NumericTextField*. The first difference is that the period key is not responded to, as an integer value cannot possibly contain a decimal point. The second change is that a sign character is only allowed if the component supports negative values. Perhaps the most significant change is the handling of the digit key presses. In this version digit characters are only processed if the numeric value represented by the area's contents after the digit character has been added to it is within the range of acceptable values; otherwise the digit is ignored.

The implementation of this design, as far as the end of the constructors, is as follows.

```
package numericinput;

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

```

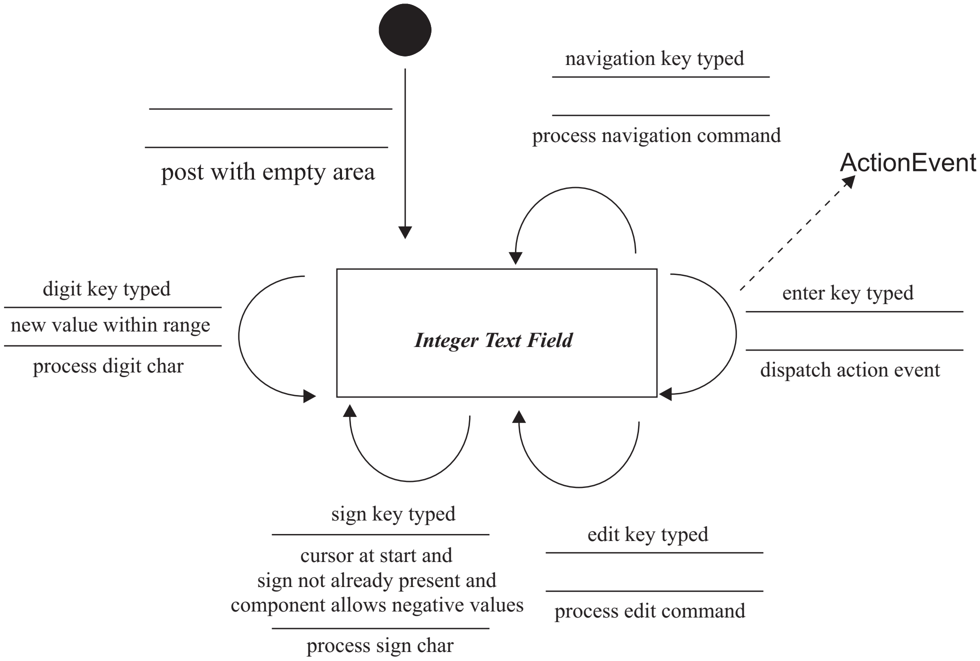


Figure 4.6 *IntegerTextField* state transition diagram.

```
0016
0017  public class IntegerTextField extends NumericTextField {
0018
0019  protected static final long ZERO = 0L;
0020
0021  private long minimumValue = ZERO;
0022  private long maximumValue = Long.MAX_VALUE;
0023
0024
0025     public IntegerTextField() {
0026        this( Long.MIN_VALUE, Long.MAX_VALUE);
0027     } // End IntegerTextField default constructor.
0028
0029     public IntegerTextField( long maximum) {
0030        this( ZERO, maximum);
0031     } // End IntegerTextField alternative constructor.
0032
0033     public IntegerTextField( long minimum, long maximum) {
0034        super();
0035        minimumValue = minimum;
0036        maximumValue = maximum;
0037        this.enableEvents( AWTEvent.KEY_EVENT_MASK);
0038     } // End IntegerTextField constructor.
```

Following the package declaration, the importations and the class declaration, line 0019 declares *ZERO* as a manifest **long** value. The two instance attributes are declared on lines 0021 and 0022. The default constructor, on lines 0025 to 0027, indirects to the second alternative constructor passing as arguments the minimum and maximum **long** values, supplied as manifest constants by the Long class. The first alternative constructor, on lines 0029 to 0031, also indirects to the second alternative constructor, passing as arguments the manifest *ZERO* value and the value of the single maximum argument.

The second alternative constructor actually constructs the instance, commencing on line 0034 by calling the **super**, *NumericTextField*, default constructor. It then stores the values of the two arguments in their corresponding instance attributes and concludes by enabling KeyEvents upon itself. This last step is strictly not necessary, as the call of the **super** constructor has already effected this. However, a decision has been taken to include this step explicitly in this class to assist developers reading this source code in working out exactly how it is operating. There is no check made in the constructor that the value of the *maximum* argument is greater than that of the *minimum* argument, a consideration that will be left to an end of chapter exercise.

The declaration of the class continues with the first part of the *processKeyEvent()* as follows.

```
0041     protected void processKeyEvent( KeyEvent event) {
0042
0043     char    thisChar      = event.getKeyChar();
0044     String  theText       = new String( this.getText());
0045     int     caretPosition = this.getCaretPosition();
0046
0047     String  firstPart      = null;
0048     String  lastPart       = null;
0049     long    candidateValue = 0;
0050
0051     boolean doIt           = true;
0052
0053        if ( ((thisChar >= '0') && (thisChar <= '9')) ){
0054
0055           firstPart = theText.substring( 0, caretPosition);
0056           if ( caretPosition < theText.length()) {
0057              lastPart = theText.substring( caretPosition);
0058           } else {
0059              lastPart = new String( "");
0060           } // End if.
0061
0062           try {
0063              candidateValue = Long.valueOf(
0064                 firstPart + thisChar + lastPart).longValue();
0065
0066              if ( candidateValue < minimumValue ||
0067                   candidateValue > maximumValue ){
0068                 throw new NumberFormatException();
0069              } // End if.
0070           } catch ( NumberFormatException exception) {
0071              doIt = false;
0072           } // End try/catch.
```

As with the corresponding method in the *NumericTextField* class this method will be called every time the user presses or releases a key, or types a character at the keyboard. It can either pass the event onwards to its **super** *processKeyEvent()* if it is to be attended to or can consume the event if it is to be ignored.

The local declarations on lines 0043 to 0044 store the character, if any, from the KeyEvent, obtain the existing text from the input area and make a note of the *caretPosition*. The use made of the two Strings declared on lines 0047 and 0048 will be explained shortly. The *candidateValue*, declared on line 0049, will be used to store the possible new value of the component if the current key is processed. This will be determined by the value of the *doIt* flag, declared on line 0051, which is set **true** to indicate that, by default, the event should be processed further.

The first branch of the method's **if** structure, controlled by the condition on line 0053, is concerned with processing digit characters. Line 0055 extracts the *firstPart*, as far as the insertion caret, of the existing contents of the input area. Lines 0056 to 0060 then extract the remaining part into *lastPart* or store an empty String in *lastPart* if the caret is at the end of the input area. On lines 0063 and 0064 the digit character is inserted between the *firstPart* and the *lastPart* and the entire combined String is converted into a **long** value which is stored in *candidateValue*. This method may throw a NumberFormatException (although never in this circumstance) and so must be contained within a **try/catch** structure. Having obtained the *candidateValue*, that is the new value of the component if the digit is inserted into its existing contents, it is compared with the range of allowed values in the inner **if** condition on lines 0066 and 0067. Should the possible new value be outside the allowed range an explicit NumberFormatException is thrown on line 0068. If an exception is thrown, either implicitly or explicitly, it indicates that the key press is to be ignored and the *doIt* flag is set **false**, in the exception handler on line 0071. The implementation of the *processKeyEvent()* concludes as follows.

```
0074        } else if ( thisChar == '.') {
0075           doIt = false;
0076
0077        } else if ( (thisChar == '-' ||
0078                     thisChar == '+' )  ){
0079           if ( minimumValue >=  0) {
0080              doIt = false;
0081           } else if ( theText.length() > 0 &&
0082                       ( theText.charAt( 0) == '-' ||
0083                         theText.charAt( 0) == '+' )  ){
0084              doIt = false;
0085           } // End if.
0086        } // End if.
0087
0088        if ( doIt) {
0089           super.processKeyEvent( event);
0090        } else {
0091           if ( (event.getID() == KeyEvent.KEY_TYPED) &&
0092                (thisChar == KeyEvent.CHAR_UNDEFINED) ){
0092             this.getToolkit().beep();
0093           } // End if.
0094           event.consume();
0095        } // End if.
0096     } // End processKeyEvent.
```

The next branch of the outer **if** structure, on lines 0074 and 0075, deals with decimal points by setting the *doIt* flag to ignore them. The next branch, commencing on lines 0077 and 0078, considers the sign characters '–' and '+'. On lines 0079 and 0080 these characters are always ignored if the minimum value is non-negative. Otherwise they are denied on lines 0081 to 0083 if there is already a sign character at the start of the input area. Ensuring that the sign is inserted at the left of the field is left to the *NumericInput processKeyEvent()* method. The end **if** on line 0085 concludes the consideration of characters in this component; all other characters, including those nominated explicitly on the STD in Figure 4.6, will be processed by the *NumericTextField processKeyEvent()* method.

The concluding part of the *IntegerTextField processKeyEvent()* method, on lines 0088 to 0096, passes the event to the **super**, *NumericTextField*, *processKeyEvent()* method if the *doIt* flag indicates that the previous part of this method has decided that the character should not be ignored. Otherwise, lines 0091 to 0094 emit a bleep, if required, and *consume()* the event, as before.

The next two methods shown on the class diagram in Figure 4.5 are straightforward inquiry methods and are implemented, without comment, as follows.

```
0099     public long getMaximum() {
0100        return maximumValue;
0101     } // End getMaximum.
0102
0103     public long getMinimum() {
0104        return minimumValue;
0105     } // End getMinimum.
```

The next two methods, *getValue()* and *getIntValue()* are comparable to the corresponding *NumericTextField getValue()* and *getFloatValue()* methods, and are implemented as follows.

```
0108     public long getValue() {
0109        if ( this.getText().length() == 0 ) {
0110           if ( minimumValue > ZERO) {
0111              super.setText((new Long(
0112                                minimumValue)).toString());
0113              return minimumValue;
0114           } else {
0115             super.setText( "0");
0116             return ZERO;
0117           } // End if.
0118        } else {
0119           return ( Long.valueOf( this.getText())).longValue();
0120        } // End if.
0121     } // End getValue.
0122
0123     public int getIntValue() {
0124        return (int) this.getValue();
0125     } // End getIntValue.
```

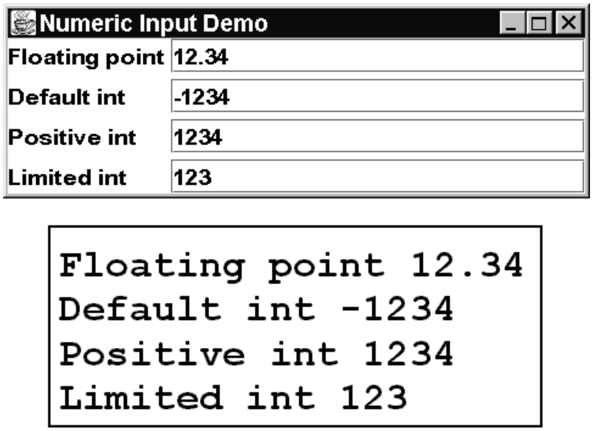
If the *getValue()* method is called when the input area is empty it will return either ZERO or the *minimumValue*, placing a String representation of the returned value into the input area to indicate this, depending upon which of the two values is the higher. That is, if *minimumValue* is less than zero it will return zero, otherwise it will return the *minimumValue*. If the input area is not empty the *getValue()* method returns the value contained in the input area. The *getIntValue()* calls *getValue()* and downcasts the **long** value obtained to an **int**. The two remaining methods conclude the class's definition, as follows.

```
0127     public void setValue( long newValue) {
0128        if ( newvalue >= minimumValue &&
0129             newvalue <= minimumValue ){
0130           super.setText( ( new Long( newValue)).toString());
0131        } // End if.
0132     } // End setValue.
0133
0134     public void setText( String newValue) {
0135
0136     long value = 0;
0137
0138        try {
0139          value = (new Long( newValue)).longValue();
0140          this.setValue( value);
0141        } catch ( NumberFormatException exception) {
0142          // Do nothing.
0143        } // End try/catch.
0144     } // End setText.
0145
0146  } // End class IntegerTextField.
```

The *setValue()* method will install a String representation of the *newValue* supplied as an argument into the input area only if the *newValue* is within the allowable range. If the value is outside the allowable range the component will be left unchanged. The call of **super**.setText(), on line 0130, is a call of the *NumericTextField setText()* method. As described above, this method will test the String value supplied to ensure that it can be interpreted as a **double** value and, if so, will install the String exactly as supplied in its argument into the input area by calling its own **super**.*setText*() method, which is the JTextArea setText() method. A **long** value can always be interpreted as a **double**, as there is no requirement that it conclude with ".0". Hence there is no danger that the **super** method will refuse to install the value, nor is there any danger that the *NumericTextField* will append ".0" to the **long** String supplied before displaying it.

The *setText()* method on lines 0134 to 0144 commences, within a **try/catch** structure, to convert the String supplied into a **long** value. If this attempt does not throw an exception the value obtained is passed on to the *setValue()* method, which has just been described. Otherwise, if the exception is thrown, the component is left unchanged.

This concludes the description of the *IntegerTextField* class, and although, like the *NumericTextField* class, it might appear that it has been robustly engineered, it still contains a number of insecurities that may cause it to fail under various conditions. The engineering, as presented in this chapter, is just sufficient to demonstrate the main considerations required to build the specialized capability. In order to produce a production-ready version of this component additional work will be needed. Some indications of the inadequacy of these implementations will be presented at the end of the chapter and some of the additional work will be suggested as end of chapter exercises.



Floating point 12.34

Default int -1234

Positive int 1234

Limited int 123

```
Floating point 12.34
Default int -1234
Positive int 1234
Limited int 123
```

Figure 4.7 *NumericInputDemo* and terminal output.

4.6 The *NumericInputDemo* class

Figure 4.7 illustrates the appearance of the *NumericInputDemo* class. Its purpose is to provide an initial indication that the two classes described so far in this chapter appear to be working. It contains, on the right-hand side from top to bottom, an instance of the *NumericTextField* class, an instance of the *IntegerTextField* class constructed with the default constructor, an *IntegerTextField* constructed so as to accept only positive values and, finally, an *IntegerTextField* constructed so as to only accept values between –999 and +999.

Also shown in Figure 4.7 is the terminal output produced by the demonstration client when the ENTER key was pressed in each of the components in turn while they contained the values indicated. This output was produced from the *NumericInputDemo*'s *actionPerformed()* method in response to *ActionEvent*s received from the components. The implementation of this class, as far as the end of its *init()* declaration, is as follows.

```
package numericinput;

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


public class NumericInputDemo extends    JApplet
                              implements ActionListener {

   public void init(){

   JPanel           labelPanel      = null;
   JPanel           inputPanel      = null;

   JLabel           floatLabel      = null;
   NumericTextField floatingPoint   = null;
   JLabel           defaultLabel    = null;
```

```
0029     IntegerTextField defaultInteger  = null;
0030     JLabel           positiveLabel   = null;
0031     IntegerTextField positiveInteger = null;
0032     JLabel           limitedLabel    = null;
0033     IntegerTextField limitedInteger  = null;
```

There are two intermediate panels, declared on lines 0023 and 0024, and four pairs of a JLabel instance and a *numericinput* component, declared on lines 0026 to 0033. The *init()* method continues as follows.

```
0035        this.setResources();
0036
0037        floatLabel      = new JLabel( "Floating point ");
0038        floatingPoint   = new NumericTextField();
0039        floatingPoint.setName(   "Floating point");
0040        floatingPoint.addActionListener( this);
0041
0042        defaultLabel    =  new JLabel( "Default int");
0043        defaultInteger  =  new IntegerTextField();
0044        defaultInteger.setName(  "Default int");
0045        defaultInteger.addActionListener(  this);
0046
0047        positiveLabel   = new JLabel( "Positive int");
0048        positiveInteger = new IntegerTextField( Long.MAX_VALUE);
0049        positiveInteger.setName( "Positive int");
0050        positiveInteger.addActionListener( this);
0051
0052        limitedLabel    =  new JLabel( "Limited int");
0053        limitedInteger  =  new IntegerTextField( -999, +999);
0054        limitedInteger.setName(  "Limited int");
0055        limitedInteger.addActionListener(  this);
```

Having set the resources for this artifact, each pair of JLabel and *numericInput* instances is constructed in turn. Each input component is given a name and has **this** instance of *NumericInputDemo* being initialized registered as its actionListener. The *init()* method concludes as follows.

```
0057        labelPanel = new JPanel( new GridLayout( 4, 1, 5, 5));
0058        inputPanel = new JPanel( new GridLayout( 4, 1, 5, 5));
0059
0060        labelPanel.add( floatLabel);
0061        labelPanel.add( defaultLabel);
0062        labelPanel.add( positiveLabel);
0063        labelPanel.add( limitedLabel);
0064
0065        inputPanel.add( floatingPoint);
0066        inputPanel.add( defaultInteger);
0067        inputPanel.add( positiveInteger);
0068        inputPanel.add( limitedInteger);
0069
0070        this.getContentPane().add(
0071                                     labelPanel, BorderLayout.WEST);
0072        this.getContentPane().add(
0073                               inputPanel, BorderLayout.CENTER);
0074     } // End init.
```

Having constructed and configured the visible parts of the artifact, this part of the *init()* method constructs the intermediate panels and assembles the interface. When the interface is presented to the user the input behavior of the *numericinput* components can be investigated and verified against their STDs. When the ENTER key is pressed an *actionEvent* will be dispatched to the *NumericInputDemo actionPerformed()* method, which is implemented as follows.

```
0077     public void actionPerformed( ActionEvent event) {
0078
0079     Component theSource = (Component) event.getSource();
0080     String    theName   = theSource.getName();
0081
0082        System.out.print( theName + " ");
0083
0084        if ( theName.equals( "Floating point")) {
0085           System.out.println( ((NumericTextField) theSource).
0086                                         getValue());
0087        } else {
0088           System.out.println( ((IntegerTextField) theSource).
0089                                           getValue());
0090        } // End if.
0091     } // End actionPerformed.
```

The identity of *theSource* of the *event*, as a Component, is obtained on line 0079 and, on line 0080, *theName* of the Component is obtained and subsequently output onto the terminal, on line 0082. The remainder of the method decides, using *theName*, whether the *event* originated from a *NumericTextField* or *IntegerTextField* instance. Having decided this, *theSource* can be cast to the appropriate type and either the *getValue()* or *getIntValue()* method called to obtain the value from the component. This value is immediately sent to the terminal to complete the line of output, as illustrated in Figure 4.7.

4.7 The *SpinBox*: design overview

A *SpinBox* consists of an integer input area and a pair of arrowed controls which can be used to adjust the value upwards or downwards. It has become a standard user interface control for the input of small integer values, such as the required size of a font, but has not been supplied by the JFC. Figure 4.8 contains a state transition diagram for a *SpinBox*'s behavior.

Unlike the *NumericTextField* and the *IntegerTextField*, a *SpinBox* is a composite component; that is, a component that contains other components. In this example the contained components are all examples of specialized extensions of a JFC component, but this need not always be the case. In these cases the implementation of the component's essential behavior will have to make use of the event source/listener mechanism to pass messages between the contained components. When designing and building composite components some care should be taken to differentiate between internal and external event dispatching.

The initial transition posts the *SpinBox* with its default value displayed, unlike an *IntegerTextArea*, which is initially shown empty. The default *static* state contains all of the transitions shown in Figure 4.6 for the *IntegerTextField* component, including the

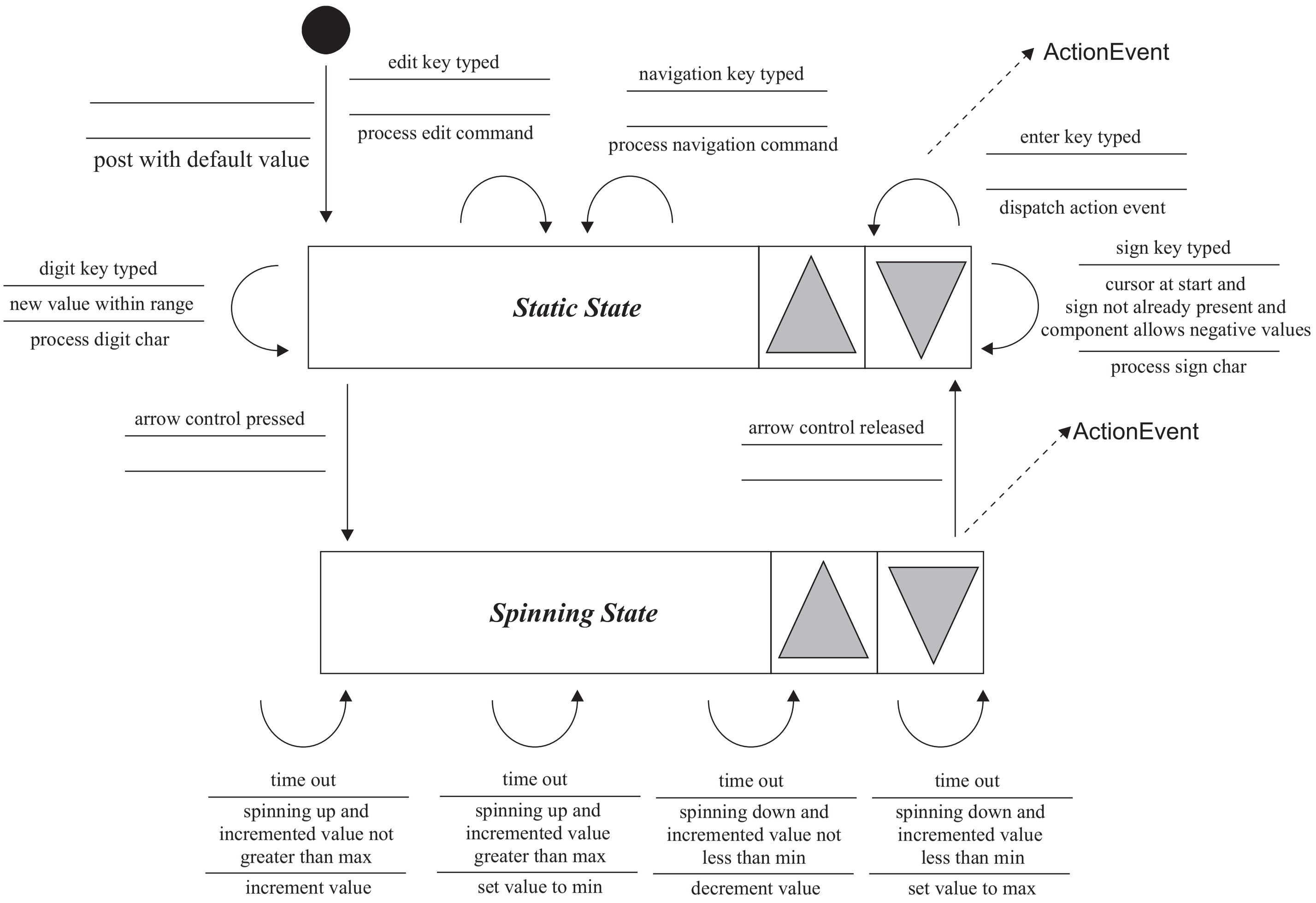


Figure 4.8 *SpinBox* state transition diagram.

dispatch of an event when the `<ENTER>` key is pressed. This indicates that the user can interact with the text input area, at the left hand side of the component, exactly as if it were an *IntegerTextField*.

There is one additional transition leading from the default *static* state to the *spinning* state, which is always followed when the mouse button is pressed down on either of the arrow controls. The four transitions at the bottom of this state are all occasioned by the occurrence of a time out event; this event will occur when the component first arrives in the *spinning* state and periodically thereafter. The four transitions form a mutually inclusive set; that is, every time a time out event occurs, one, and only one, of the transitions will be followed. The first transition shows that if the component is spinning up (that is, the up arrow control was pressed) and incrementing the value in the text input area will not take it beyond the maximum, the value shown in the area is incremented. Otherwise, if incrementing the value would take it beyond the maximum, the second transition shows that the minimum value will be placed into the input area.

The two remaining transitions at the bottom of the state are comparable, but deal with spinning down and are concerned with spinning around the minimum value. The other transition originating from the *spinning* state, shown at the top right of the state, is followed when the arrow control is released and leads back to the *static* state; an event is also dispatched when this transition is followed.

Hence if the user presses and immediately releases the up arrow control the value shown in the input area will increment or spin around to the minimum, and an event will be dispatched to inform the component's listeners that the value has changed. Alternatively, if the user presses and holds the up arrow control down, the value in the input area will spin continually between the minimum and the maximum, with an event being dispatched only when the user releases the up arrow control.

An alternative version of this state transition diagram may have three different states: the *static* state, a *spinning up* state and a *spinning down* state. Each of the spinning states would have a transition leading to it from the *static* state, each would have two transitions returning to itself and each would have a transition leading back to the *static* state with an event dispatched from it. Although this design would be equivalent, it would contain three rather than two states and would duplicate the event-dispatching transition. Accordingly, the design as presented is favored, as it has fewer transitions and states.

> *This is a common design heuristic: when two equivalent designs are being considered the simplest should always be favored. The heuristic can be remembered as Keep It Sufficiently Simple (KISS), as complexity adds to production and maintenance costs.*

The *SpinBox* is a composite component, that is a component which itself contains further components. Figure 4.9 contains an instance diagram for a single *SpinBox* instance.

The instance diagram shows that a *SpinBox* instance contains an *IntegerTextArea* instance, called *inputArea*, and two *ArrowBox* instances, called *upArrow* and *downArrow*. The diagram also shows that the *SpinBox* instance listens to the events which are dispatched from each of these three included components. Each *ArrowBox* contains its own Timer instance, called *timer* and that they, in turn listen, to the events dispatched from these objects.

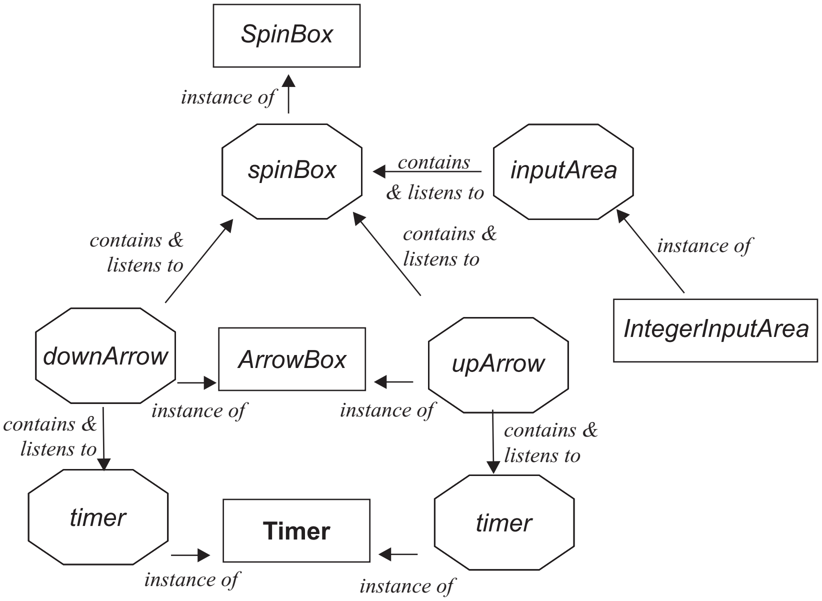


Figure 4.9 *SpinBox* instance diagram.

The Timer class being used here is the JFC Timer class not the *stopwatch Timer* class as developed in Chapter 1. The *stopwatch.Timer* class was developed so as to not only inform its listeners that a period of time had elapsed, but also to inform them of how many periods of time had elapsed since the timer was started or last reset. There is no requirement in this component for the number of periods to be communicated so the simpler JFC Timer class is more appropriate.

It might seem wasteful for each *ArrowButton* to have its own Timer instance, and a different design might allow the two *ArrowButton*s to share a single Timer instance. However, this would compromise the integrity of an individual *ArrowButton* instance, making it dependent upon another object to supply it with timing capability, and would cause problems if a large number of *ArrowButton*s were required in a particular artifact.

> *Objects are cheap! A design should never be compromised to reduce the number of object instances required. The costs, to the developer, of having to work with a compromised design far outweigh the costs, to the machine, of having to create the additional objects.*

The *ArrowBox* class is not supplied by the JFC and is implemented as a non-public class in the *numericinput* package.

4.8 The *ArrowBox*: design and implementation

The *ArrowBox* class diagram is given in Figure 4.10 it shows that the *ArrowBox* class extends the JComponent class, is a member of the *numericinput* package of classes and implements the ActionListener interface in order to be able to listen to the ActionEvents

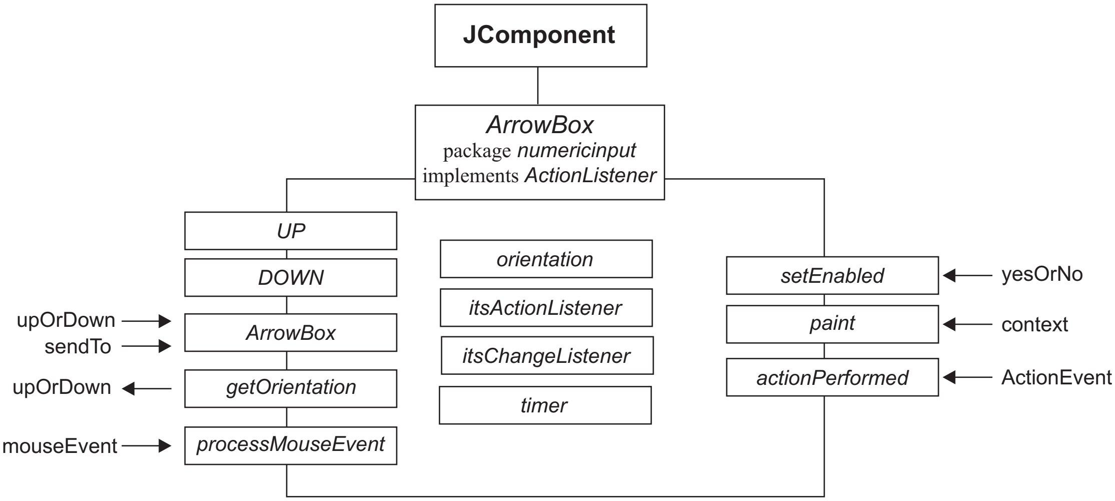


Figure 4.10 *ArrowBox* class diagram.

dispatched to it from its encapsulated *timer* instance. The other instance attributes are its *orientation*, either *UP* or *DOWN*, and the identity of the object to which it is to dispatch ChangeEvents and ActionEvents.

The **protected** constructor requires the orientation of the arrow to be specified, making use of the **protected** manifest constants, and the identity of the object to which its events are to be *sentTo*. The first **protected** method is the *getOrientation()* inquiry method. The other **protected** method is *processMouseEvent()*, taking a MouseEvent as an argument, and is comparable to the *processKeyEvent()* method in the *NumericTextField* and *IntegerTextField* classes. This method will be called by the Java environment to inform the component of a mouse event within the component's boundaries and will be used to implement the *ArrowBox*es internal behavior, as will be described below.

The three **public** methods override existing **public** methods and so cannot be declared **protected**. The *setEnabled()* method will set the sensitivity of the component and, as expected, a disabled component will present itself in a grayed-out manner and will not respond to the user's actions. The *paint()* method is required for the component to display itself in an appropriate state and the *actionPerformed()* method will receive events from the *timer* instance attribute.

Before the description of the *ArrowBox*es' implementation can be given the behavior of an *ArrowBox* will have to be designed. The state transition diagram for an *ArrowBox* is given in Figure 4.11.

The diagram shows that the component first becomes visible in the *inactive* state, with the border and arrow shown in an intermediate color. When the mouse pointer enters the component it transits to the *mouse entered* state, where the border is shown in a darker color. From this state, if the mouse pointer leaves the component it transits back to the *inactive* state; alternatively, if the mouse button is pressed down the component transits to the *button down* state, where both the border and the arrow are shown in a dark color. While the component remains in this state *time out* events will cause the component to dispatch ChangeEvents to its listener. The component will remain in this state even if the user moves the mouse pointer outside the extent of the component. When the user

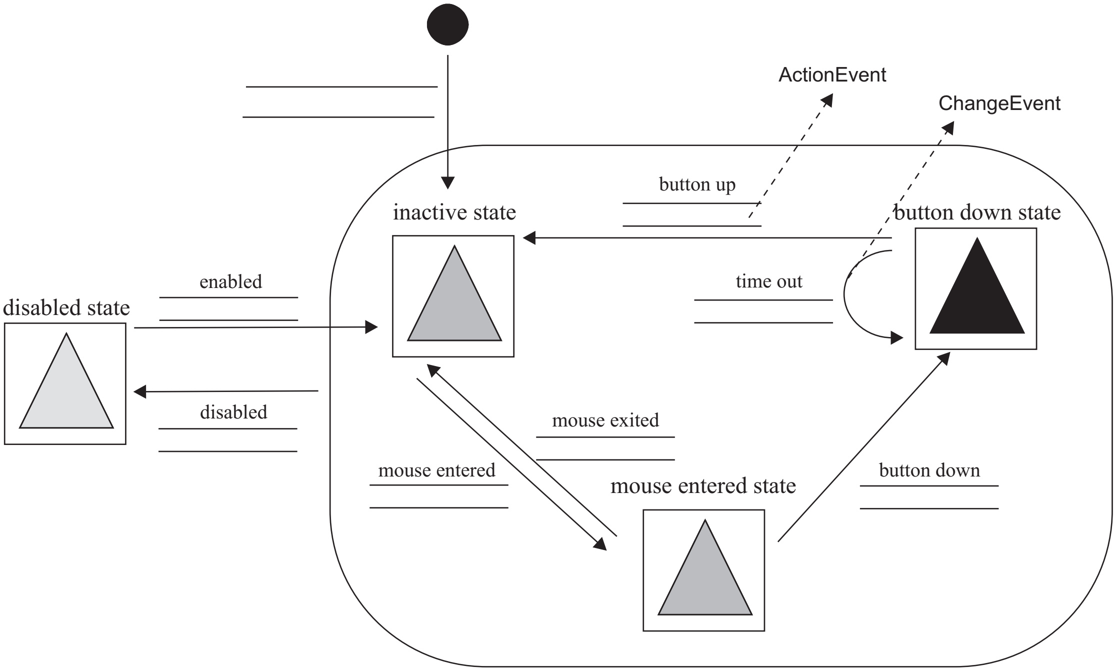


Figure 4.11 *ArrowBox* state transition diagram.

releases the mouse button in the *button down* state the component transits to the *inactive* state, dispatching an *ActionEvent* as it does so.

From any of these states, if the component becomes disabled it transits to the *disabled* state, where it is shown with a lighter border and arrow. When the component becomes enabled again it always transits to the *inactive* state, irrespective of the position of the mouse pointer or state of the mouse button. The implementation of this design commences as follows.

```
package numericinput;

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

class ArrowBox extends    JComponent
               implements ActionListener {

private final static int DELAY = 250;

private final static int DISABLED      = 0;
private final static int INACTIVE      = 1;
private final static int MOUSE_ENTERED = 2;
private final static int BUTTON_DOWN   = 3;
private int state = INACTIVE;

```

```
0028  protected final static int UP   = 0;
0029  protected final static int DOWN = 1;
0030
0031  private int            orientation       = UP;
0032  private ChangeListener itsChangeListener = null;
0033  private ActionListener itsActionListener = null;
0034  private Timer          timer             = null;
```

Following the **package** declaration, the required importations and non-**public** class declaration, line 0020 declares a **private** constant called *DELAY*, which will determine the delay between the dispatch of ChangeEvents when the component is in the *button down* state. Lines 0022 to 0025 declare four **private** manifest constants to define the possible states of the component and, on line 0026, a **private** attribute, called *state*, to record the state of the component. The two **protected** manifest values defining the direction of the arrow are declared on lines 0028 and 0029, followed by the *orientation* attribute on line 0031. The attribute declarations conclude, on lines 0032 to 0034, with the three remaining instance attributes. The implementation of the constructor is as follows.

```
0037     protected ArrowBox( int       upOrDown,
0038                         Component sendTo) {
0039        super();
0040        orientation = upOrDown;
0041        itsChangeListener = (ChangeListener) sendTo;
0042        itsActionListener = (ActionListener) sendTo;
0043        timer = new Timer( DELAY, this);
0044        timer.setInitialDelay( 0);
0045        this.enableEvents( AWTEvent.MOUSE_EVENT_MASK);
0046     } // End ArrowBox constructor.
```

The constructor commences, on line 0039, by calling the **super**, JComponent, default constructor and continues by storing the *upOrDown* argument in the *orientation* attribute. The second constructor argument, *sendTo*, is the identity of the object to which both ChangeEvents and ActionEvents are to be dispatched, and its identity is stored, with suitable casting, in the two *listener* attributes on lines 0041 and 0042.

The next step, on line 0043, constructs the Timer instance; the arguments to the constructor indicate the *DELAY*, in milliseconds, between the dispatch of events and the identity, **this**, of the object to send the events to. The construction of a Timer instance prepares it to start dispatching events, but no events will be dispatched until the *timer* is started. The setting of the *timer*'s initialDelay attribute to zero, on line 0044, ensures that the *timer* will dispatch an initial event as soon as it is started rather than wait *DELAY* milliseconds.

Object → javax.swing.Timer

```
public Timer( int delay, ActionListener listener)
```

Constructs an instance that will dispatch ActionEvents to its listener every *delay* milliseconds, when it has been started.

```
public void start()
public void stop()
public void restart()
```

Methods to start and stop the dispatch of events; restart stops and immediately starts the timer.

```
public void    setDelay( int newDelay)
public int     getDelay()
public void    setInitialDelay( int newInitialDelay)
public int     getInitialDelay()
public void    setRepeats( boolean yesOrNo)
public boolean isRepeats()
```

The delay attribute determines the time, in milliseconds, between the dispatch of events. The initialDelay attribute defaults to delay and determines the dispatch of the first event after the timer has been started. The repeats attribute, default **true**, determines whether a single event or a series of events should be dispatched.

The final step in the constructor enables MouseEvents upon this component. As with the enabling of KeyEvents earlier in this chapter, this will inform Java that the component is interested in receiving MouseEvents when the user interacts with the component. These events will be sent as arguments to the *processMouseEvent()* method, which will be described below. The implementation continues with the simple *orientation* inquiry method, as follows.

```
0048     protected int getOrientation(){
0049        return orientation;
0050     } // End getOrientation.
```

The next method is the *setEnabled()* method, as follows.

```
0053     public void setEnabled( boolean yesOrNo) {
0054        super.setEnabled( yesOrNo);
0055        if ( yesOrNo) {
0056           timer.stop();
0057           state = DISABLED;
0058        } else {
0059           state = INACTIVE;
0060        } // End if.
0061        this.repaint();
0062     } // End setEnabled.
```

The method commences, on line 0054, by calling the **super**, JComponent, *setEnabled()* method passing on the argument *yesOrNo*. Having done this it sets its *state* to *DISABLED* if the argument is **true** or *INACTIVE* if it is **false**, in accord with the state transition diagram in Figure 4.11. If the ArrowBox is being disabled it also, on line 0056, stops the *timer* to prevent it from continuing to dispatch events if it is being disabled from the *button down* state. This may mean that a *timer* that has not been started is being stopped, but as this does not cause any problems it is not regarded as a fault. In order that the consequences of the state change are made visible to the user the last step of the method is to repaint() the component. The *processMouseEvent()* method is implemented as follows.

```
0065     protected void processMouseEvent( MouseEvent event) {
0066
0067       if ( this.isEnabled()) {
```

```
0068          switch ( event.getID()) {
0069
0070          case MouseEvent.MOUSE_ENTERED:
0071             state = MOUSE_ENTERED;
0072             break;
0073
0074          case MouseEvent.MOUSE_EXITED:
0075             state = INACTIVE;
0076             break;
0077
0078          case MouseEvent.MOUSE_PRESSED:
0079             state = BUTTON_DOWN;
0080             timer.start();
0081             break;
0082
0083          case MouseEvent.MOUSE_RELEASED:
0084             state = INACTIVE;
0085             timer.stop();
0086             itsActionListener.actionPerformed(
0087                      new ActionEvent( this,
0088                             ActionEvent.ACTION_PERFORMED,
0089                             "arrow"));
0090             break;
0091          } // End switch.
0092          this.repaint();
0093       } // End if.
0094     } // End processMouseEvent.
```

The substantive part of this method is a four-way **switch** structure containing a **case** for each possible ID of the MouseEvent received as an argument. The selection branches implement the transitions, as shown on the state transition diagram, by changing the value of the *state* attribute, as appropriate, and the subsequent call of the repaint() method will make the change visible to the user. All of this is contained within an **if** structure which causes the entire **switch** structure and repaint() call to be ignored if the component is not currently enabled.

The transition from the *MOUSE_ENTERED* state to the *BUTTON_DOWN* state, caused by the receipt of a MouseEvent whose ID is MOUSE_PRESSED, also, on line 0080, causes the *timer* to be started. This will result in ActionEvent's being dispatched from the *timer* to this component's *actionPerformed()* method, which will be described shortly.

Likewise, the transition from the *BUTTON_DOWN* state to the *INACTIVE* state, caused by the receipt of a MouseEvent whose ID is MOUSE_RELEASED, on lines 0084 to 0098, not only stops the *timer* but also dispatches an ActionEvent from this component to *itsActionListener*, as indicated on the state transition diagram. The implementation of the *actionPerformed()* method is as follows.

```
0098    public void actionPerformed( ActionEvent event) {
0099       itsChangeListener.stateChanged(
0100                                  new ChangeEvent( this));
0101     } // End actionPerformed.
```

This method will be called periodically while the component is in the *BUTTON_DOWN* state, and upon receipt of the ActionEvent from the *timer* it constructs and dispatches a

ChangeEvent to *itsChangeListener*. Again, this is in accord with the ArrowBox's state transition diagram.

The *paint()* method has the responsibility of drawing the component in a manner appropriate to its *state* within the area accorded to it by its parent, *SpinBox*, container. It will be called indirectly, via repaint(), every time the *ArrowBox* transits from one state to another. The first part of the implementation of the *paint()* method is as follows.

```
0106     public void paint( Graphics context) {
0107
0108     Color background         = this.getBackground();
0109     Color activeForeground   = this.getForeground();
0110     Color inactiveForeground = null;
0111     Color disabledForeground = Color.lightGray;
0112
0113     Rectangle area = context.getClipBounds();
0114
0115     int midX    = area.width /2;
0116     int topY    = 4;
0117     int lowerY  = area.height -4;
0118     int leftX   = 4;
0119     int rightX  = area.width -4;
0120
0121     int xPositions[] = new int[ 3];
0122     int yPositions[] = new int[ 3];
0123
0124     int midRed   = ( activeForeground.getRed()   +
0125                      background.getRed()) /2;
0126     int midGreen = ( activeForeground.getGreen() +
0127                      background.getGreen()) /2;
0128     int midBlue  = ( activeForeground.getBlue()  +
0129                      background.getBlue()) /2;
0130
0131        inactiveForeground = new Color( midRed,
0132                                        midGreen, midBlue);
```

As described above, when the state transition diagram was explained, an *ArrowBox* instance needs four distinct colors. These are: the *background* color, the *disabledForeground* color used when the component is disabled, the *inactiveForeground* color used when the component is inactive and the *activeForeground* color used for the border in the *MOUSE_ENTERED* state and also for the arrow in the *BUTTON_DOWN* state. These four color variables are declared on lines 0108 to 0111, and all, apart from the *inactiveForeground*, can be initialized upon declaration. The *inactiveForeground* color is constructed, on lines 0131 and 0132, using red, green and blue (RGB) portions that are midway between the RGB values of the *activeForeground* and *background* colors, as determined on lines 0124 to 0129.

The *area* instance, obtained from the *context* on line 0113, contains four **public** attributes called x, y, width and height, which correspond to the location, width and height of the *ArrowBox*'s on-screen window. The five **int** variables, declared on lines 0115 to 0119, are initialized to delimit the extent of the two possible arrows, as shown in Figure 4.12. The *x* values of the required triangle will be placed into the three-element **int** array called

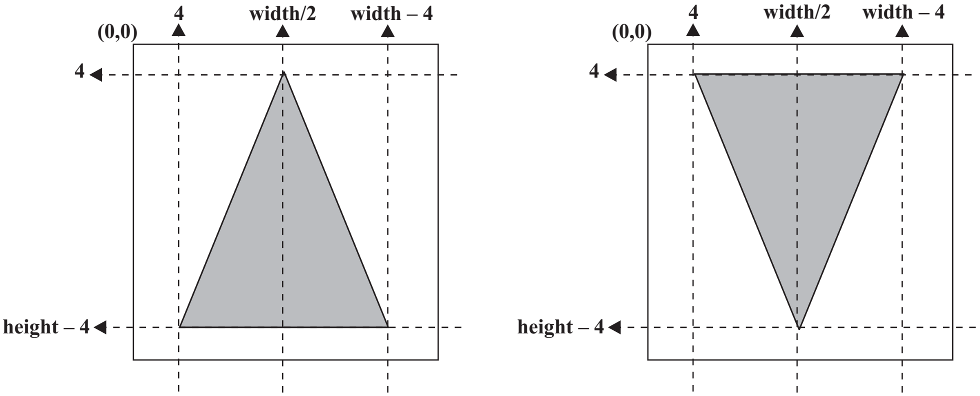


Figure 4.12 *ArrowBox* triangle vertices.

xPositions[] declared on line 0121, and likewise the *y* values in *yPositions[]*, to define the extent of the triangle later in the method. The *paint()* method continues as follows.

```
0134        context.setColor(background);
0135        context.fillRect( area.x,     area.y,
0136                          area.width, area.height);
0137
0138        if ( ! this.isEnabled()) {
0139           context.setColor( disabledForeground);
0140        } else if ( state == INACTIVE) {
0141           context.setColor( inactiveForeground);
0142        } else {
0143           context.setColor( activeForeground);
0144        } // End if.
0145        context.drawRect( area.x, area.y,
0146                          area.width-1, area.height-1);
0147        context.drawRect( area.x+1, area.y+1,
0148                          area.width-2, area.height-2);
```

The first stage of this method, on lines 0134 to 0136, clears the entire on-screen window by setting the *context*'s color attribute to the *background* color and then drawing a filled rectangle over the window's entire *area*. The next stage, on lines 0138 to 0144, prepares for drawing the border by setting the *context*'s color attribute to *disabledForeground*, *inactiveForeground* or *activeForeground*, according to the *state* of the component. The final stage, on lines 0145 to 0148, draws the border as two concentric outline rectangles around the entire extent of the *area*. The remainder of the *paint()* method is implemented as follows.

```
0150        if ( this.getOrientation() == UP) {
0151           xPositions[ 0] = midX;   yPositions[ 0] = topY;
0152           xPositions[ 1] = leftX;  yPositions[ 1] = lowerY;
0153           xPositions[ 2] = rightX; yPositions[ 2] = lowerY;
0154        } else {
0155           xPositions[ 0] = midX;   yPositions[ 0] = lowerY;
0156           xPositions[ 1] = leftX;  yPositions[ 1] = topY;
0157            xPositions[ 2] = rightX; yPositions[ 2] = topY;
0158         } // End if.
0159
0160         if ( ! this.isEnabled()) {
0161            context.setColor( disabledForeground);
0162         } else if ( state == BUTTON_DOWN) {
0163            context.setColor( activeForeground);
0164         } else {
0165            context.setColor( inactiveForeground);
0166         } // End if.
0167         context.fillPolygon(xPositions, yPositions, 3);
0168      } // End paint.
0169
0170 } // End class ArrowBox.
```

The first stage in this fragment, on lines 150 to 158, is to initialize the contents of the *xPositions* and *yPositions* arrays from the five prepared values, as indicated in Figure 4.12, depending upon the *orientation* of the arrow. Lines 0160 to 0166 then set the color attribute of the context, according to the *state* of the component, before the arrow is drawn with a call of *context*.fillPolygon(), on line 0167. There is an assumption here that the height of the component will be sufficient for the arrows to be sufficiently distinct, an assumption that will be challenged in an end of chapter exercise.

The overall effect of the *paint()* method is to show the ArrowBox on the screen in a manner appropriate to its internal *state* so that the user can be assured that the operations they are performing upon it with the mouse are being effected. This completes the implementation of the *ArrowBox* class, two instances of which are used by the *SpinBox* class which will be described next. A demonstration of the effectiveness of the *ArrowBox* implementation will be deferred until the *SpinBox* is demonstrated.

4.9 The *SpinBox*: design and implementation

An overview of the *SpinBox* design has already been given in Section 4.7, which included its state transition diagram in Figure 4.8 and an instance diagram in Figure 4.9. The *SpinBox* class diagram is given in Figure 4.13.

The class diagram shows that the *SpinBox* class extends the JComponent class, is a member of the *numericinput* **package** of classes and that it implements the ActionListener and ChangeListener **interface**s, in order to be able to listen to the events dispatched by its encapsulated *upBox* and *downBox ArrowBox* attributes. The third visible component, an instance of the *IntegerTextField* class called *inputArea*, is also shown as an instance attribute. The instance attributes are completed by its *actionListenerList* and the *propagateEvents* flag, the use of which will be described below.

The *SpinBox* has three constructors, corresponding to the three *IntegerTextField* constructors as described above. The first two methods, *getValue()* and *getIntValue()*, correspond to the *IntegerTextField* methods with the same name, returning the value indicated by the component. The next method, *setEnabled()*, is required to allow a *SpinBox* to be disabled and should be supplied by all extended components. The *setEditable()* method prevents, or allows, the user to type values into the input area and *isEditable()* is an inquiry attribute for this attribute.

The *getPreferredSize()*, *getMinimumSize()*, *getMaximumSize()* and *doLayout()* methods are required for layout negotiations and subsequently to actually lay out the component.

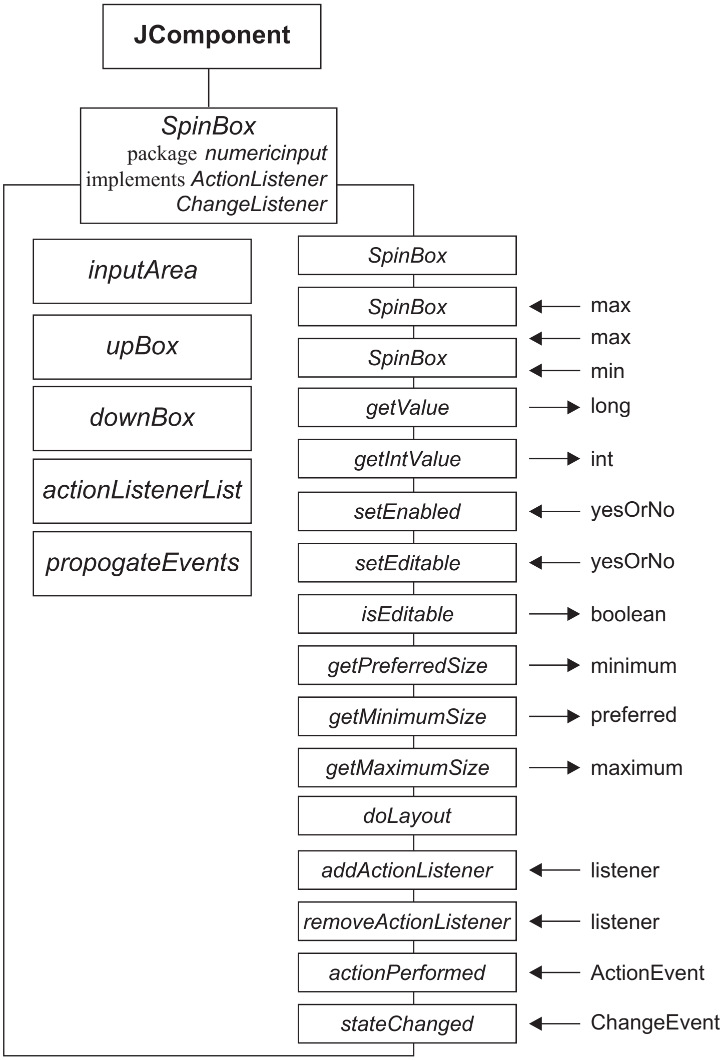


Figure 4.13 *SpinBox* class diagram.

The *addActionListener()* and *removeActionListener()* methods identify instances of this class as *ActionEvent* sources, as also indicated by the component's STD. The two remaining methods *actionPerformed()* and *stateChanged()* are mandated by the ActionListener and ChangeListener **interface**s respectively.

The implementation of the class, as far as the end of the second constructor, is as follows.

```
package numericinput;

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

```
0016 import javax.swing.border.*;
0017
0018
0019 public class SpinBox extends    JComponent
0020                      implements ActionListener,
0021                                 ChangeListener {
0022
0023 private IntegerTextField inputArea          = null;
0024 private ArrowBox         upBox              = null;
0025 private ArrowBox         downBox            = null;
0026 private ActionListener   actionListenerList = null;
0027 private boolean          propagateEvents    = true;
0028
0029    public SpinBox() {
0030       this( Long.MIN_VALUE, Long.MAX_VALUE);
0031    } // End IntegerTextField default constructor.
0032
0033    public SpinBox( long maximum) {
0034       this( 0, maximum);
0035    } // End IntegerTextField alternative constructor.
```

This part of the code proceeds as expected, declaring the class and its five attributes as shown on the class diagram. The two constructors indirect to the third constructor, passing on arguments in an identical manner to the *IntegerTextField* constructors. The implementation of the third constructor is as follows.

```
0037    public SpinBox( long minimum, long maximum) {
0038       inputArea = new IntegerTextField( minimum, maximum);
0039       inputArea.addActionListener( this);
0040       inputArea.getValue();
0041
0042       upBox     = new ArrowBox( ArrowBox.UP,   this );
0043       downBox   = new ArrowBox( ArrowBox.DOWN, this);
0044
0045       this.setLayout( null);
0046       this.add( inputArea);
0047       this.add( upBox);
0048       this.add( downBox);
0049
0050       this.setBackground(    inputArea.getBackground());
0051       this.setForeground(    inputArea.getForeground());
0052       upBox.setBackground(   this.getBackground());
0053       upBox.setForeground(   this.getForeground());
0054       downBox.setBackground( this.getBackground());
0055       downBox.setForeground( this.getForeground());
0056       this.setBorder( new LineBorder( Color.black, 1));
0057    } // End IntegerTextField constructor.
```

This constructor commences, on lines 0038 to 0040, by constructing and configuring the *inputArea*. The two arguments delineating the range of values that the *SpinBox* is to accept are passed on to the *IntegerTextField* constructor, following which **this** *SpinBox* being constructed is registered as the *inputArea*'s *actionListener*. The final step, on line 0040, calls the *inputArea*'s *getValue()* method and effectively throws away the value returned. It does this in order to ensure that the *SpinBox* is showing a value when it first becomes visible to the user, as indicated on its STD. As described above, a call of an *IntegerTextField*'s *getValue()* method when the input area is empty will cause it to display either zero or its *minimumValue*.

The next stage of the constructor, on lines 0042 and 0043, constructs the two *ArrowBox*es, one *UP* and one *DOWN*. The second argument to the constructor indicate that the ActionEvents and the ChangeEvents that it generates should be sent to **this** *SpinBox* currently being constructed.

Line 0045 removes any existing layout manager by calling the setLayout() method with the value **null**. As will be described below, this component has specialized layout requirements and takes responsibility for its own layout management. However, the three components which it takes responsibility for laying out are still added to it, on lines 0046 to 0048.

The remaining parts of the constructor ensure that the component has a consistent appearance by setting the foreground and background colors of **this** SpinBox and the *ArrowBox*es to those of the *inputArea*. The very last line of the constructor installs a single pixel black *LineBorder* into the *SpinBox*.

The *getValue()* and *getIntValue()* methods are implemented as wrapper methods indirecting to the corresponding *inputArea* methods, as follows.

```
0060     public long getValue() {
0061        return inputArea.getValue();
0062     } // End getValue.
0063
0064     public int getIntValue() {
0065        return inputArea.getIntValue();
0066     } // End getIntValue.
```

The *setEnabled()* method is likewise a wrapper method indirecting to the **super** and corresponding *setEnabled()* methods of the three contained components. The *setEditable()* method calls the *inputArea* setEditable() method and *isEditable()* is also a wrapper.

```
0069     public void setEnabled( boolean yesOrNo) {
0070        super.setEnabled(     yesOrNo);
0071        inputArea.setEnabled( yesOrNo);
0072        upBox.setEnabled(     yesOrNo);
0073        downBox.setEnabled(   yesOrNo);
0074     } // End setEnabled.
0075
0076
0077     public void setEditable( boolean yesOrNo) {
0078        inputArea.setEditable( yesOrNo);
0079     } // End setEditable.
0080
0081     public boolean isEditable() {
0082        return inputArea.isEditable();
0083     } // End isEditable.
```

The next method is *getPreferredSize()*, which is consulted during layout negotiations to determine how large the *SpinBox* would like to be. It is implemented as follows.

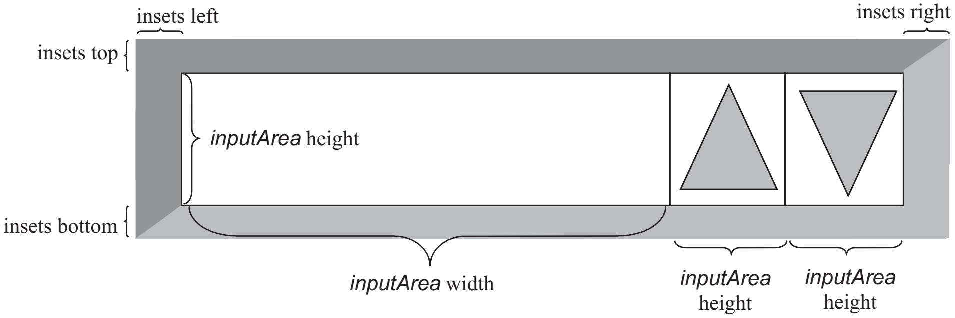


Figure 4.14 *SpinBox* layout requirements.

```
0085     public Dimension getPreferredSize() {
0086
0087     Dimension inputSize  = inputArea.getPreferredSize();
0088     Insets    borderSize = this.getInsets();
0089     Dimension required   = new Dimension();
0090
0091        required.width  = inputSize.width       +
0092                          (inputSize.height *2) +
0093                          borderSize.left       +
0094                          borderSize.right;
0095        required.height = inputSize.height      +
0096                          borderSize.top        +
0097                          borderSize.bottom;
0098        return required;
0099     } // End getPreferredSize.
```

This method has to return a Dimension instance, which is a simple class containing two public **int** attributes called width and height. The basis of how the *SpinBox* decides how large it would like to be is illustrated in Figure 4.14. The image shows an exaggerated beveled border in order to illustrate the four inset components, *top*, *bottom*, *left* and *right*, of its installed border.

The *SpinBox getPreferredSize()* method commences, on line 0087, by obtaining the *preferredSize* of the *inputArea* and, on line 0088, the *insets* of **this** *SpinBox*es' border. With these sizes obtained the *required* width and height of the *SpinBox* can be computed and subsequently returned from the method, on line 0098. The *minimum* and *maximum* sizes of a *SpinBox* are the same as its *preferred* size, as follows.

```
0102     public Dimension getMinimiumSize() {
0103        return this.getPreferredSize();
0104     } // End getMinimiumSize.
0105
0106     public Dimension getMaximumSize() {
0107        return this.getPreferredSize();
0108     } // End getMaximumSize.
```

During layout negotiations the *SpinBox*'s container parent in the window hierarchy will consult it for its size, using the three methods just described. The parent will then decide how large the *SpinBox* is to be and resize it accordingly, following which it will call its *doLayout()* method, as follows, so that it can inform its children where they should position themselves.

```
0111     public void doLayout() {
0112
0113     Dimension size            = this.getSize();
0114     Insets    borderSize      = this.getInsets();
0115     Rectangle position        = new Rectangle();
0116     int       componentHeight = 0;
0117
0118        componentHeight = size.height -
0119                          borderSize.top - borderSize.bottom;
0120
0121        position.x = size.width - borderSize.right -
0122                                                 componentHeight;
0123        position.y = borderSize.top;
0124        position.width  = componentHeight;
0125        position.height = componentHeight;
0126        downBox.setBounds( position);
0127
0128        position.x = position.x - componentHeight;
0129        upBox.setBounds( position);
0130
0131        position.x     = borderSize.left;
0132        position.width = size.width - borderSize.left -
0133                    borderSize.right - ( 2* componentHeight);
0134        inputArea.setBounds( position);
0135     } // End doLayout.
```

The *doLayout()* method must decide upon the location and size of each of the components that it contains and inform them of this by calling their setBounds() method. The method commences, on line 0113, by determining the *size* that the *SpinBox* has been accorded and continues by obtaining details of the *borderSize*. As shown on Figure 4.14, the height of each component should be the overall height of the *SpinBox* minus the top and bottom border insets; the result of this calculation, on lines 0118 and 0119, is stored in the *componentHeight* variable.

The left-hand location of the down-facing *ArrowBox* is the width of the *SpinBox* minus the right border inset, minus the width of the box (which is the same as its height). This calculation is effected on lines 0121 and 0122, and the result stored in the *position* x attribute. The *position* y attribute is the top border inset, as shown on line 0123. The *position* width and height are the *componentHeight*, as shown on lines 0124 and 0125. Having installed the location and size of the *downBox* in *position*'s attributes, the information is communicated to the *downBox*, on line 0126, by calling its setBounds() method, passing *position* as its argument. Similar considerations apply to the change made to *position*.x on line 0128 before the *upBox* setBounds() method is called on line 0129; likewise for the *inputArea* in the remainder of the method.

After the *SpinBox doLayout()* method has been called, when the component is to be painted onto the screen, Java will call each of the three components' paint() methods with a Graphics argument delineating the area of the screen that has been accorded to it. The effect is that this method arranges for the components contained within it to be positioned appropriately when they are subsequently shown on the screen. The *doLayout()*

method will also be called if the component is ever resized before it is repainted, so that the geometrical relationships can be reestablished for the new size. It might seem that a FlowLayout manager would be equally effective for a *SpinBox*'s layout requirement; however, there are some considerations that will be introduced at the end of the chapter which make it unsuitable.

The implementation of the *SpinBox* class continues with the *addActionListener()* and *removeActionListener()* methods, as follows.

```
0139     public void addActionListener( ActionListener toAdd) {
0140        actionListenerList = AWTEventMulticaster.add(
0141                                     actionListenerList, toAdd);
0142     } // End addActionListener.
0143
0144     public void removeActionListener( ActionListener toRemove) {
0145        actionListenerList = AWTEventMulticaster.remove(
0146                                     actionListenerList, toRemove);
0147     } // End removeActionListener.
```

The absence of an exclamation mark in the *addActionListener()* method in the class diagram and the signature of the method's declaration, on line 0139, indicate that this implementation will not throw exceptions and so is an event multicaster. Hence it has a requirement to maintain a list of ActionListeners, instead of the single ActionListener maintained by the *Timer* class in Chapter 1. This is a common requirement that the designers of the AWT recognized, and they supplied a class called AWTEventMulticaster to support developers in satisfying this requirement. The *actionListenerList* attribute was declared on line 0026 as an instance of the ActionListener interface type with a **null** value.

The implementation of the *SpinBox addActionListener()* method, on lines 0139 to 0142, uses the AWTEventMulticaster add() method to add the new listener, passed in the argument *toAdd*, to the existing list. Likewise, the implementation of the *SpinBox removeActionListener()* method, on lines 0144 to 0147, uses the AWTEventMulticaster remove() method to remove the listener, *toRemove*, from the existing list. Should the list become empty when a listener is removed, the *actionListenerList* will have its **null** value restored to it. This mechanism can be used by any extended component that has a requirement to maintain a list of listeners; the two required *add* and *remove* methods can be implemented as shown above.

The *SpinBox* will receive an ActionEvent from the *inputArea* when the user presses the <ENTER> key. It will also receive ActionEvents from the *upBox* and *downBox* when the user releases the mouse button, as shown in the *ArrowBox* STD in Figure 4.11. The *SpinBox* STD, in Figure 4.8, indicates that the receipt of these events should result in the *SpinBox* dispatching an ActionEvent to its listeners. Accordingly the *SpinBox actionPerformed()* method is responsible for both receiving an ActionEvent from its constituent components and propagating an ActionEvent onwards, as follows.

```
0149     public void actionPerformed( ActionEvent event) {
0150       if ( actionListenerList != null &&
0151            propagateEvents            ){
0152          actionListenerList.actionPerformed(
0153                new ActionEvent( this,
0154                                 ActionEvent.ACTION_PERFORMED,
0155                                 inputArea.getText()));
0156         } // End if.
0157      } // End actionPerformed.
```

The compound condition on lines 0150 and 0151 prevents this method from attempting to dispatch any events if there are no listeners in the *actionListenerList*; i.e. if it has a **null** value. It will also not dispatch events if the *propagateEvents* flag is **false**, for reasons that will be described below. If an ActionEvent is to be dispatched from this *SpinBox* it is constructed on lines 0153 to 0155. The three arguments to the constructor are the identity of **this** *SpinBox*, the manifest reason and, in its actionCommand attribute, the text contents of the *inputArea*.

The newly constructed ActionEvent is apparently passed as an argument to the actionPerformed() method of the *actionListenerList*. However, because the *actionListenerList* is being maintained by the AWTEventMulticaster class this call will be multiplexed. That is, the actionPerformed() method of every actionListener in the list will be called with a copy of the ActionEvent passed to each. However, there is no guarantee that the listeners in the list will be called in any particular sequence. The final method in the *SpinBox* class is the *stateChanged()* method, mandated by the ChangeListener interface, and implemented as follows.

```
0161      public void stateChanged( ChangeEvent event) {
0162
0163      ArrowBox source   = (ArrowBox) event.getSource();
0164      long     value    = inputArea.getValue();
0165
0166
0167         if ( source.getOrientation() == ArrowBox.UP) {
0168            if ( value == inputArea.getMaximum()) {
0169               value = inputArea.getMinimum();
0170            } else {
0171               value++;
0172            } // End if.
0173         } else {
0174            if ( value == inputArea.getMinimum()) {
0175               value = inputArea.getMaximum();
0176            } else {
0177               value-;
0178            } // End if.
0179         } // End if.
0180
0181         propagateEvents = false;
0182         inputArea.setValue( value);
0183         propagateEvents = true;
0184
0185      } // End stateChanged.
0186
0187 } // End SpinBox class.
```

This method will be called from one of the *ArrowBox*es' *actionPerformed()* methods, which in turn is called by the *ArrowBox*es' *timer* while the mouse button is held down. Accordingly this method has to implement the four transitions shown at the bottom of the *SpinBox*'s STD in Figure 4.8.

The method commences, on line 0163, by obtaining the identity of the *ArrowBox* which was the *source* of the event and, on line 0164, the *value* of the *inputArea*. The substantive part of the method, between lines 0167 and 0179, uses a nested **if** structure to decide whether the box is spinning up or down, and in each outer branch deciding whether the value can be incremented or decremented or whether it needs to spin around the range of values. Having adjusted the *value* it is reinstalled into the *inputArea*, on line 0182, by calling its *setValue()* method, which will update the value shown in the input area.

However, the call of *setValue()* on line 0182 will result in the *inputArea* dispatching an ActionEvent to inform its listeners that its contents have changed. This will result in the *SpinBox*'s *actionPerformed()* method being called and would result in the *SpinBox* dispatching a sequence of ActionEvents as its contents change, rather than a single event when spinning finishes. To prevent this from happening the *propagateEvents* flag is set **false**, which, as explained above, will prevent the *SpinBox*'s *actionPerformed()* method from dispatching an event when the value is changed on line 0182. The last step of the *stateChanged()* method is to reset the *propagateEvents* flag to **true.** Figure 4.15 illustrates these considerations.

The series of steps labeled (***1***), (***2***), (***3***) shows the propagation of an *ActionEvent* from the *inputArea* when the user presses ENTER, through the *SpinBox*'s *actionPerformed()* method to the *SpinBox*'s listeners. The series of steps labeled (***a***) to (***k***) shows the sequence of actions caused by the user pressing down an arrow button and subsequently releasing it. The first consequence (***b***) is to call the *timer* start() method; this causes the *timer* to dispatch ActionEvents to the *ArrowBox* *actionPerformed()* method (***c***), which in turn propagate ChangeEvents to the *SpinBox*es *stateChanged()* method (***d***). This method sets the *propagateEvents* flag **false** (***e***) and then calls the *inputArea*'s *setValue()* method (***f***). This results in an ActionEvent being dispatched to the *SpinBox*'s *actionPerformed()* method (***g***), but does not result in the *SpinBox* dispatching an event. Although not shown on this diagram the *propagateEvents* flag will be reset to **true** at this point.

When the user releases the mouse button (***h***), the *timer* stop() method is called (***i***) and an ActionEvent being dispatched to the *SpinBox*'s *actionPerformed()* method (**j**) which, as the *propagateEvents* flag is **true**, will result in an ActionEvent being dispatched to the*SpinBox*'s listeners (***k***).

A small change to the *NumericInputDemo* class's *init()* method adds a *SpinBox* instance to the interface, as shown in Figure 4.16. The width of the *SpinBox* is determined by the width of the *numericinput* areas above it, not by its *preferredSize* resource. However, the *SpinBox*'s *doLayout()* method is able to accommodate to these dimensions, as shown. A consequential change to its *actionPerformed()* method, given below, causes it to output the value of the *SpinBox* when it dispatches an *ActionEvent*.

```
0087    public void actionPerformed( ActionEvent event) {
0088
0089    Component theSource = (Component) event.getSource();
0090    String    theName   = theSource.getName();
0091
0092       System.out.print( theName + " ");
0093
0094       if ( theName.equals( "Floating point")) {
0095          System.out.println( ((NumericTextField) theSource).
0096                                                getValue());
0097       } else if ( theName.equals( "Spin box")) {
```

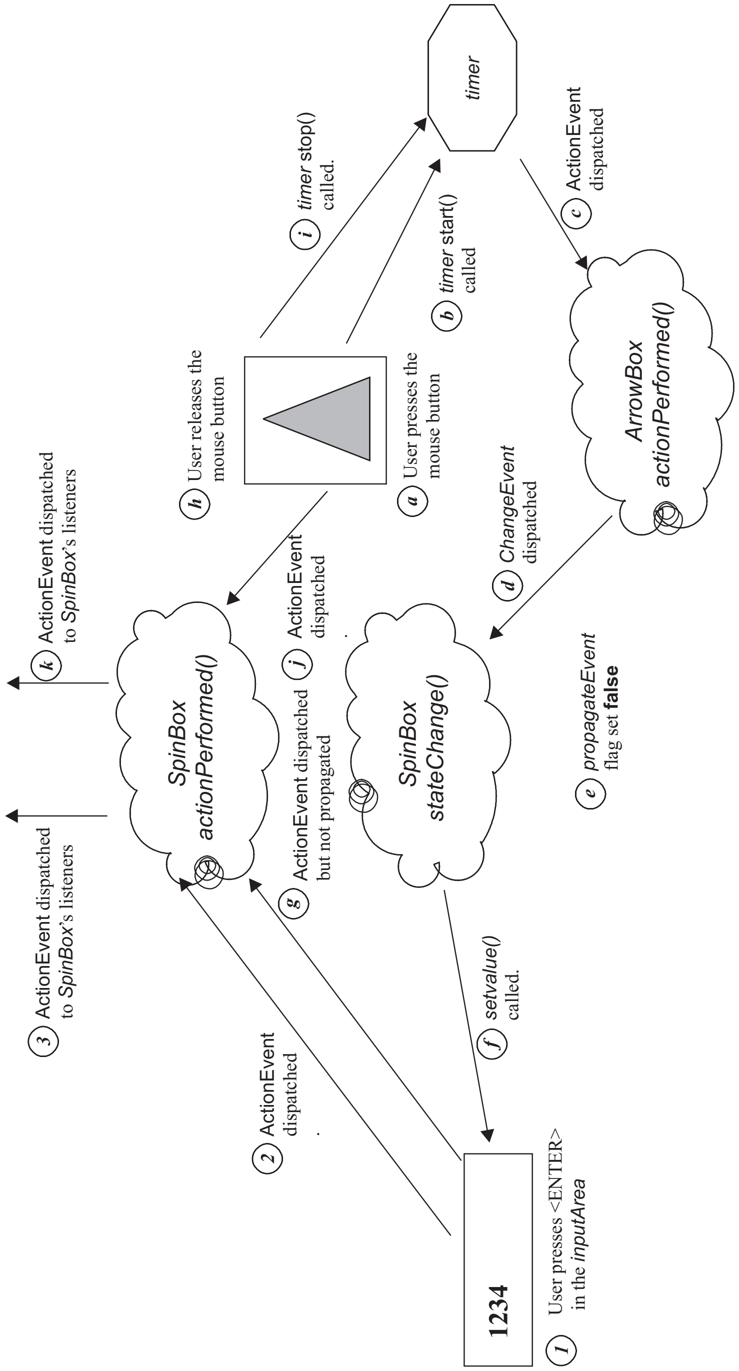
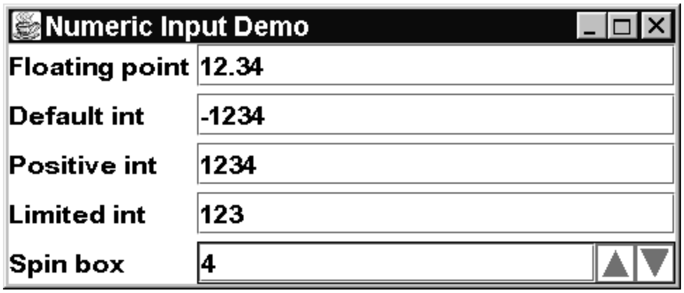

1234
1 User presses <ENTER> in the inputArea
2 ActionEvent dispatched
3 ActionEvent dispatched to SpinBox's listeners
SpinBox actionPerformed()
g ActionEvent dispatched but not propagated
f setvalue() called.
SpinBox stateChange()
e propagateEvent flag set false
k ActionEvent dispatched to SpinBox's listeners
j ActionEvent dispatched
h User releases the mouse button
a User presses the mouse button
d ChangeEvent dispatched
ArrowBox actionPerformed()
i timer stop() called.
b timer start() called
timer
c ActionEvent dispatched

Figure 4.15 *SpinBox* event propagation.



Numeric Input Demo

Floating point 12.34

Default int -1234

Positive int 1234

Limited int 123

Spin box 4

Figure 4.16 Extended *NumericInputDemo*.

```
          System.out.println( ((SpinBox) theSource).
                                                    getValue());
        } else {
          System.out.println( ((IntegerTextField) theSource).
                                                    getValue());
        } // End if.
    } // End actionPerformed.
```

This amended artifact can be used to demonstrate that the ActionEvents dispatched by the *inputArea* as it is spinning do not result in the demonstration client being informed: only the release of the mouse button causes it to receive an ActionEvent.

4.10 The *numericinput* hierarchy in retrospect

Although the specialized components presented in this chapter are reasonably robust and engineered, they are not yet sufficiently developed to be on a par with the JFC components. For example the processing of the key events in the *NumericTextField* and *IntegerTextField* components deliberately ignored any cut/copy/paste key presses. Although cut and copy operations would be easy to accommodate, the paste operation would require the changed contents of the field to be examined to ensure that they remain acceptable.

Perhaps more noticeably, many of the methods which have been inherited from JTextField have not been explicitly determined to be acceptable for these components. To complete the development of the components, every method inherited from JTextField should be examined and classified as: acceptable in its existing form, acceptable with amendments or unacceptable. If there are no unacceptable methods then those methods which require amendment can be overridden to implement the required changes. If there are any unacceptable methods then either the methods should be overridden without a call of the unacceptable super method or, if this is not appropriate, implementation as a composite object which contains only a single component should be considered.

For example, all three specialized components inherit setForeground(), set Background() and setFont() methods from the AWT Component class. These methods are acceptable as they stand to the *NumericTextField* and *IntegerTextField* components, but need to be amended for the *SpinBox* component. When either of the colors change this message has to be propagated to all contained components and the *ArrowBox* will have to compute a new *inactiveForeground* color. Likewise, the preferred size of a *SpinBox* is determined by the Font that is used, so the *setFont()* method will have to be overridden to allow for this.

Summary

- ♦ Specialized components can be developed from the JFC components to satisfy particular requirements.
- ♦ Internal event handling, by explicitly enabling events and overriding the **protected** methods that process them, is the preferred way to implement a specialized component's essential behavior.
- ♦ KeyEvents are dispatched to a component as the user interacts with the keyboard and are processed by the processKeyEvent() method.
- ♦ MouseEvents are dispatched to a component as the user interacts with the mouse and are processed by the processMouseEvent() and processMouseMovementEvent() methods.
- ♦ Specialized components may be composite components, containing other components, and internal event processing may involve the event source/listener mechanism.
- ♦ A specialized composite component may take responsibility for its own layout management.
- ♦ Any event that should not be further processed should be consumed.
- ♦ The development of a specialized component should consider the possible effects of every inherited method.

Exercises

4.1 Devise a formal black box test plan for the specialized components and use the demonstration test harness to apply it.

4.2 Extend the *IntegerTextField* constructor so that it throws an exception if the minimum is greater than the maximum. Use this component to investigate effect of an event being thrown during construction.

4.3 Make a list of all the methods inherited by the specialized components and re-engineer the implementations as appropriate, repeating all the tests from Exercise 4.1 as well as any new tests which are required.

4.4 Design and implement a *P*ersonal *I*dentification *N*umber (PIN) component. It should extend the JPasswordField component, so that the digit characters are not displayed. The component should allow exactly four digits to be input and should dispatch an ActionEvent when four digits have been supplied.

4.5 Design and implement a *TimeOfDay* component that will allow the user to input the time of day using a 24 hour representation in the format *(h)h:(m)m*. That is, one or two digits in the range 0 to 12, a colon, and one or two digits in the range 0 to 59.

4.6 Extend the *NumericTextField* to support a minimum and maximum value and also to allow the user to input a floating point value in the format 1.2345E±12.

4.7 Reimplement the *SpinBox*'s layout policy so that if the height of the component is greater than a crucial value (say 30 pixels) the two arrow boxes should be laid out at the right-hand edge, one above the other.

5

More specialized components – the *cards* package

5.1 Introduction

In this chapter the design and development of specialized components will be reinforced by the consideration of a second example, the *cards* package. A *card* contains two images: one for the front of the card and one for the back of the card, and provides methods to switch between them. It is also responsive to the user interacting with it using the mouse.

The intention behind this first part of the chapter is not only to reinforce the concepts of internal and external behavior introduced in the previous chapter, but also to illustrate again the techniques that allow a specialized component to respond to the mouse. As with the *numericinput* components, the implementations presented in this chapter will not be fully engineered; they are developed only so far that the most fundamental and salient points can be illustrated.

Once the *cards* hierarchy has been developed to the point where instances can respond to the user it will be used to implement an artifact known as *MemoryGame*. This is a child's game, played with a set of cards containing matching pairs. The game is played by laying out all the cards, face downwards, on a surface. Each player, in turn, then turns over two cards and removes them if they match. The development of this game will allow the use of the hierarchy to be illustrated in a realistic environment.

The next chapter will further develop the *cards* hierarchy, allowing instances of the class to be dragged and dropped. This capability is essential to the provision of interfaces that support direct manipulation, which can be regarded as one of the most powerful interface styles.

5.2 The *cards* package

The *cards* class hierarchies contained in the package are illustrated in Figure 5.1. The abstract root class, *StateCard*, extends the JComponent class and supports the manifest states that a card can be in. The *TurnCard* class supports the front and back images that a card can display. The *ActiveCard* class supports interactive behavior with the user, dispatching events when the user clicks upon it. The *DragCard* class supports dragging behavior, allowing the user to directly manipulate the position of a card.

The first three classes in the *cards* hierarchy will be described in the first part of the chapter and instances of the *ActiveCard* class will be used in the *MemoryGame* artifact. The *DragCard* class will be described in the next chapter.

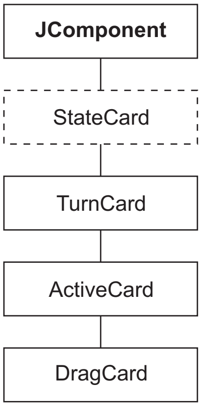


Figure 5.1 The *cards* package.

5.3 The *StateCard* class

The **abstract** *StateCard* class is at the root of the *cards* hierarchy and supplies the fundamental attributes and methods concerned with the two faces of a card; its class diagram is given in Figure 5.2.

The first four **public** resources are manifest values describing the state of the card: at any moment a card can be showing its front or its back or be in the process of turning, or it may have been removed from the playing surface. The only attribute, *cardState*, is used to record the current state of the card. The five **public** methods are inquiry methods allowing the state of the card to be determined. The default constructor initializes the card to the *FACE_SHOWING* state and the only remaining method, *setCardState()*, allows the state of the card to be changed. This method has **protected** visibility so that only other classes in the same hierarchy can make use of it.

The implementation of this class is relatively straightforward and is presented, as follows, without comment.

```
package cards;

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

abstract class StateCard extends JComponent {

public  final static int FACE_SHOWING = 0;
public  final static int BACK_SHOWING = 1;
public  final static int TURNING      = 2;
public  final static int REMOVED      = 3;
private int              cardState    = FACE_SHOWING;


   public StateCard() {
      super();
      cardState = FACE_SHOWING;
      this.setBackground( Color.white);
```

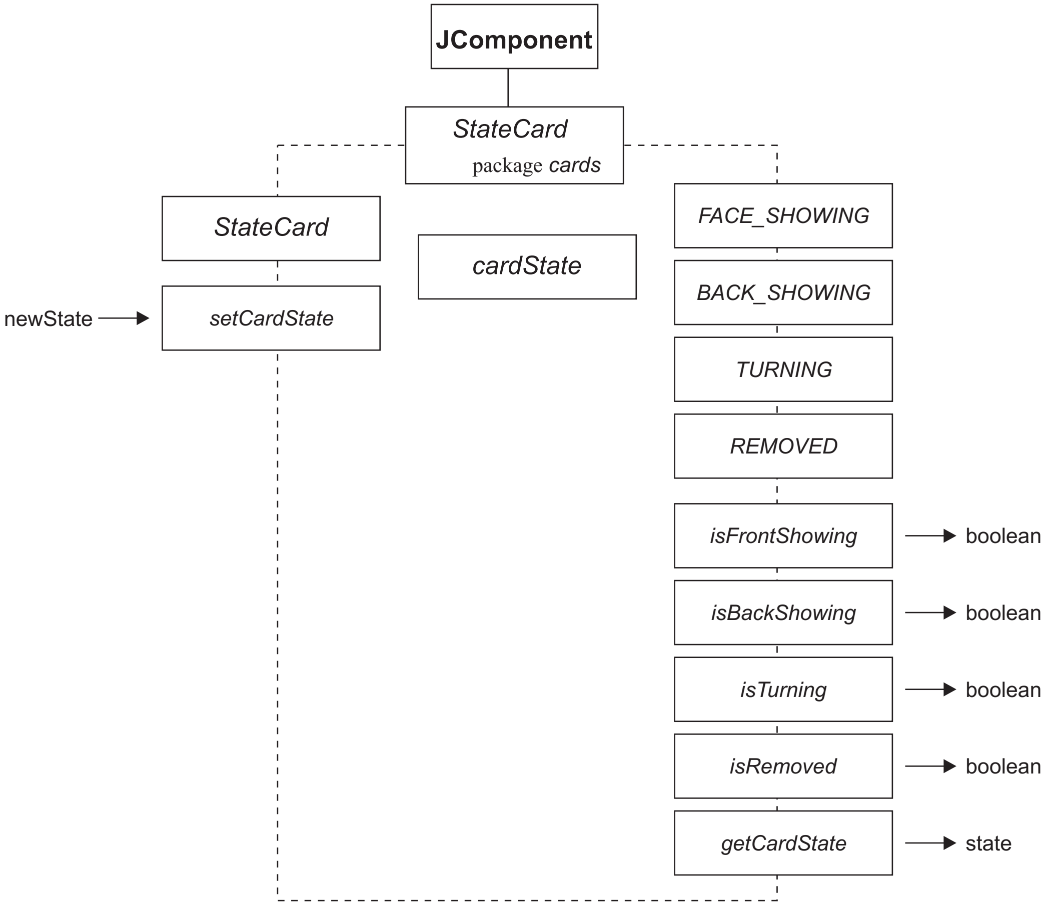


Figure 5.2 The *StateCard* class diagram.

```
      this.setOpaque( true);
   } // End StateCard constructor.

   public boolean isFrontShowing() {
      return cardState == FACE_SHOWING;
   } // End isFrontShowing.

   public boolean isBackShowing() {
      return cardState == BACK_SHOWING;
   } // End isBackShowing.

   public boolean isTurning() {
      return cardState == TURNING;
   } // End isTurning.

   public boolean isRemoved() {
      return cardState == REMOVED;
   } // End isRemoved.

   protected void setCardState( int newState) {
      cardState = newState;
   } // End setCardState.

   public int getCardState() {
      return cardState;
   } // End getCardState.

} // End class StateCard.
```

5.4 The *TurnCard* class

The *TurnCard* class diagram is given in Figure 5.3. It shows that the class has two major attributes, *frontOfCard* and *backOfCard*, which are the images to show in the appropriate state, the only constructor requires that both images are supplied. Also shown as an attribute is a *timer*, which will be used to control the animated turning of the card. As the

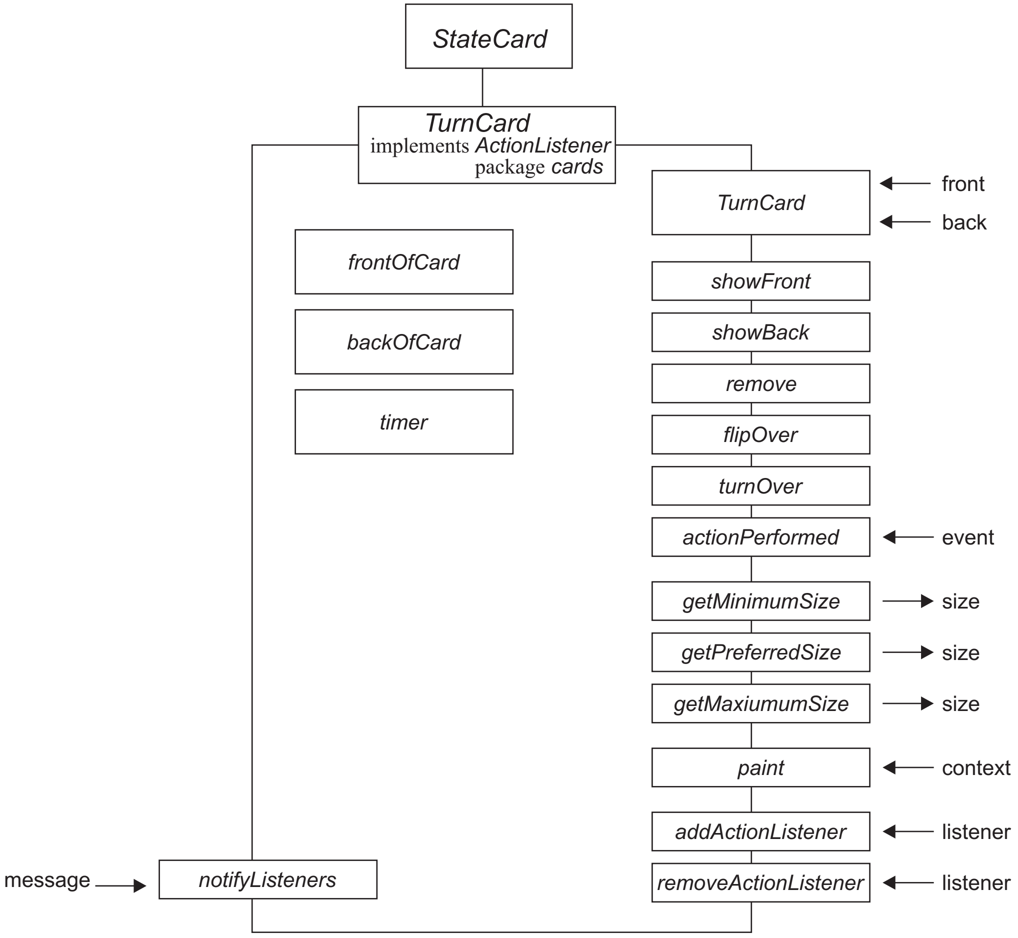


Figure 5.3 The *TurnCard* class diagram.

timer is an ActionEvent source, the *TurnCard* must implement the ActionListener interface so that it can receive the events that it dispatches.

The first three **public** methods cause the card to immediately redisplay itself in the state indicated in their names. The *flipOver()* method has no effect if the card is in the turning or removed states; otherwise it will immediately toggle the card between showing its front or back. The *turnOver()* method also has no effect in the turning or removed states and changes the image shown otherwise. However, while the *flipOver()* method changes the appearance immediately the *turnOver()* method does it gradually, revealing the card from top to bottom. While the *turnOver()* method is turning the card it will be in the *TURNING* state, as supplied by the *StateCard* class. The *actionPerformed()* method is shown associated with this group of methods as it plays a significant role in turning the card.

The next group of three methods is required for effective layout negotiation as explained in the previous chapter. The *paint()* method is supplied so that the card can display itself with an appearance appropriate to its state. The two remaining public methods, *addActionListener()* and *removeActionListener()*, identify instances of this class as multicasting event sources. This facility is introduced into the hierarchy by this class as it will dispatch events to its listeners to indicate that it is starting to turn and that it has finished turning, so that they will know not to send any messages to it while this happening. The dispatch of an ActionEvent is effected by the **protected** *notifyListeners()* method, whose *message* argument is copied into the actionCommand resource of the event dispatched.

The implementation of this design, as far as the start of the first constructor, is as follows.

```
0010  package cards;
0011
0012  import java.awt.*;
0013  import java.applet.*;
0014  import java.awt.event.*;
0015  import javax.swing.*;
0016  import javax.swing.border.*;
0017
0018
0019  public class TurnCard extends    StateCard
0020                        implements ActionListener {
0021
0022  private Image          frontOfCard        = null;
0023  private Image          backOfCard         = null;
0024
0025  private Timer          timer              = null;
0026
0027  private ActionListener actionListenerList = null;
0028  private int            turningTo          = FACE_SHOWING;
0029  private int            phase              = 0;
0030  private int            imageHeight        = 0;
0031  private int            imageWidth         = 0;
```

The class is declared on lines 0019 and 0020, exactly as expected from the class diagram. The *frontOfCard* and *backOfCard* attributes are declared on lines 0022 and 0023 as instances of the AWT Image class. An example of how these resources can be supplied will be given when the *TurnCardDemo* class is briefly described at the end of this section.

The *timer* attribute, an instance of the Timer class, is declared on line 0025. The remaining attributes are concerned with the detailed internal operation of the class and so were not shown on the class diagram. The *actionListenerList* maintains the list of objects to which events should be dispatched. The *turningTo* and *phase* attributes are concerned with the *turnOver()* method and their role will be described when the turning operation is described below. The implementation of the constructor is as follows.

```
0033     public TurnCard( Image front,
0034                      Image back) {
0035        super();
0036        frontOfCard  = front;
0037        backOfCard   = back;
0038
0039        timer = new Timer( 50, this);
0040
0041        imageWidth  = frontOfCard.getWidth(  this)
0042        imageHeight = frontOfCard.getHeight( this)
0043     } // End TurnCard constructor.
```

The constructor commences by calling its **super**, *StateCard*, default constructor, and continues by storing the arguments in their corresponding attributes. The next construction action, on line 0037, is to construct the *timer* so that it will dispatch events to **this** *TurnCard* every twentieth of a second (i.e. every 50 milliseconds). The constructor concludes on lines 0041 and 0042 by determining the width and height of the *frontOfCard* image, assuming that *backOfCard* image has the same dimensions. The next four methods change the state and appearance of the card, as follows.

```
0045     public void showFront() {
0046        this.setCardState( FACE_SHOWING);
0047        this.repaint();
0048     } // End showFront
0049
0050     public void showBack() {
0051        this.setCardState( BACK_SHOWING);
0052        this.repaint();
0053     } // End showBack
0054
0055     public void remove() {
0056        this.setCardState( REMOVED);
0057        this.repaint();
0058     } // End remove.
0059
0060     public void flipOver() {
0061        if ( this.isFrontShowing()) {
0062           this.showBack();
0063        } else if ( this.isBackShowing()) {
0064           this.showFront();
0065        } // End if.
0066     } // End flipOver.
```

The first three of these methods are each constructed in a similar manner, calling the inherited *setCardState()* with an appropriate argument, and then calling the repaint() method, which, as explained in the previous chapter, will cause the *paint()* method to be called in due course, showing the card in its changed state. The *flipOver()* method will indirect either to the *showBack()* or *showFront()* method if the face or back is currently showing, respectively, and will do nothing in the other two possible states. The *turnOver()* and *actionPerformed()* methods are implemented as follows.

```
0069     public void turnOver() {
0070        if ( this.isFrontShowing() ||
0071             this.isBackShowing()  ){
0072           if ( this.isFrontShowing()) {
0073              turningTo = BACK_SHOWING;
0074           } else if ( this.isBackShowing()) { ;
0075              turningTo = FACE_SHOWING;
0076           } // End if.
0077           phase = 0;
0078           this.setCardState( TURNING);
0079           this.notifyListeners( "turning");
0080           timer.start();
0081        } // End if.
0082     } // End turnOver.
0083
0084     public void actionPerformed( ActionEvent event) {
0085        phase++;
0086        if ( phase > imageHeight) {
0087           this.setCardState( turningTo);
0088           this.notifyListeners( "turned");
0089           timer.stop();
0090        } // End if.
0091        this.repaint();
0092     } // End actionPerformed.
```

The *turnOver()* method commences with a guard, on lines 0070 and 0071, that causes the method to do nothing if it is ever called while it is in the *TURNING* or *REMOVED* states. If the card is to be turned, lines 0072 to 0076 set the value of the *turningTo* attribute to indicate which side of the card will be showing after it has turned. Line 0077 sets the value of the *phase* attribute to zero to indicate that it is at the very start of the turning operation. The method continues by setting the state of the card to *TURNING*, notifies its listeners that it is turning and then start()s the *timer*.

As a consequence of the *timer* being started by the *turnOver()* method, it will dispatch ActionEvents to the *actionPerfomed()* method, implemented on lines 0084 to 0092. Upon the receipt of each *event* the value of *phase* is incremented and, as will be explained, when it becomes greater than the value of *imageHeight* the card has completely turned over. As a consequence, on lines 0087 to 0089, the *state* of card is set to *turningTo*, its listeners are notified that it has *turned* and the *timer* is stopped. In all cases the card is repainted before the method finishes and if the card is in the process of turning the *paint()* method, which follows below, will reveal a little more of the face that it is turning to.

The next three methods are required for layout negotiations, and are implemented as follows.

```
0094     public Dimension getMinimumSize() {
0095 
0096     Insets    insets   = this.getInsets();
0097     Dimension required = new Dimension();
0098 
0099        required.width  = imageWidth +
0100                          insets.left + insets.right;
0101        required.height = imageHeight +
0102                          insets.top + insets.bottom;
0103        return required;
0104     } // End getMinimumSize.
0105 
0106     public Dimension getPreferredSize() {
0107        return this.getMinimumSize();
0108     } // End getPreferredSize.
0109 
0110     public Dimension getMaximumSize() {
0111        return this.getMinimumSize();
0112     } // End getMaximumSize.
```

The *getPreferredSize()* and *getMaximumSize()* methods both indirect to the *getMinimumSize()* method, indicating that all three will request the minimum required size. The *getMinimumSize()* method returns a *required* width based upon the width of the *frontOfCard* image and the left and right attributes of the *insets* reserved for the border. The *required* height is likewise based upon the height of the *frontOfCard* image and the top and bottom *insets*. These methods are implemented upon the assumption that the *frontOfCard* and *backOfCard* images have the same dimensions; an assumption that in a more secure implementation would have been checked upon construction.

The *paint()* method is by far the most complex method in the *TurnCard* class and commences as follows. Figure 5.4 illustrates some of the considerations which will shortly be explained and also clarifies the *getMinimimSize()* method, as just described.

```
0114    public void paint( Graphics context) {
0115 
0116    int    windowWidth  = this.getSize().width;
0117    int    windowHeight = this.getSize().height;
0118    Insets insets       = this.getInsets();
0119 
0120    int top  = ( windowHeight - imageHeight) /2;
0121    int left = ( windowWidth  - imageWidth ) /2;
0122 
0123       context.setClip( insets.left, insets.top,
0124                        windowWidth  - insets.left - insets.right,
0125                        windowHeight - insets.top  - insets.bottom);
0126 
0127       context.setColor( this.getBackground());
0128       context.fillRect( 0, 0, windowWidth, windowHeight);
0129       context.setColor( this.getForeground());
```

Lines 0114 to 0118 declare and initialize local variables whose values are as indicated in Figure 5.4. Lines 0120 and 0121 then compute the location of the *top left*-hand corner of the position of the image. Having initialized these variables, on lines 0123 to 0125, the

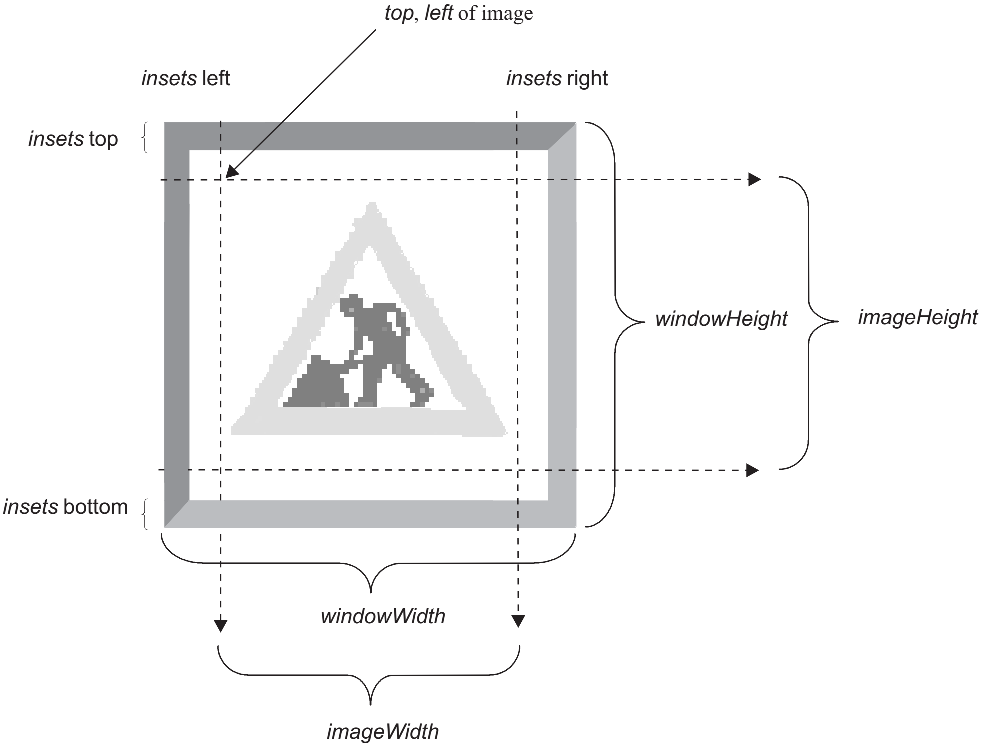


Figure 5.4 *TurnCard*: layout and paint considerations.

area of the window that the Graphics *context* can draw upon is clipped to the area within the border's insets. Having done this, the entire window is filled with the current background color, on lines 0127 and 0128, and on line 0129 the state of the *context* is restored so as to draw with the foreground color. If the *TurnCard* instance is currently in the *REMOVED* state these are the only steps that the *paint()* method will take, leaving the appearance of the component as a bordered area of the background color.

The next stage of the *paint()* method takes care of the appearance of the component when it is in the *FRONT_SHOWING* or *BACK_SHOWING* states, as follows.

```
0131       if ( this.isFrontShowing()) {
0132          context.drawImage( frontOfCard, left, top, this);
0133
0134       } else if ( this.isBackShowing()) {
0135          context.drawImage( backOfCard, left, top, this);
```

Each of these fragments will draw either the *frontOfCard* or the *backOfCard* image specifying the *left*-hand *top* location of where the image is to be placed. The substantive part of the method is concerned with the appearance of the card while it is turning and commences as follows.

```
0137       } else if ( this.isTurning()) {
0138          if ( turningTo == BACK_SHOWING) {
0139             context.setClip( left, top, imageWidth, phase);
0140             context.drawImage( backOfCard, left, top, this);
```

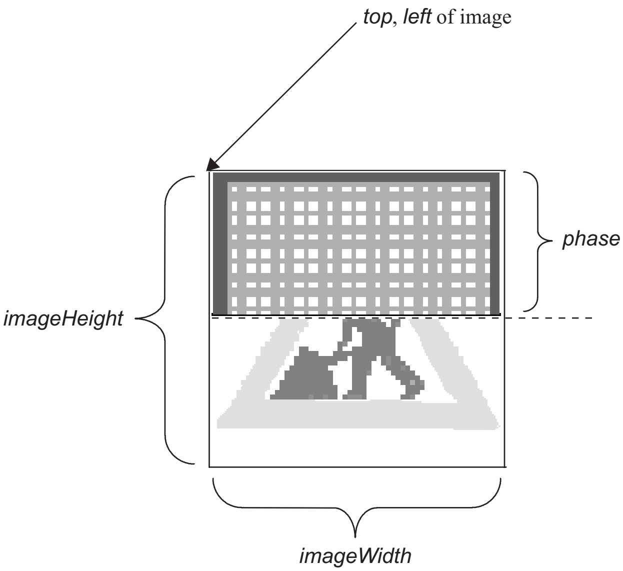
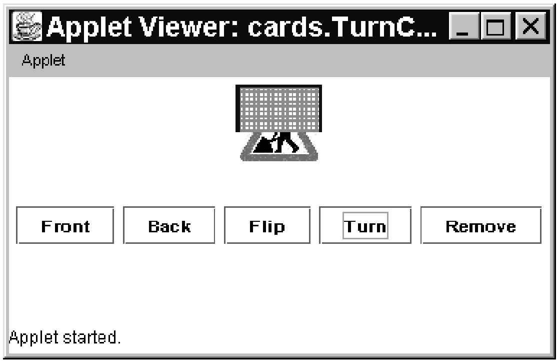


Figure 5.5 *TurnCard* during turning.

```
0141                 context.setClip( left, top + phase,
0142                                  imageWidth, imageHeight - phase);
0143                 context.drawImage( frontOfCard, left, top, this);
0144              } else {
0145                 context.setClip( left, top, imageWidth, phase);
0146                 context.drawImage( frontOfCard, left, top, this);
0147                 context.setClip(  left, top + phase,
0148                                   imageWidth, imageHeight - phase);
0149                 context.drawImage( backOfCard, left, top, this);
0150              } // End if.
0151           } // End if.
0152        } // End paint.
```

The basis of turning the card is illustrated in Figure 5.5, where a card is shown approximately halfway through turning from the front to the back. At this stage the value of *phase* is halfway between zero and *imageHeight*. Line 0138 establishes that the card is turning from the front to the back, and line 0139 establishes a clip window whose bounds commence at the *top, left* location and extend the full width of the image (*imageWidth*) with a height equal to *phase*. Following this, on line 0140, the *backOfCard* image is drawn with its top left corner positioned at the *top*, *left* coordinates. Due to the clip window just established, only that part of the image that lies within the clipped area will appear on the screen.

Having drawn the upper part of the turning image containing half of the back image, lines 0141 and 0142 establish a clip window whose extent covers the lower half of the area occupied by the images. Line 0143 then draws the *frontOfCard* image with its top left corner positioned at the *top*, *left* coordinates. Due to the second clip window just established, only that part of the image that lies within the new clip area will appear on the



Applet started.

Figure 5.6 *TurnCardDemo* artifact.

screen. Lines 0145 to 0149 are essentially identical and are concerned with turning the card from the back to the front.

```
0154      public void addActionListener( ActionListener toAdd) {
0155         actionListenerList = AWTEventMulticaster.add(
0156                                  actionListenerList, toAdd);
0157      } // End addActionListener.
0158
0159      public void removeActionListener( ActionListener toRemove) {
0160         actionListenerList = AWTEventMulticaster.remove(
0161                                  actionListenerList, toRemove);
0162      } // End removeActionListener.
0163
0164      protected void notifyListeners( String message) {
0165         if ( actionListenerList != null) {
0166            actionListenerList.actionPerformed(
0167                  new ActionEvent( this,
0168                                  ActionEvent.ACTION_PERFORMED,
0169                                  message));
0170         } // End if.
0171      } // End notifyListeners.
0172
0173   } // End TurnCard.
```

These three methods maintain the list of listeners in the *actionListenerList* attribute and dispatch an ActionEvent, containing the *message* argument, to them in exactly the same manner as was explained in the previous chapter.

5.5 The *TurnCardDemo* artifact

This concludes the implementation of the *TurnCard* class. Figure 5.6 illustrates the *TurnCardDemo* artifact, constructed in order to give an indication that the first two classes in the hierarchy appear to be working correctly. The applet contains two intermediate JPanel instances, the upper one having as its single child a *TurnCard* instance. The lower panel contains five JButton instances, each of which is connected via the *TurnCardDemo*'s *actionPerformed()* method to the appropriate *TurnCard* method.

The code fragment that follows contains the declaration of the *TurnCardDemo* class as far as the end of the local declarations in its *init()* method. It is presented so as to allow techniques for the loading of Images to be illustrated.

```
0019  public class TurnCardDemo extends JApplet
0020                         implements ActionListener {
0021
0022  private TurnCard demoCard = null;
0023
0024     public void init() {
0025
0026     JPanel  upperPanel = new JPanel();
0027     JPanel  lowerPanel = new JPanel();
0028
0029     Image        theFront = null;
0030     Image        theBack  = null;
0031     MediaTracker tracker  = null;
0032
0033     JButton frontButton  =  null;
0034     JButton backButton   =  null;
0035     JButton flipButton   =  null;
0036     JButton turnButton   =  null;
0037     JButton removeButton =  null;
```

The *TurnCard* instance, *demoCard*, is declared as an attribute of the class as its methods have to be called from the *actionPerformed()* method, which is called as a consequence of the user pressing one of the five buttons. The two JPanels and five JButtons only need to be referenced during initialization, and so are declared as local variables of the method. The method also declares, on lines 0029 and 0030, two instances of the Image class. As will be explained, these resources will be loaded asynchronously on a separate thread of control and the *MediaTracker* instance, declared on line 0031, is supplied in order to allow their loading to be monitored and synchronized. The *init()* method continues as follows.

```
0039        this.setResources();
0040        tracker = new MediaTracker( this);
0041
0042        theFront = this.getToolkit().getImage(
0043                              "cards/signs/sign1.gif");
0044        theBack = this.getToolkit().getImage(
0045                              "cards/signs/back.gif");
0046
0047        tracker.addImage( theFront, 0);
0048        tracker.addImage( theBack,  1);
0049
0050        try {
0051            tracker.waitForAll();
0052        } catch ( InterruptedException exception) {
0053            // Do nothing.
0054        } // End try/catch.
0055
```

```
0056         if ( tracker.isErrorAny()) {
0057           System.err.println( Images could not be loaded ...);
0058           System.exit( -1);
0059         } // End if.
0060
0061         demoCard = new TurnCard( theFront, theBack);
```

The MediaTracker, called *tracker*, is constructed on line 0040 after the UIManager resources have been set on line 0039. Due to the inherent latency of Internet connections, which will be used to obtain them, the images will be loaded on a separate thread of control. A MediaTracker instance can be used to synchronize the loading process with the thread that is executing the *init()* method, as will be explained in detail below.

Object → java.awt.MediaTracker

```
public MediaTracker( Component toTrack)
```

Constructs a MediaTracKer to track the processes of loading resources for the component *toTrack*.

```
public void addImage( Image imageToTrack, int arbitaryID)
public void addImage( Image imageToTrack, int arbitaryID,
                      int width,          int height)
```

Adds the *imageToTrack* to the list of images whose loading (or loading and resizing) is being processed. The *arbitaryID* can be used to monitor the progress and duplicate values are allowed.

```
public boolean checkAll()
public boolean checkAll( boolean loadNow)
public boolean checkID(  int arbitaryID)
public boolean checkID(  int arbitaryID, boolean loadNow)
```

Methods to determine whether a particular image or all images have (**true**) or have not (**false**) completed loading. The *loadNow* argument can be used to initiate loading if it has not already commenced.

```
public void    waitForAll() throws InterruptedException
public boolean waitForAll( long milliseconds)
                            throws InterruptedException
public void    waitForID(  int arbitaryID)
                            throws InterruptedException
public boolean waitForID ( int arbitaryID, long milliseconds)
                            throws InterruptedException
```

Methods to wait for an image, or all images, to be loaded; either indefinitely or just for the period specified.

```
public boolean isErrorAny()
public boolean isErrorID(int arbitaryID)
```

Methods to determine whether the loading process completed successfully.

Lines 0042 to 0045 initiate the loading of the two Image resources making us of **this** applet's toolkit's *getImage()* method. The `sign1.gif` file contains the first of the UK traffic sign images, as illustrated in Figure 5.4, and the `back.gif` file contains an identically sized background image, as illustrated in the upper part of Figure 5.5. Having initiated the loading of the two images, lines 0047 and 0048 inform the *tracker* that it should monitor the loading processes. The two arguments to the call of the *tracker* addImage() method are the Image to monitor and an arbitrary integer identifier that can be used to identify the resource.

On line 0051 the loading of the images is synchronized with the applet's process by calling the *tracker*'s *waitForAll()* method. This call will block until all attempts to load the images that the tracker has been informed about have completed, or will throw an InterruptedException if, for any reason, the process is interrupted. Accordingly, the call of *waitForAll()* has to be contained within a **try/catch** structure which, in this implementation, does nothing in the exception handler.

It is possible that one or both of the images could not be loaded, possibly because the files that contain them do not exist or because the information they contain has become corrupt. The *tracker* isAnyError() method, called on line 0056, will return **true** if this is the case and will result in the program abending with a message on the standard error stream. The *MemoryGame* artifact, described later in this chapter will contain a more elegant technique for dealing with this problem.

The effect of this fragment is to ensure that the two images have been successfully loaded before the *init()* method terminates and the artifact subsequently becomes visible to the user. On line 0061 the two resources are used as arguments to the *TurnCard* constructor and the successful loading of the two Image resources can be seen in Figure 5.6. The remaining parts of the *init()* method construct and configure the five JButtons and then assemble the artifact as shown in Figure 5.6. Each button has **this** applet registered as its *ActionListener* which requires it to supply an *actionPerfomed()* method, implemented as follows.

```
0090    public void actionPerformed( ActionEvent event) {
0091
0092    String command = event.getActionCommand();
0093
0094       if ( command.equals( "front")) {
0095          demoCard.showFront();
0096       } else if ( command.equals( "back")) {
0097          demoCard.showBack();
0098       } else if ( command.equals( "flip")) {
0099          demoCard.flipOver();
0100       } else if ( command.equals( "turn")) {
0101          demoCard.turnOver();
0102       } else if ( command.equals( "remove")) {
0103          demoCard.remove();
0104       } // End if.
0105    } // End actionPerformed.
```

The *actionPerformed()* method will be called as a consequence of the user pressing one of the buttons. It is implemented as a five-way selection containing a branch for every button, each of which calls the corresponding *democard* method. The artifact can be used to demonstrate that the implementation of the *TurnCard* appears to be correct.

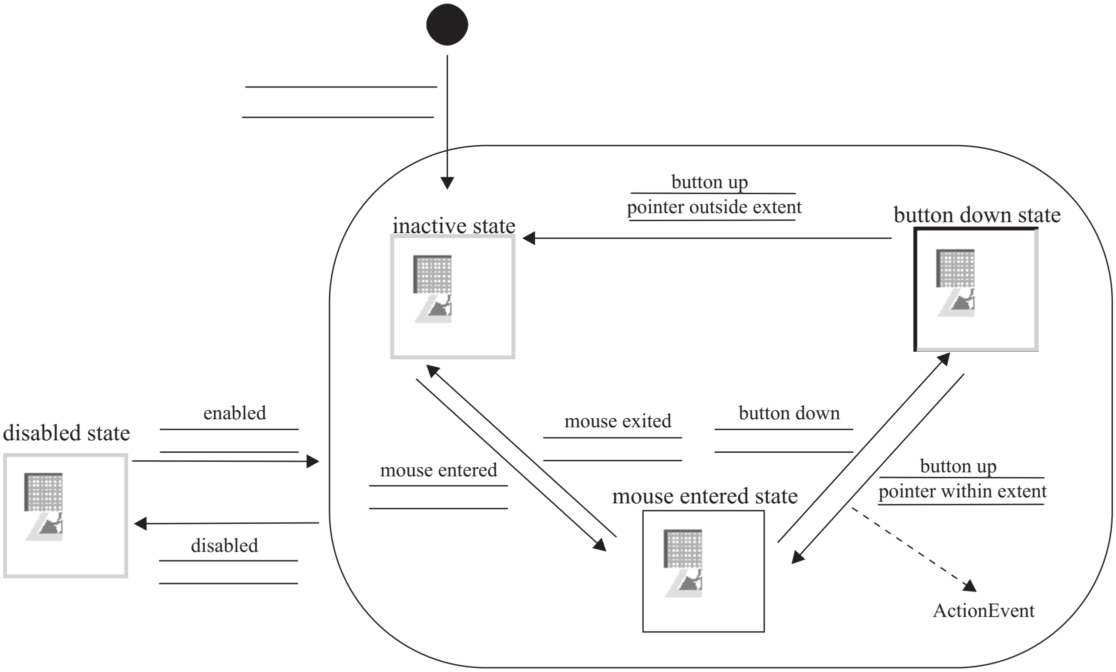


Figure 5.7 *ActiveCard* state transition diagram.

5.6 The *ActiveCard* class: design

The *ActiveCard* class introduces interactive behavior into the *cards* hierarchy; a state transition diagram for its required behavior is given in Figure 5.7. It shows that the card is initially posted in the *inactive* state. The representation of the card in this state shows a combination of the front, back and removed appearance to indicate that this behavioral state is independent from the logical state modeled by the *StateCard* and *TurnCard* class. That is, the interactive behavior supplied by this extension to the cards hierarchy is always applicable whatever face the card is showing, or even if the card is removed and so not showing either of its faces.

A novice design error would have been to combine the interactive and logical behavior, building the turning of the card directly into the interaction and preventing the behavior when the card was removed. This would not only complicate the construction of this class but would also limit its applicability. There may be some circumstances where a card should not be turned when it is clicked upon, and other circumstances where a removed card can be restored when it is clicked upon. Although these circumstances are difficult to foresee, good design does not restrict the potential reuse of a class by designing it specifically for its first intended use, but attempts to make it as general purpose as possible.

As will be demonstrated in the *MemoryGame* artifact, the turning of a card as a consequence of clicking on it when it is in its front or back state is a responsibility of the artifact's behavioral logic, as is the prevention of such behavior when the card is removed. The state transition diagram does show that a card in the *disabled state* will not respond to the user's interactions with it, but this is in accord with the behavior of any AWT or JFC component when disabled and so is an appropriate design decision. The transition to the disabled state can be effected from any of the other states. The transition from the disabled state will return the card to whatever interactive state is appropriate, as indicated by the termination of the transition arrow at the bounded state box.

From the *inactive* state the card will transit to the *mouse entered* state, where a different border is displayed to indicate this to the user, when the mouse pointer enters the component. The reverse transition occurs when the mouse pointer leaves the component. These two transitions will supply rollover behavior. That is, when there are a number of cards on a surface and the mouse is moving across them, the card which will react if the mouse button were pressed down is visually distinguished from all the others.

From the *mouse entered* state a button down event will cause a transition to the *button down* state, where a beveled border is shown. If the mouse button is released while the pointer is within the extent of the component the transition back to the *mouse entered* state is followed, and an ActionEvent is dispatched. However, if the mouse pointer is outside the extent of the component when the mouse button is released the transition to the *inactive* state is followed, and no ActionEvent is dispatched.

This behavior is similar to, but slightly different from, the behavior of an *ArrowBox* as shown in Figure 4.11. Both of these behaviors are also similar to, but slightly different from, the behavior of a standard JButton. This can be predicted to cause some users problems, as they might bring their expectations of general button behavior to these components. However, in practice this is not a significant problem as the appearance of and the context within which these specialized components appear are sufficiently distinct to visually indicate that they are not buttons. Also, people, in general, have flexible rather than rigid expectations of artifact behavior, and, providing they have sufficient visual cues, can rapidly adapt.

The class diagram for the *ActiveCard* class is given in Figure 5.8. The arguments to the constructor are the same as those of the *TurnCard* class constructor, as explained earlier in this chapter. The *setEnabled()* method overrides the method introduced by the AWT Component class and has the effect of making the instance unresponsive to the user's actions. The *paint()* method is required as this component has to redraw its border as it changes state. The protected *processMouseEvent()* method is required as this

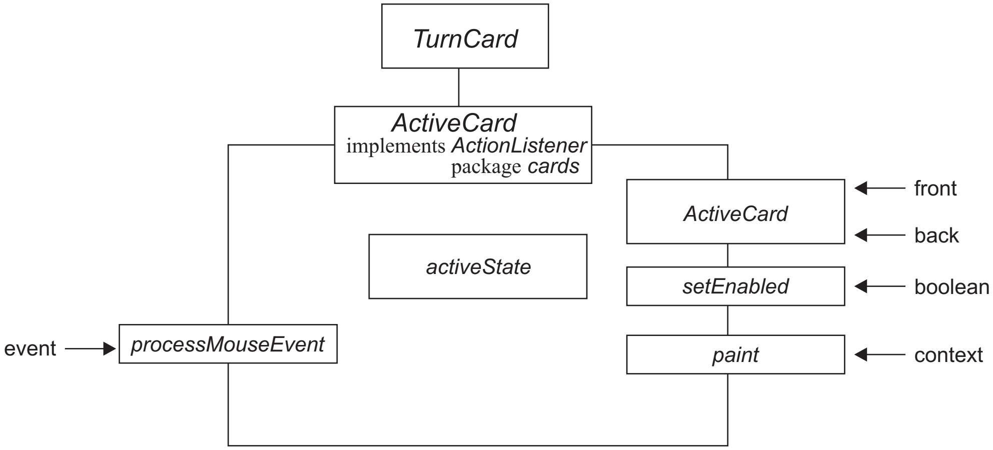


Figure 5.8 *ActiveCard* class diagram.

component will respond internally to mouse events in order to be responsive to the user's operation of the mouse, as was explained for the *ArrowBox* class in the previous chapter.

5.7 The *ActiveCard* implementation

The implementation of this class, as far as the start of the first constructor, is as follows.

```
0010  package cards;
0011
0012  import javax.swing.*;
0013  import javax.swing.border.*;
0014  import java.awt.*;
0015  import java.awt.event.*;
0016  import java.applet.*;
0017
0018
0019  public class ActiveCard extends TurnCard {
0020
0021  private final static int  INITIAL       = 0;
0022  private final static int  INACTIVE      = 1;
0023  private final static int  MOUSE_ENTERED = 2;
0024  private final static int  BUTTON_DOWN   = 3;
0025  private              int  activeState   = INITIAL;
0026
0027  private LineBorder  disabledBorder =
0028                           new LineBorder(  Color.gray,  2);
0029  private LineBorder  inactiveBorder =
0030                           new LineBorder(  Color.black, 2);
0031  private LineBorder  enteredBorder  =
0032                           new LineBorder(  Color.pink,   2);
0033  private BevelBorder downBorder     =
0034                           new BevelBorder( BevelBorder.LOWERED,
0035                                          Color.red, Color.pink);
```

Following the **package** statement and the necessary importations the class is declared on line 0019. Lines 0021 to 0024 declare **private** manifest names for the possible active states of the component and, on line 0025, the **int** variable *activeState* which will record the current state of the component. The remaining parts of this fragment declare and construct the four borders which will be used to indicate the interactive state of the component to the user. The choice of pink for the *enteredBorder* and red and pink for the *downBorder* is purely arbitrary, and in a more robustly engineered component these would be attributes with methods to set and query them. The constructor is implemented as follows.

```
0045     public ActiveCard( Image     front,
0046                        Image     back ){
0047        super( front, back);
0048        activeState = INACTIVE;
0049        this.setBorder( inactiveBorder);
0050        this.enableEvents( AWTEvent.MOUSE_EVENT_MASK);
0051     } // End ActiveCard constructor.
```

The constructor commences, on line 0047, by calling the **super**, *TurnCard*, constructor passing on the arguments. It then sets the *activeState* attribute to show that the component is *INACTIVE* and installs the appropriate border to indicate this to the user. Its last step, on line 0050, is to enable mouse events upon itself in order that they will be processed internally by the instance. The implementation of the class continues with the *setEnabled()* method, as follows.

```
0054     public void setEnabled( boolean yesOrNo) {
0055
0056        super.setEnabled( yesOrNo);
0057
0058        if ( yesOrNo) {
0059           switch ( activeState) {
0060           case INACTIVE:
0061              this.setBorder( inactiveBorder);
0062              break;
0063
0064           case MOUSE_ENTERED:
0065              this.setBorder( enteredBorder);
0066              break;
0067
0068           case BUTTON_DOWN:
0069              this.setBorder( downBorder);
0070              break;
0071           } // End switch.
0072        } else {
0073          this.setBorder( disabledBorder);
0074        } // End if.
0075        this.paintBorder( this.getGraphics());
0076     } // End setEnabled.
```

The method commences by calling the inherited, **super**, *setEnabled()* method passing on the argument *yesOrNo*. The substantive part of the method is a two-way decision containing a branch for setting the component enabled and one for setting the component disabled. The branch followed if the component is being enabled, on lines 0060 to 0070, consists of a **switch** structure containing a branch for each of the possible non-disabled states. Within each branch the appropriate border is installed into the component. The branch followed if the component is being disabled, on line 0073, installs the disabled border.

The final step of the method, on line 0075, is to make the changed border visible to the user by calling the *paintBorder()* method, passing as an argument the Graphics context obtained with a call of *getGraphics()*. As the only visible change to the component as it moves between the active states is the border, and as they are the same size, it is sufficient to redraw only the border; redrawing the whole component, with a call of repaint() is not required. The implementation continues with the *processMouseEvent()* method, as follows.

```
0079     protected void processMouseEvent( MouseEvent event) {
0080
0081     Border newBorder = null;
0082
0083        switch ( event.getID()) {
0084
0085        case MouseEvent.MOUSE_ENTERED:
0086             activeState = MOUSE_ENTERED;
0087             newBorder   = enteredBorder;
0088             break;
0089
0090          case MouseEvent.MOUSE_EXITED:
0091             activeState = INACTIVE;
0092             newBorder   = inactiveBorder;
0093             break;
0094
0095          case MouseEvent.MOUSE_PRESSED:
0096             activeState = BUTTON_DOWN;
0097             newBorder   = downBorder;
0098             break;
0099
0100          case MouseEvent.MOUSE_RELEASED:
0101             if ( this.contains( event.getX(),
0102                                 event.getY()) ){
0103                activeState = MOUSE_ENTERED;
0104                newBorder   = enteredBorder;
0105                if ( this.isEnabled()) {
0106                  this.notifyListeners( "clicked");
0107                } // End if.
0108             } // End if.
0109             break;
0110
0111          } // End switch.
0112
0113          if ( this.isEnabled()  &&
0114               newBorder != null ){
0115             this.setBorder(   newBorder);
0116             this.paintBorder( this.getGraphics());
0117          } // End if.
0118          super.processMouseEvent( event);
0119     } // End processMouseEvent.
```

The implementation of this method is very similar to the implementation of the *processMouseEvent()* method in the *ArrowBox* class. The substantive part of it contains a **switch** structure with a branch for each possible ID attribute of the *event* passed as an argument. Each of the first three branches responds to the event by changing the *activeState* of the component and preparing to install *newBorder*. The final branch, commencing on line 0100, is concerned with MOUSE_RELEASED events and will not only change to the *MOUSE_ENTERED* state but will also, on line 0106, if the component is not disabled and the pointer is within the extent of the component, fire an *ActionEvent* containing the *message clicked* to its listeners. This implementation of the *processMouseEvent()* method concludes by chaining its parent *processMouseEvent()* method, for reasons that will be explained in the next chapter.

Following the end of the **switch** structure, on lines 0113 to 0117, if the component is not disabled and a *newBorder* is required, it is installed and made visible to the user. The class concludes with the *paint()* method, as follows.

```
0121     public void paint( Graphics context) {
0122        this.paintBorder( context);
0123        super.paint( context);
0124     } // End paint.
0125
0126  } // End ActiveCard.
```

This method will be called by Java when the component needs repainting and commences by painting the border before calling its **super**, *TurnCard*, *paint()* method to fill in the rest of the component's window. A demonstration harness for the *ActiveCard* class will not be presented in detail. It would construct a single instance of the *ActiveCard* class and register itself as its only listener. The *actionPerformed()* method of the class might be implemented as follows.

```
0068     public void actionPerformed( ActionEvent event) {
0069
0070     String command = event.getActionCommand();
0071
0072        if ( command.equals( "clicked")) {
0073           demoCard.turnOver();
0074        } else if ( command.equals( "turning")) {
0075           demoCard.setEnabled( false);
0076        } else {
0077           demoCard.setEnabled( true);
0078        } // End if.
0079     } // End actionPerformed.
```

When the user clicks upon the *demoCard* it will dispatch an *actionEvent* whose *actionCommand* contains *clicked*; this will result, on line 0073, in the *demoCard*'s *turnOver()* method being called. As explained when the *turnOver()* method was described, its first step is to dispatch an *actionEvent* whose *actionCommand* contains *turning*; this will result, on line 0075, in the *demoCard* being disabled, preventing the user from interacting with it until it has finished turning. When it has turned, a third *actionEvent*, whose *actionCommand* contains *turned*, is dispatched, resulting, on line 0077, with the *demoCard* being re-enabled.

The user would see a single *TurnCard* whose border would change from black to pink as the mouse pointer enters it, turns red when the user presses down the mouse button and starts to turn when it is released within the card's extent. While the card turns, the border will be shown gray to indicate that it cannot be clicked upon and would restore the black or pink border, depending upon the position of the mouse pointer, when it finished turning.

The implementation of this method reinforces the comments made above that the card instance should just dispatch events to its listener indicating what is happening to it. The consequential actions of these messages, turning or flipping the card over and disabling and re-enabling it as required, are the responsibility of the environment that the card is being used within.

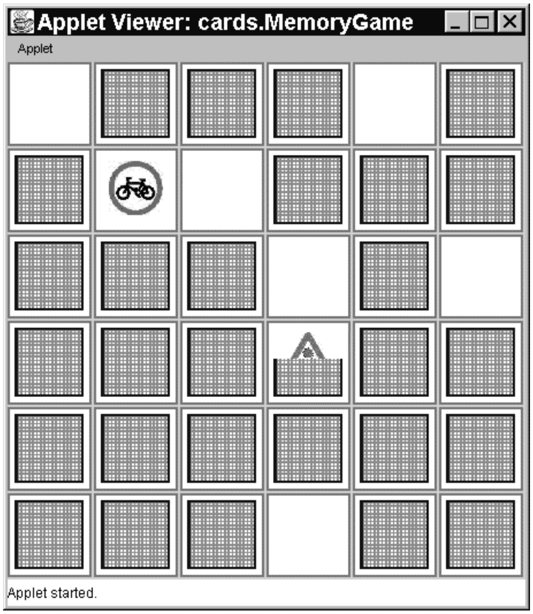


Figure 5.9 *MemoryGame* during play.

5.8 The *MemoryGame*: first implementation

The visual appearance of the first implementation of the *MemoryGame* artifact is given in Figure 5.9. It shows that, in this configuration, the game contains 18 pairs of cards arranged in a six by six grid. It also shows that at this point in the game three pairs have been removed from the playing area and that the current attempt is not going to reveal a fourth pair.

A state transition diagram for the game is given in Figure 5.10. It shows that the initial transition leads to the *no card turned* state with all cards showing their backs. A single transition leads from this state to the *one card turned* state; this transition is followed when the user clicks upon a (non-removed) card and results in the chosen card being turned over and subsequently disabled. The reason for disabling the card at this stage is to prevent the user from clicking upon the turned card and turning it back before a second card has been turned over. From the *one card turned* state the user can click upon a second card to turn it over and transit to the *two card turned* state.

The game remains in this state for a short period of time in order that the user can see the consequences of turning the second card; during this time all cards are disabled. All of the transitions leading from this state have *time out* as their event label, indicating that they will not be considered until the time period has expired. The upper of the pair of transitions leading back to the *no card turned* state is followed if the two turned cards do not match; it causes the turned cards to flip back over and all non-removed cards to be enabled. The lower transition is followed if the two turned cards do match but are not the penultimate pair in the game, and results in those two cards being removed and all non-removed cards becoming enabled. The third transition originating from the *two card turned* state leads to the terminal state and is followed when the penultimate pair of cards is matched. In this state all cards are disabled and show their fronts.

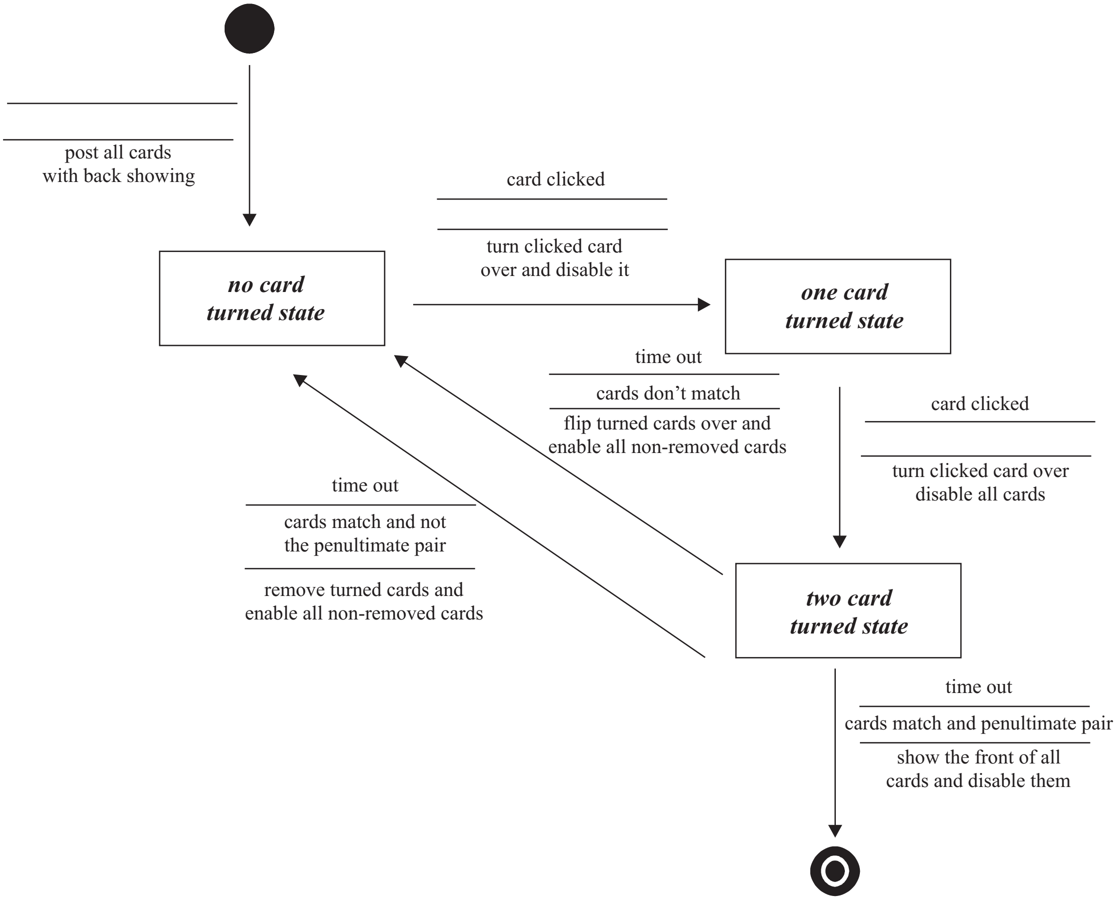


Figure 5.10 *MemoryGame* state transition diagram.

The implementation of this class, as far as the start of its *init()* method, is as follows.

```
public class MemoryGame extends JApplet
                     implements ActionListener {

private static final int NUMBER_OF_PAIRS = 18;
private static final int NUMBER_OF_CARDS = NUMBER_OF_PAIRS *2;

private ActiveCard cards[] = new ActiveCard[ NUMBER_OF_CARDS];

private static final String[] cardFileNames =
                  { "sign1.gif",  "sign2.gif",  "sign3.gif",
                    "sign4.gif",  "sign5.gif",  "sign6.gif",
                    "sign7.gif",  "sign8.gif",  "sign9.gif",
                    "sign10.gif", "sign11.gif", "sign12.gif",
                    "sign13.gif", "sign14.gif", "sign15.gif",
                    "sign16.gif", "sign17.gif", "sign18.gif"};


```

```
0035  private static final int INITIAL         = 0;
0036  private static final int NO_CARD_TURNED  = 1;
0037  private static final int ONE_CARD_TURNED = 2;
0038  private static final int TWO_CARD_TURNED = 3;
0039  private static final int FINISHED        = 4;
0040  private              int gameState       = INITIAL;
0041
0042  private ActiveCard firstCardTurned     = null;
0043  private ActiveCard secondCardTurned    = null;
0044  private int        numberOfPairsTurned = 0;
0045  private Timer      delay               = null;
```

The **package** and **import** statements have been omitted from the listing in the interests of brevity. The class is declared on lines 0018 and 0019 as implementing the *ActionListener* interface as it will have to receive the ActionEvents generated by the *ActiveCard*s that it contains. The *NUMBER_OF_PAIRS* of cards, and from it the *NUMBER_OF_CARDS*, are declared as manifest constants on lines 0021 and 0022. Line 0024 then uses the latter of these values to define the length of an array of *ActiveCard* called *cards*.

The declarations continue, on lines 0026 to 0032, with the declaration of an array of String constants containing the names of the 18 files that contain the images that will be used for the fronts of the cards. The manifest states of the game are then defined on lines 0035 to 0039 and the variable to record the current state, *gameState*, on line 0040. The next two attributes, *firstCardTurned* and *secondCardTurned*, will be used to record the identity of the two cards as they are turned. The declarations conclude, on lines 0044 to 0045, with an **int** variable to record the *numberOfPairsTurned* and a Timer instance to provide the *delay* while the game is in the *two cards turned* state. The class's implementation continues with its *init()* method, as follows.

```
0048     public void init() {
0049
0050        this.prepareCards();
0051        this.shuffleCards();
0052
0053        this.getContentPane().setLayout(
0054                              new GridLayout( 6, 6, 2, 2));
0055        for (int index =0; index < cards.length; index++) {
0056           this.getContentPane().add( cards[ index]);
0057        } // End for.
0058        delay = new Timer( 1000, this);
0059        delay.setRepeats( false);
0060        gameState = NO_CARD_TURNED;
0061     } // End init.
```

The method commences, on lines 0050 and 0051, by calling two **private** methods to prepare the cards and then to shuffle them; the implementation of these methods will be described shortly. With the cards available in the *cards* array, a six row by six column GridLayout is established and then each card from the *cards* array is added to the applet's *contentPane*. The next two lines, 0058 and 0059, construct the *delay* Timer so that when it is started it waits one second before sending a single event to **this** applet. The final step in the *init()* method is to record, in *gameState*, that it is in the *NO_CARD_TURNED* state.

The first part of the implementation of the *prepareCards()* method is as follows.

```
0121    private void prepareCards() {
0122
0123    MediaTracker tracker   = new MediaTracker( this);
0124    Image        theBack   = null;
0125    Image        thePics[] = new Image[ NUMBER_OF_PAIRS];
0126
0127       theBack = this.getToolkit().getImage(
0128                                "cards/signs/back.gif");
0129       tracker.addImage( theBack, 19);
0130
0131       for (int index =0; index < thePics.length; index++) {
0132          thePics[ index] = ( this. getToolkit().getImage(
0133                                  "cards/signs/" +
0134                                  cardFileNames[ index]));
0135          tracker.addImage( thePics[ index], index);
0136       } // End for.
0137       try {
0138           tracker.waitForAll();
0139       } catch ( InterruptedException exception) {
0140         // do nothing.
0141       } // End try/catch.
0142
0143       if ( tracker.isErrorAny()) {
0144         System.err.println("Error loading images, abending!");
0145         System.exit( -1);
0146       } // End if.
```

This part of the method loads the image for the back of the cards and the 18 images for the front of the cards from the location where the applet was obtained. It uses the same techniques, involving a *MediaTracker* instance, as was described for the *turnCardDemo* artifact above. Assuming that all of the resources can be obtained, on line 0147 the Image array *thePics* will contain the front images and *theBack* will contain the background image. The conclusion of the *prepareCards()* method is as follows.

```
0148       for (int index =0; index < cards.length; index++) {
0149          cards[ index] = new ActiveCard(thePics[ index/2],
0150                                                  theBack);
0151          cards[ index].setName( cardFileNames[ index/2]);
0152          cards[ index].showBack();
0153          cards[ index].addActionListener( this);
0154       } // End for.
0155    } // End prepareCards.
```

This part of the method constructs the 36 cards, configures them and stores their identities in the *cards* array. Each card is constructed in turn within the definite iteration on lines 0148 to 0154. The first argument to the constructor specifies an image from the 18-element array *thePics*. The division by 2 of the loop *index* will ensure that adjacent cards contain the same image. This same expression is used on line 0151, where each card has its name attribute established. The effect is that adjacent cards in the *cards* array not only have the same front image but also have the same name. Line 0152 ensures that the cards will show their backs when they are first visible and line 0153 registers **this** applet as their *ActionListener* resource.

As soon as the *prepareCards()* method called from *init()* has finished, the *shuffleCards()* method, implemented as follows, is called. In this method each card in turn is swapped with another card chosen at random from the array.

```
private void shuffleCards() {

int        swapWith;
ActiveCard swapCard = null;

   for (int index =0; index < cards.length; index++) {
      swapWith = (int) (Math.random() * NUMBER_OF_CARDS);
      swapCard = cards[ swapWith];
      cards[ swapWith] = cards[ index];
      cards[ index] = swapCard;
   } // End for.
} // End shuffleCards.
```

When the *init()* method concludes the 36 cards will become visible to the user, who will initiate the game by clicking upon one of them. The card will dispatch an ActionEvent to the applet's *actionPerformed()* method, which has the responsibility of responding to the user's actions to implement the game. A card will send an event whose actionCommand contains *clicked* when the user interacts with it, additionally it will send one containing *turning* as it starts to turn and one containing *turned* as it finishes turning. The *actionPerformed()* method will also be receiving events that originated from the Timer, whose *actionCommand* attribute is **null**.

These four possible types of event may be received when the game is in any one of its three playing states: *NO_CARD_TURNED*, *ONE_CARD_TURNED* and *TWO_CARD_TURNED*. This gives a structure to the *actionPerformed()* method for deciding what caused the event to be dispatched and within that decision deciding what consequences are appropriate for the state the card is in. Table 5.1 illustrates the relationships between the possible events received and the state of the game when they are received.

A table such as this is known as a *state/event table* and provides a means of proceeding from a state transition diagram to an event listener method. The contents of the table are derived from the STD and validated against it with a run-through of playing the game. For example, the following narrative shows how the table can be used to show how that part of

Table 5.1 *MemoryGame* state/event table.

		Event			
		Clicked	Turning	Turned	Time out
State	NO_CARD_TURNED	Set all cards disabled and start turning	Ignore	Transit to ONE_CARD_TURNED and enable playable cards	Not possible
	ONE_CARD_TURNED	Set all cards disabled and start turning	Ignore	Transit to TWO_CARD_TURNED and start timer	Not possible
	TWO_CARD_TURNED	Not possible	Not possible	Not possible	Decide on match and transit to NO_CARD_TURNED or FINISHED_TURNED

the behavior required by the STD will be satisfied, if the user starts playing the game and immediately manages to get a match.

> The game starts in the *NO_CARD_TURNED* state, where the user clicks on a card which causes a *clicked* event to be received. This will disable all the cards and cause the card clicked upon to start turning, causing a *turning* event to be received which can be ignored. When the card has finished turning, a *turned* event causes a transition to the *ONE_CARD_TURNED* state and enables all cards that are showing their backs. The user will then click upon a second card, which causes a second *clicked* event which, as before, causes a *turning* event and then a *turned* event, upon which the transition to the *TWO_CARD_TURNED* state occurs and the timer is started. When the *time out* event is received in the *TWO_CARD_TURNED* state the cards are checked and, if they match are removed, before the game transits back to the *NO_CARD_TURNED* state.

The *state/event table* can be used to construct the outline of the listener method that implements the logic. The cells labeled *not possible* indicate that an event of the type indicated should never be received while the game is in the state identified, so these possibilities need not be considered in the program code. Likewise, the cells labeled *ignore* can be omitted from the logic. This allows the following outline logic of the *MemoryGame actionPerformed()* method to be produced, which can subsequently be populated with the necessary actions.

```
if clicked event
   if NO_CARD_TURNED state
      disable all cards
      turn first card
   else if ONE_CARD_TURNED state
      disable all cards
      turn second card
   end if
else if turned event
   if NO_CARD_TURNED state
      enable all playable cards
      transit to the ONE_CARD_TURNED state
   else if ONE_CARD_TURNED state
      transit to TWO_CARD_TURNED state
      start timer
   end if
else if time out event
   if cards match
      remove matched cards
   else
      turn unmatched cards back over
   end if
   if all but 1 pairs matched
      transit to FINISHED state
   else
      enable all playable cards
      transit to NO_CARD_TURNED state
   end if
end if.
```

Taking this as a template, the implementation of the first branch of the outermost **if** structure is as follows.

```
0061    public void actionPerformed( ActionEvent event) {
0062
0063    String command = event.getActionCommand();
0064
0065       if ( command == null) {
0066          command = new String( "time-out");
0067       } // End if.
0068
0069       if ( command.equals( "clicked")) {
0070
0071          if ( gameState == NO_CARD_TURNED) {
0072             this.setAllCardsDisabled();
0073             firstCardTurned = (ActiveCard) event.getSource();
0074             firstCardTurned.turnOver();
0075          } else if ( gameState == ONE_CARD_TURNED) {
0076             this.setAllCardsDisabled();
0077             secondCardTurned = (ActiveCard) event.getSource();
0078             secondCardTurned.turnOver();
0079          } // End if.
```

The implementation of this method commences, on lines 0065 to 0068, with a check to see if the actionCommand resource of the *event* received is **null**. This indicates that the *event* originated from the *delay* timer and the local variable *command* is initialized to a String containing *time-out*, in order to simplify subsequent processing.

The *turnOver()* method calls on lines 0074 and 0078 will result in the cards dispatching an ActionEvent whose actionCommand attribute contains *turning*, which is ignored and, when they finish turning, a second ActionEvent whose actionCommand attribute contains *turned*, which will be responded to by the second branch of the outermost **if** structure as follows.

```
0081      } else if ( command.equals( "turned")) {
0082
0083          if ( gameState == NO_CARD_TURNED) {
0084             this.setAllPlayableEnabled();
0085             gameState = ONE_CARD_TURNED;
0086          } else if ( gameState == ONE_CARD_TURNED) {
0087             gameState = TWO_CARD_TURNED;
0088             delay.start();
0089          } // End if.
```

The first branch of the inner **if** structure will be executed when the first card that the user chooses has finished turning and causes the game to transit to the *ONE_CARD_TURNED* state with all the playable cards enabled so that the user can choose a second card. The second branch will be executed when the second card has finished turning and transits to the *TWO_CARD_TURNED* state before the *delay*.start() method is called on line 0088. This will result in a subsequent ActionEvent being dispatched from the Timer, as explained above the actionCommand of the *event* received is **null** and this has been replaced with *time-out* at the start of the method. The implementation of the third branch of the outermost **if** structure is as follows.

```
0091       } else if ( command.equals( "time-out")) {
0092
```

```
0093          if ( firstCardTurned.getName().equals(
0094                            secondCardTurned.getName()) ){
0095             firstCardTurned.remove();
0096             secondCardTurned.remove();
0097             numberOfPairsTurned++;
0098          } else {
0099             firstCardTurned.showBack();
0100             secondCardTurned.showBack();
0101          } // End if.
0102
0103          if ( numberOfPairsTurned == NUMBER_OF_PAIRS-1) {
0104             gameState = FINISHED;
0105             this.setFinishedState();
0106          } else {
0107             gameState = NO_CARD_TURNED;
0108             this.setAllPlayableEnabled();
0109          } // End if.
0110       } // End if.
0111    } // End actionPerformed.
```

When the event is received from the Timer the first of the nested **if** structures, commencing on line 0093, decides whether the names (and hence the front images) of the two turned cards match. If they match, lines 0094 to 0095 remove the two cards and increment the *numberOfPairsTurned* attribute. Otherwise, they do not match and lines 0099 and 0100 turn the cards back onto their back.

The following second level **if** structure, commencing on line 0103, decides whether the game has finished. This occurs when all but one of the pairs have been matched, as the two remaining cards must be a matching pair, and causes the game to transit to the *FINISHED* state on lines 0107 and 0108. Otherwise the game has to continue, so lines 0107 and 0108 transit back to the *NO_CARD_TURNED* state and re-enable the playable cards.

> *The logic contained within the two levels of nested* **if** *structures in this method implements the game as described in the rules, the written description, and STD. Human cognition is only just about capable of simultaneously maintaining all the ramifications of the decision structures involved in a sequence of this complexity. Hence the use of the state/event table as an intermediate representation of the logic is advisable if debugging of the artifact is to be minimized.*

The implementations of the *setAllCardsDisabled()* and *setAllPlayableEnabled()* methods are as follows.

```
0113    private void setAllCardsDisabled() {
0114       for (int index =0; index < cards.length; index++) {
0115             cards[ index].setEnabled( false);
0116       } // End for.
0117    } // End setAllCardsDisabled.
0118
0119    private void setAllPlayableEnabled() {
0120       for (int index =0; index < cards.length; index++) {
```

```
          if ( cards[ index].isBackShowing()) {
             cards[ index].setEnabled( true);
          } else {
             cards[ index].setEnabled( false);
          } // End if.
       } // End for.
    } // End setAllPlayableEnabled.
```

The *setAllCardsDisabled()* method iterates through all the cards in the *cards* array and calls their *setEnabled()* method with the argument **false**. The *setAllPlayableEnabled()* method is similar. It enables all cards that have their back showing and ensures that all others (those removed, those with their front showing and those that might be turning) are disabled. The only other method is *setFinishedState()*, implemented without comment as follows.

```
    private void setFinishedState() {
       for (int index =0; index < cards.length; index++) {
         cards[ index].showFront();
         cards[ index].setEnabled( false);
      } // End for.
    } // End setFinishedState.
```

5.9 The *MemoryGame*: second implementation

The first implementation of the memory game provided the essential game-playing routines but is not yet in an acceptable form. The first problem with it is that there can be a very noticeable delay as the game starts, while the images are loading. During this time the user has no indication of what is happening or how long they might have to wait until the game can commence. Due to the inherent latency of Internet connections nothing can be done to speed up the loading of the images; instead some indication should be given to the user of what is causing the delay and how much progress has been made.

The second problem concerns any errors with loading the images. In the first version this resulted in a message being output on the standard error stream and the applet finishing. In the context of a Web browser this will leave the area reserved for the game still visible on the page with no indication of what has happened. What is required is a message to users, in the game's area, informing them of what caused the problem and what they might be able to do about it. The third problem is that once the game has been completed there is no way of restarting it without restarting the applet. This will cause the images to be reloaded, resulting in another unacceptable and unnecessary delay. A means of restarting the game from its finished state is required.

An additional problem, which will not be obvious to the user, concerns the software architecture of the implementation. The first version of the *MemoryGame* is architecturally no more than a demonstration harness and test bed. It allows the behavior of the game, and of the code that implements the STD, to be investigated but is essentially a one-layer implementation. The revised version will provide a more sophisticated and extensible architecture, as shown in the instance diagram in Figure 5.11.

The diagram shows that the *memoryGame* makes use of two presentation objects, an instance of *MemoryGameOpeningScreen*, called *openingScreen*, which it shows while the card images are being loaded and the cards prepared. The second presentation object is an instance of *MemoryGameArea*, called *gameArea*, upon which it displays the state of the game and from which it receives events informing it of the user's actions.

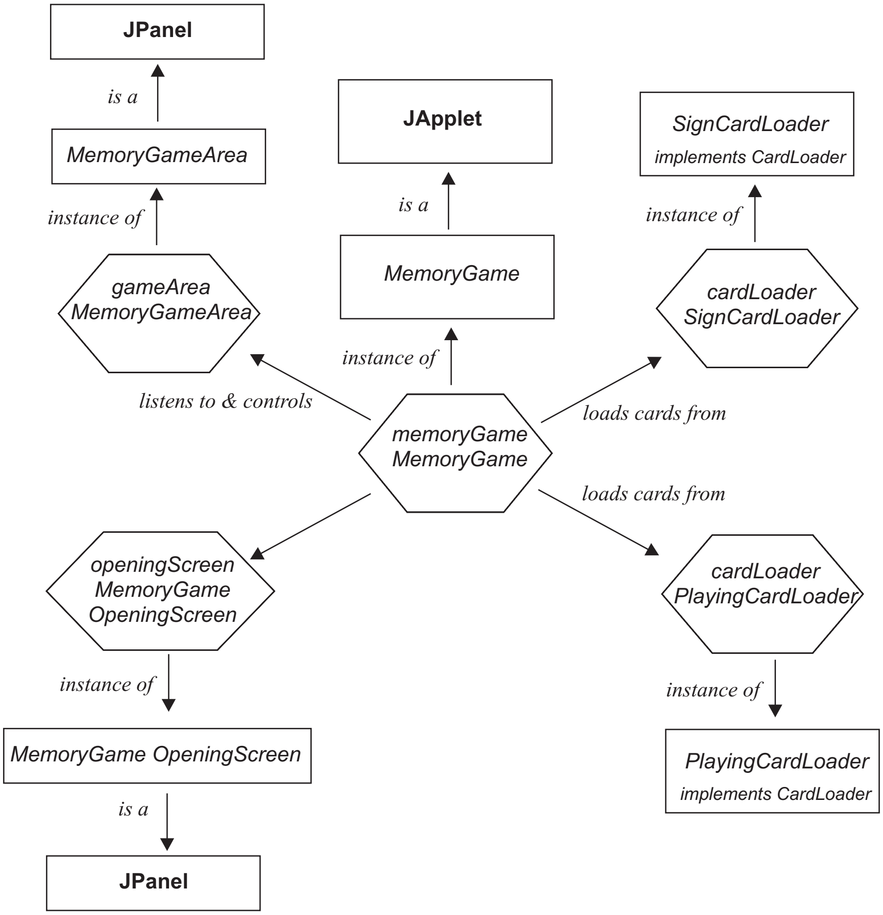


Figure 5.11 *MemoryGame* instance diagram.

The *memoryGame* also makes use of one of two application-level objects which load the images and construct the cards. The *SignCardLoader* instance will supply the 36 road signs, as used in the first version of the game. The *PlayingCardLoader* will supply 48 of the 52 cards from a standard pack of playing cards. The cards showing the 10 of clubs, spades, hearts and diamonds have been omitted in order that a more convenient 6 by 8 game board, rather than the 4 by 13 layout that a full pack would require, can be used. Figure 5.12 shows the revised *memoryGame* using the playing cards. Both of the objects which load the cards are instances of classes which implement the *CardLoader* interface. This interface is contained within the *cards* package and has been provided in order to facilitate the provision of additional sets of cards.

Figure 5.13 shows the *MemoryGameOpeningScreen*; at the top of the screen is a message informing the user of what is causing the delay and at the bottom an instance of the JProgressBar class is being used to indicate the proportion of images that have loaded and how many are still to load. The screen also installs a *busy* cursor (in MS Windows environments this is an egg timer cursor) to show that the system will not be able to respond to any user's actions.

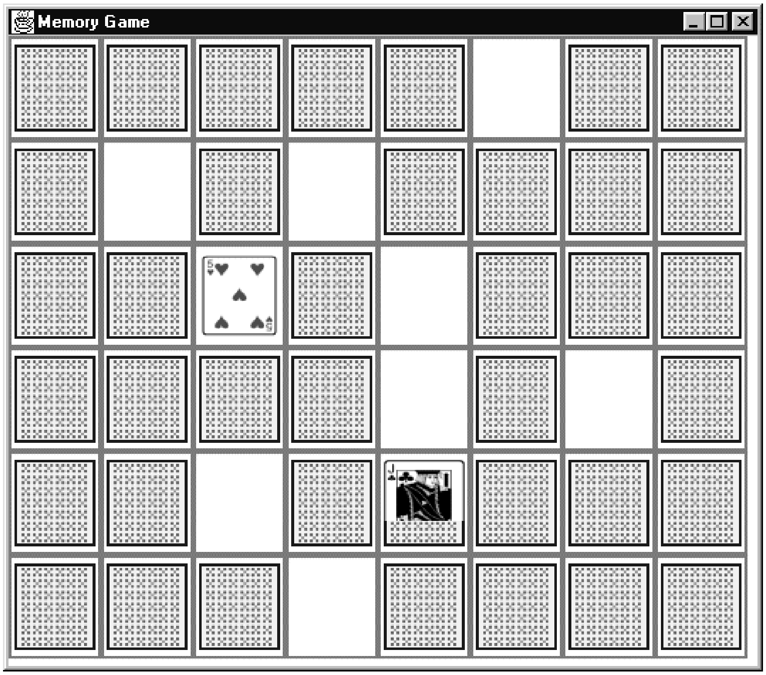
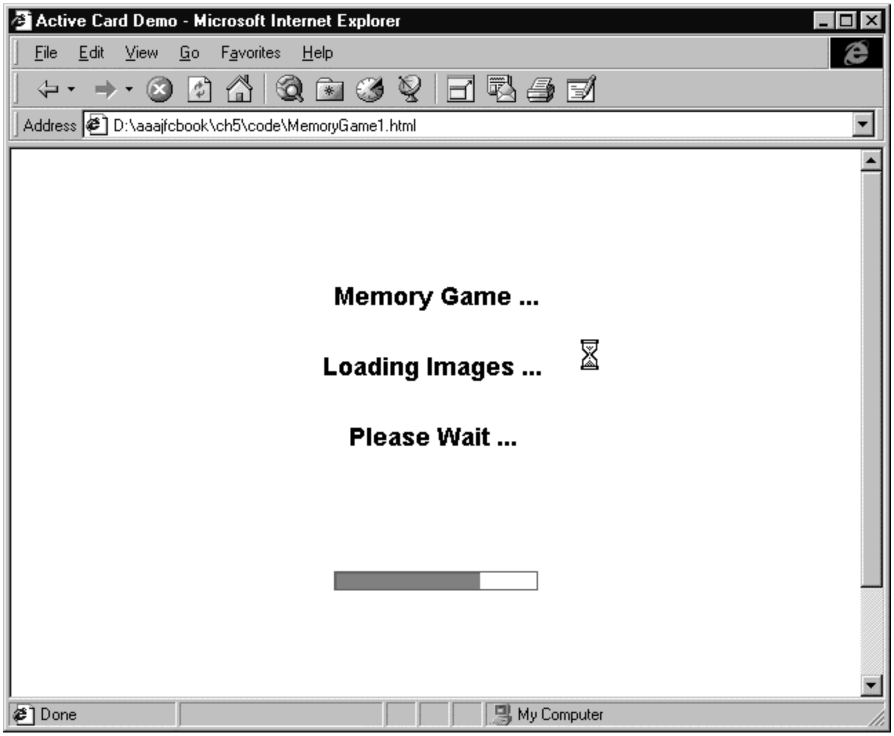


Figure 5.12 *MemoryGame* in the playing card configuration.


Active Card Demo - Microsoft Internet Explorer
File Edit View Go Favorites Help
Address D:\aaajfcbook\ch5\code\MemoryGame1.html
Done
My Computer

Memory Game ...

Loading Images ...

Please Wait ...

Figure 5.13 *MemoryGame*: initial appearance.

The *MemoryGame*'s opening screen, as shown in the instance diagram in Figure 5.11, is supplied by an instance of the *MemoryGameOpeningScreen* class which has a default constructor. It also has a single public method, called *updateProgress()*, which takes an argument indicating the proportion of images which are known to have loaded and updates the state of the progress bar accordingly.

The *MemoryGameOpeningScreen* class extends the JPanel class; full details of its constructor will not be given. Essentially, it constructs the message from three JLabel instances and installs them, and the other components, into a BoxLayout manager. The JProgressBar instance, called *indicatorBar*, is constructed, configured and added to the screen in the following fragment.

```
0056         indicatorBar = new JProgressBar();
0057
0058         progressPanel.add( Box.createGlue());
0059         progressPanel.add( indicatorBar);
0060         progressPanel.add( Box.createGlue());
0061
0062         this.add( progressPanel);
0063         this.setCursor( Cursor.WAIT_CURSOR);
```

The use of the default JProgessBar constructor, on line 0056, ensures that it has a horizontal layout with minimum and maximum values of 0 and 100. The remaining parts of this constructor show that the *indicatorBar* is mounted on its own *progressPanel*, sandwiched in a horizontal BoxLayout between two glue instances. This positions the *indicatorBar* in the central third of the available width, below the message.

JComponent → javax.swing.JProgressBar

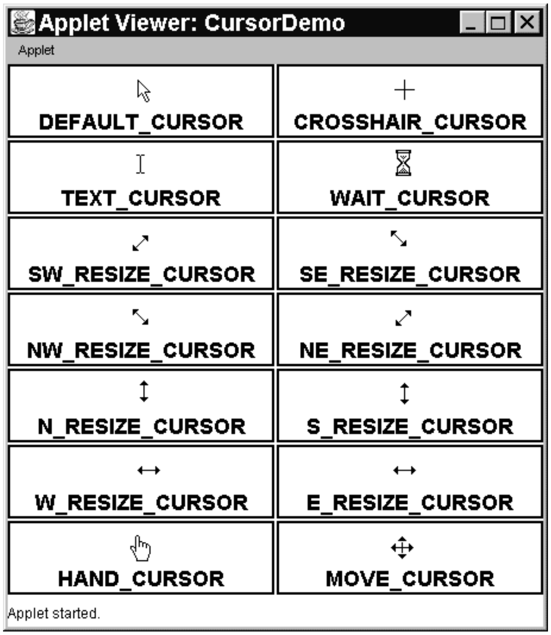
```
public JProgressBar()
public JProgressBar( int orientation)
public JProgressBar( int minimum, int maximum)
public JProgressBar( int orientation, int minimum, int maximum)
```

Constructs a JProgressBar with *minimum* (default 0), *maximum* (default 0) and *orientation*, either JProgressBar.HORIZONTAL (default) or JProgressBar.VERTICAL, as specified.

```
public int     getMinimum()
public int     getMaximum()
public int     getOrientation()
public void    setValue( int newValue)
public int     getValue()
public double  getPercentageComplete()
```

Inquiry methods on the three principal attributes. The *value* attribute (default *minimum*) is interpreted relative to *minimum* and *maximum* and can be obtained directly or as a percentage (0.0 to 1.0) value.

The constructor also, on line 0064, installs a busy cursor into the panel so that when the mouse pointer is over it it will indicate to the user that the system is unresponsive and cannot be interacted with. The names and appearance of the pre-defined cursors are shown in Figure 5.14.



Applet started.

Figure 5.14 Standard cursors in Windows.

Object → java.awt.Cursor

```
public Cursor(  int    manifestName)
public Cursor ( String name)
```

Constructs a cursor using one of the pre-supplied manifest names (see below) or the name of a developer-supplied cursor.

```
CROSSHAIR_CURSOR  CUSTOM_CURSOR  DEFAULT_CURSOR  E_RESIZE_CURSOR
HAND_CURSOR  MOVE_CURSOR  N_RESIZE_CURSOR  NE_RESIZE_CURSOR
NW_RESIZE_CURSOR  S_RESIZE_CURSOR  SE_RESIZE_CURSOR  SW_RESIZE_CURSOR
TEXT_CURSOR  W_RESIZE_CURSOR  WAIT_CURSOR
```

The *manifestNames* of the pre-supplied Cursors.

Object → java.awt.Toolkit

```
public Cursor createCustomCursor( Image cursorImage, Point hotSpot,
                                  String cursorName)
                                   throws indexOutOfBoundsException
```

Creates a Cursor from the resources specified and installs it into the Toolkit for possible later use.

java.awt.Component

```
public setCursor( Cursor toShow)
```

Installs the cursor *toShow* when the Component has mouse focus.

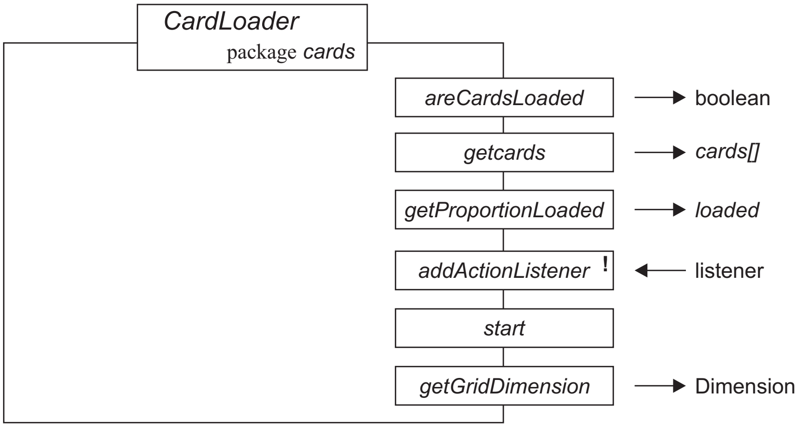


Figure 5.15 *CardLoader* interface class diagram.

The only *MemoryGameOpeningScreen* method is *updateProgress()* which is implemented as a wrapper on the *indicatorBar* setValue() method, as follows. The effect of calling the method is to cause the visual appearance of the bar to change, indicating the value of the argument passed to it.

```
public void updateProgress( int proportionLoaded) {
    indicatorBar.setValue( proportionLoaded);
} // End updateProgress.
```

The class diagram for the *CardLoader* **interface** is given in Figure 5.15. It shows that any class implementing the interface must supply five methods. The first, *areCardsLoaded()*, is an inquiry method indicating whether the cards have, or have not, been loaded. The *getCards()* method will return **null** if the cards have not, or cannot, be loaded and *getProportionLoaded()* will return a value between 0 and 100 indicating how many are currently loaded. The *addActionListener()* method indicates that *CardLoader* instances are unicaster ActionEvent sources. ActionEvents will be dispatched by the instances to inform the *memoryGame* of the progress in loading the cards and finally to confirm that the attempt to load them has finished.

The start() method suggests that they will be instances of a class that extends the Thread class. This is necessary, as the loading of the cards must take place on a separate thread of control in order that the principal thread can periodically update the opening screen's progress bar. The final method is purely for convenience and indicates, in the Dimension attributes, the preferred layout of the cards. The implementation of the interface is as follows.

```
package cards;

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

public interface CardLoader {

```

```
0018     public boolean areCardsLoaded();
0019
0020     public ActiveCard[] getCards();
0021
0022     public int getProportionLoaded();
0023
0024     public void addActionListener( ActionListener listener)
0025                           throws TooManyListenersException;
0026
0027     public void start();
0028
0029     public Dimension getGridDimension();
0030
0031  } // End CardLoader interface.
```

The two classes which implement the *CardLoader* interface are *SignCardLoader* and *PlayingCardLoader*. Only the *PlayingCardLoader* class will be described in detail; the implementation of the *SignCardLoader* class is essentially identical. The header of the *PlayingCardLoader* class, as far as the start of its constructor, is as follows.

```
0018  public class PlayingCardLoader extends Thread
0019                               implements CardLoader {
0020
0021  private ActionListener itsListener    = null;
0022  private Applet         itsApplet      = null;
0023  private int            numberLoaded   = 0;
0024  private boolean        problemLoading = false;
0025  private ActiveCard     cards[]        = null;
0026
0027  private static final String    pcardBack = "back.gif";
0028  private static final String    pcardDir  = "cards/pcards/";
0029  private static final Dimension pcardGrid = new Dimension(6, 8);
0030
0031  private static final String[] pcardFilenames = {
0032  "aceclubs.gif", "acedimnd.gif", "aceheart.gif", "acespade.gif",
0033  "2clubs.gif",   "2dimnd.gif",   "2hearts.gif",  "2spade.gif",
0034  "3clubs.gif",   "3dimnd.gif",   "3hearts.gif",  "3spade.gif",
0035  "4clubs.gif",   "4dimnd.gif",   "4hearts.gif",  "4spade.gif",
0036  "5clubs.gif",   "5dimnd.gif",   "5hearts.gif",  "5spade.gif",
0037  "6clubs.gif",   "6dimnd.gif",   "6hearts.gif",  "6spade.gif",
0038  "7clubs.gif",   "7dimnd.gif",   "7hearts.gif",  "7spade.gif",
0039  "8clubs.gif",   "8dimnd.gif",   "8hearts.gif",  "8spade.gif",
0040  "9clubs.gif",   "9dimnd.gif",   "9hearts.gif",  "9spade.gif",
0041  "jackclub.gif", "jackdmnd.gif", "jackhrts.gif", "jackspad.gif",
0042  "kingclub.gif", "kingdmnd.gif", "kinghrts.gif", "kingspad.gif",
0043  "quenclub.gif", "quendmnd.gif", "quenhrts.gif", "quenspad.gif"}
```

The declaration of the class, on lines 0018 and 0019, indicates that it extends the Thread class and implements the *CardLoader* interface, for reasons described above. The first two declarations on lines 0021 and 0022 are the identity of *itsListener*, to send ActionEvents to, and the identity of *itsApplet*, in order to obtain the toolkit to load the images with. The *numberLoaded* **int**eger attribute and the *problemLoading* flag are used to monitor the loading of the images and the *cards* array will contain the *ActiveCard* instances, if they can be successfully loaded.

The remaining class-wide declarations, on lines 0027 to 0043, define the location and filenames of the 48 cards and the image to be used for the backs of the cards, as well as the suggested dimensions for laying them out. The implementation continues with the constructor and five of the methods mandated by the *cardLoader* interface, as follows.

```
0045     public PlayingCardLoader( Applet applet) {
0046        super();
0047        itsApplet = applet;
0048     } // End PlayingCardLoader constructor.
0049
0050     public boolean areCardsLoaded() {
0051        return cards != null;
0052     } // End areCardsLoaded.
0053
0054     public ActiveCard[] getCards() {
0055        return cards;
0056     } // End getCards.
0057
0058     public int getProportionLoaded() {
0059        return (int) (((float) numberLoaded /
0060                       (float) pcardFilenames.length) * 100.0);
0061     } // End if.
0062
0063     public void addActionListener( ActionListener listener)
0064                              throws TooManyListenersException {
0065        if ( itsListener == null) {
0066           itsListener = listener;
0067        } else {
0068           throw new TooManyListenersException();
0069        } // End if.
0070     } // End addActionListener.
0071
0072     public Dimension getGridDimension() {
0073        return pcardGrid;
0074     } // End getGridDimension.
```

The constructor, on lines 0045 to 0048, need only store the identity of the *applet* argument in *itsApplet* attribute, after calling the super, Thread, constructor. Three of the inquiry methods, *areCardsLoaded()*, getCards() and *getGridDimension()* are straightforward, returning information about the state of the attributes and the *addActionListener()* method needs only to store the *listener* argument into the *itsListener* attribute if no listener is already registered or throw a suitable exception otherwise. The *getProportionLoaded()* method uses the *numberLoaded* attribute and the length of the *pcardFilenames* array to compute the proportion of images which have currently loaded.

This sixth method mandated by the *CardLoader* interface, *start()*, need not be explicitly supplied by the class, as it is inherited from the Thread class. However, it is expected that the Thread run() method will be overridden and implement the process of loading the images and constructing the cards. The *PlayingCardLoader run()* method commences as follows.

```
0078     public void run() {
0079
0080     MediaTracker tracker   = new MediaTracker( itsApplet);
0081
0082     Image  theBack       = null;
0083     Image  thePics[]     = new Image[ pcardFilenames.length];
0084     int    numberCounted = 0;
0085
0086        numberLoaded   = 0;
0087        problemLoading = false;
0088
0089        theBack = itsApplet.getToolkit().getImage(
0090                                       pcardDir + pcardBack);
0091        tracker.addImage( theBack,  49);
0092
0093        for (int index =0; index < thePics.length; index++) {
0094           thePics[ index] = itsApplet.getToolkit().getImage(
0095                                          pcardDir +
0096                                          pcardFilenames[ index]);
0097           tracker.addImage( thePics[ index], index);
0098        } // End for.
```

This part of the method is comparable with the corresponding part of the original *MemoryGame* implementation. After setting the *numberLoaded* attribute to zero and *problemLoading* flag to **false**, on lines 0086 and 0087, it initiates the loading of all the images and informs the tracker of them. The method continues as follows.

```
0100        while ( numberLoaded < thePics.length &&
0101                ! problemLoading               ){
0102
0103           numberCounted = 0;
0104           for (int index =0; index < thePics.length; index++) {
0105              if ( tracker.checkID( index, true)) {
0106                 numberCounted++;
0107              } // End if.
0108           } // End for.
0109
0110           if ( tracker.isErrorAny()) {
0111              problemLoading = true;
0112           } else {
0113
0114              if ( ( numberCounted > numberLoaded) &&
0115                   ( itsListener   != null       ) ){
0116                 numberLoaded = numberCounted;
0117                 itsListener.actionPerformed(
0118                          new ActionEvent( this,
0119                              ActionEvent.ACTION_PERFORMED,
0120                              "loading"));
0121              } // End if.
0122
0123              synchronized( this) {
0124                 try {
0125                    this.wait( 1000);
0126                 } catch ( InterruptedException exception) {
0127                    // do nothing.
0128                 } // End try/catch.
0129              } // End synchronized.
0130            } // End if.
0131         } // End while.
```

This fragment contains a **while** loop structure which will iterate until all the images have been loaded, or it is known that there is a problem loading them. The first part of the loop, between lines 0103 and 0108, counts the number of images which have already loaded storing the value in the *numberCounted* local variable. The next part of the loop body, on lines 0110 to 0111, sets the *problemLoading* flag if the *tracker* is able to determine that any of the images could not be obtained.

If no problems have been detected and the number of images that are now known to have completely loaded is greater than the number that were previously known to have loaded and a listener has been registered, on lines 0117 to 0120, an ActionEvent whose actionCommand attribute contains *loading* is dispatched to *itsListener*. Hence it is possible that this method will dispatch as many such ActionEvents as there are images to be loaded. It is the receipt of these events, as will be explained, that allows the opening screen to update the position of its progress bar. The final part of the loop, on lines 0123 to 0130, suspends **this** thread for 100 milliseconds to allow the loading of the images to progress further before the loop iterates again. The final part of the *run()* method is implemented as follows.

```
0133         if ( ! problemLoading) {
0134
0135           cards = new ActiveCard[ pcardFilenames.length];
0136           for (int index =0; index < cards.length; index++) {
0137              cards[ index] = new ActiveCard(
0138                                        thePics[ index],
0139                                        theBack);
0140              cards[ index].setName( pcardFilenames[ index/4]);
0141              cards[ index].addActionListener( itsListener);
0142           } // End for.
0143         } // End if.
0144
0145         if ( itsListener != null) {
0146            itsListener.actionPerformed(
0147                          new ActionEvent( this,
0148                              ActionEvent.ACTION_PERFORMED,
0149                              "loaded"));
0150         } // End if.
0151     } // End run.
0152
0153  } // End PlayingCardLoader.
```

If all the images have loaded without any problems this final stage commences on line 0135 by creating the array of *ActiveCard*s referenced by the *cards* attribute. Each card is then constructed and configured within the loop on lines 0136 to 0141. The setting of each card's name attribute, on line 0140, gives each set of four cards the same name, allowing any pair of the four cards to be matched when the game is played. The final step of the method, on lines 0145 to 0150, is to dispatch an ActionEvent whose actionCommand contains *loaded* to any registered listener. Upon receipt of this event, the *MemoryGame* instance which initiated the loading can request the cards, using the *getCards()* method, and either start the game or inform the user that there was a problem loading the images.

With these two new classes available the revised *MemoryGame init()* method is implemented as follows.

```
0045        public void init() {
0046           this.setResources();
0047
0048           cardLoader = new PlayingCardLoader( this);
0049           try {
0050              cardLoader.addActionListener( this);
0051           } catch ( TooManyListenersException exception) {
0052              // do nothing.
0053           } // end try catch.
0054           cardLoader.start();
0055           gameState = INITIAL;
0056
0057           delay = new Timer( 1000, this);
0058           delay.setRepeats( false);
0059
0060           openingScreen = new MemoryGameOpeningScreen();
0061           this.getContentPane().add( openingScreen);
0062        } // End init.
```

The method begins by constructing and starting a *PlayingCardLoader* instance, the argument indicating that **this** instance being initialized is the applet to use to obtain a toolkit. As soon as it has been constructed its *addActionListener()* method is called to register **this** applet as the object to inform of the loading progress and also the object that the cards are to inform of the user's interaction with them. Before concluding, an instance of *MemoryGameOpeningScreen*, called *openingScreen*, is constructed and added to this applet's contentPane. Hence the applet will appear as shown in Figure 5.13 when it first becomes visible to the user, although the *indicatorBar* will initially show that none of the images have been loaded.

The call of the *cardLoader* start() method, on line 0054, will cause its *run()* method to start execution upon a separate flow of control. As the images are loaded ActionEvents containing *loading* in their *actionCommand* attribute will be dispatched and when the process has finished a single ActionEvent containing *loaded* in its actionCommand attribute will follow. These events will be processed in the revised *actionPerformed*() method which commences as follows.

```
0087       public synchronized void actionPerformed(
0088                                           ActionEvent event) {
0089
0090       String command = event.getActionCommand();
0091
0092          if ( command == null) {
0093             command = new String("time-out");
0094          } // End if.
0095
```

```
0096         if ( command.equals( "loading") ){
0097             openingScreen.updateProgress(
0098                           cardLoader.getProportionLoaded());
0099
0100         } else if ( command.equals( "loaded")) {
0101
0102             if ( cardLoader.areCardsLoaded()) {
0103                this.startGame();
0104             } else {
0105                this.showLoadFailure();
0106             } // End if.
0107
0108         } else if ( command.equals( "clicked")) {
```

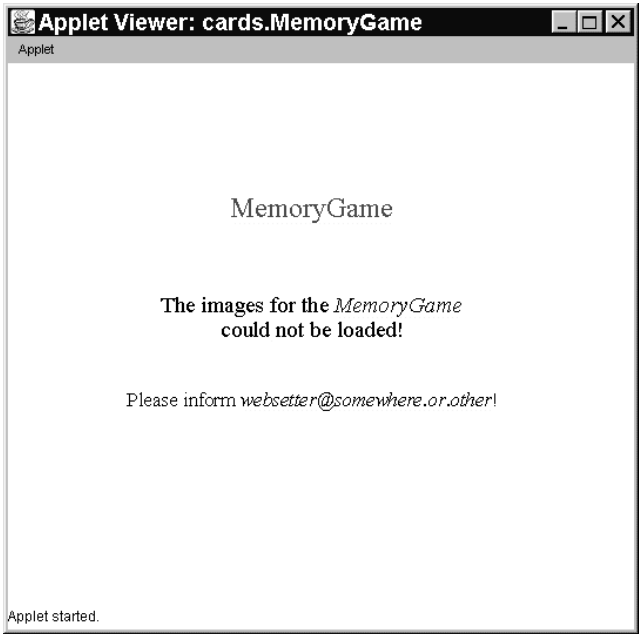
The method now has to be declared with the modifier **synchronized** as it may be called from different flows of control. The receipt of an event whose *actionCommand* attribute is *loading* causes the *openingScreen updateProgress()* method to be called, passing as an argument the value obtained from the *cardLoader getProportionLoaded()* method. The effect is that the progress bar will change its appearance to indicate to the user how much longer they might have to wait.

After the attempt to load the images and construct the cards has completed the *cardLoader* will dispatch a single event whose *actionCommand* is *loaded*. This results, on line 0102, in a call of the *cardLoader areCardsLoaded()* method. If the attempt to prepare the cards was successful the value returned will be **true** and cause the *startGame()* method to be called, on line 0103. Otherwise the cards have not been obtained and the *showLoadFailure()* method is called on line 0105.

The remaining parts of the *actionPerformed()* method are almost unchanged from the original version. If the *showLoadFailure()* method is called the *openingScreen* will still be showing, so it has to be removed and replaced with a screen that informs the user of the failure. It is implemented as follows.

```
0194      private void showLoadFailure() {
0195
0196      JLabel loadFailure = new JLabel( loadFailureMessage);
0197
0198          this.getContentPane().removeAll();
0199          this.getContentPane().add( loadFailure,
0200                                     BorderLayout.CENTER );
0201          this.getContentPane().validate();
0202      } // End showLoadfailure.

0270  private final static String loadFailureMessage =
0272  "<html><CENTER><H1><FONT color=red> Memory Game</FONT></H1>" +
0273  "<P><H2>The images for the <I>MemoryGame</I> <BR>could " +
0274  "not be loaded!</H2><P> <H3>Please inform " +
0275  “<I>websetter@somewhere.or.other</I>!</CENTER></H3>";
```

Line 0198 removes any components that have been added to **this** applet's contentPane and lines 0199 to 200 install a JLabel instance called *loadFailure* into it. The *loadFailure* constructor, on line 0196, specifies *loadFailureMessage* as its argument. This is an HTML fragment, declared at the end of the class, which will be rendered and displayed by the



MemoryGame

The images for the *MemoryGame* could not be loaded!

Please inform *websetter@somewhere.or.other*!

Applet started.

Figure 5.16 *MemoryGame* load failure screen.

JLabel as described in Chapter 3. Having replaced the *openingScreen* with the *loadFailure* screen, the call of validate() on line 0201 will cause it to become visible. The appearance of the *MemoryGame* in this state is shown in Figure 5.16. Once in this state the user can do nothing, but they have at least been informed of this.

The *startGame()* method will be called from *actionPerformed()* if the cards have been successfully constructed. It has the responsibility of placing the interface into the initial *NO_CARD_TURNED* state and is implemented as follows.

```
0065      private void startGame() {
0066
0067         if ( gameArea == null) {
0068            gameArea = new MemoryGameArea(
0069                                 cardLoader.getCards(),
0070                                 cardLoader.getGridDimension());
0071            numberOfPairs =  cardLoader.getCards().length /2;
0072            this.getContentPane().removeAll();
0073            this.getContentPane().add( gameArea);
0074            this.validate();
0075         } // End if.
0076
0077         gameArea.prepareGame();
0078         numberOfPairsTurned = 0;
0079         gameState = NO_CARD_TURNED;
0080      } // End startGame.
```

As will be shown, this method may also be called after a game has finished and so must ensure that all appropriate attributes are set to a suitable state. If it is called for the first

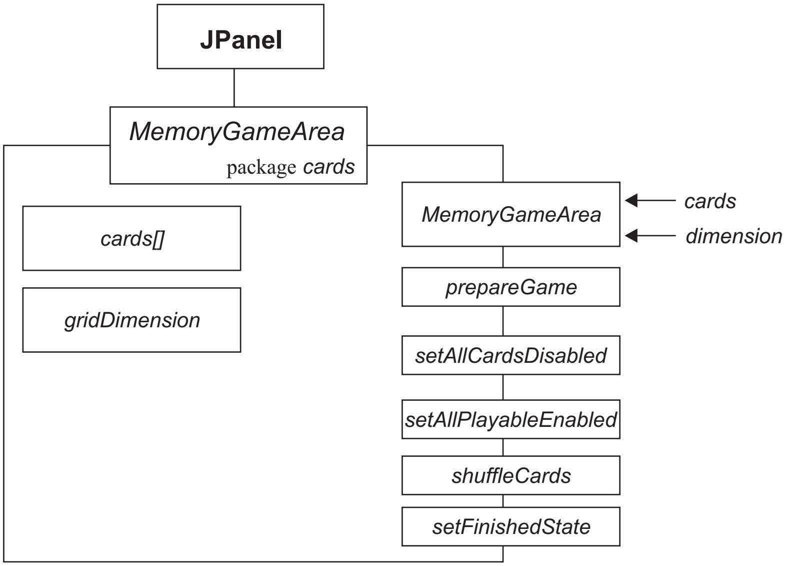


Figure 5.17 *MemoryGameArea* class diagram.

time the *gameArea* will be **null**, so lines 0068 to 0075 prepare a new instance of the *MemoryGameArea*, initialize the *numberOfPairs* attribute and install the *gameArea* into the applet replacing the *openingScreen*. The remaining steps in the method will always be executed and inform the *gameArea* to prepare a new game, reset the *numberOfPairsTurned* attribute and note that the *gameState* is now *NO_CARD_TURNED*.

The class diagram for the *MemoryGameArea* class is given in Figure 5.17. It shows that it encapsulates the *cards* array and the dimension to display them with, both of which are supplied to the constructor. The *prepareGame()* method initializes the interface to start a new game, as will be described. The remaining methods are essentially identical to the comparable methods from the original implementation of the *MemoryGame* and will not be described in this section.

The implementation of this class as far as the end of the constructor is implemented, without comment, as follows.

```
0020  public class MemoryGameArea extends JPanel {
0021
0022  private ActiveCard cards[]       = null;
0023  private Dimension  gridDimension = null;
0024
0025     public MemoryGameArea( ActiveCard  cardsToShow[],
0026                            Dimension  dimensionToUse) {
0027        super();
0028        cards         = cardsToShow;
0029        gridDimension = dimensionToUse;
0030     } // End MemoryGameArea constructor.
```

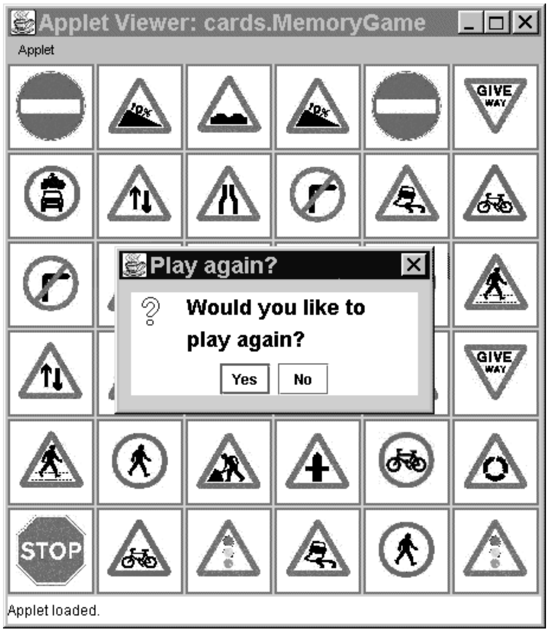
The implementation of the *prepareGame()* method is as follows.

```
0033     public void prepareGame() {
     this.removeAll();
     this.setLayout(
             new GridLayout( gridDimension.width,
                             gridDimension.height));
     this.shuffleCards();
     for ( int index =0; index < cards.length; index++) {
        cards[ index].showBack();
        this.add( cards[ index]);
     } // End for.
     this.setCursor( new Cursor( Cursor.DEFAULT_CURSOR));

     this.validate();
     this.setAllPlayableEnabled();
} // End prepareGame.
```

The method prepares the game area for a new game by installing, or reinstalling, the shuffled cards into itself, making sure that they are showing their backs and are all enabled, and also ensuring that the default cursor is installed.

When this method is called the user will see the backs of all the cards and can commence the game by clicking upon one to turn it over. This will cause an ActionEvent whose actionCommand is *turning* to be dispatched, which will initiate the same sequence of actions and events as described above in the latter part of the *actionPerformed()* method.

The third required change to the *MemoryGame* is to allow the game to be restarted from the finished state. This is accomplished by means of a dialog presented to the user after the game has been completed, as illustrated in Figure 5.18. A JOptionPane modal dialog posted from a revised *MemoryGame setFinishedState()* method is used, implemented as follows.



Applet Viewer: cards.MemoryGame

Play again?

Would you like to play again?

Figure 5.18 *MemoryGame* play again dialog.

```
0170      private void setFinishedState() {
0171
0172      int    replayReply   = 0;
0173      String replayMessage = "Would you like to \n" +
0174                             "play again?";
0175      String replayTitle   = "Play again?";
0176
0177        gameArea.setFinishedState();
0181        replayReply = JOptionPane.showConfirmDialog(
0182                                         this, replayMessage,
0183                                         replayTitle,
0184                                         JOptionPane.YES_NO_OPTION);
0185        if ( replayReply == JOptionPane.YES_OPTION) {
0186           this.startGame();
0187        } else {
0188           gameState = FINISHED;
0189        } // End if.
0190      } // End setFinishedState.
```

The first part of this method calls the *gameArea setFinishedState()* method which will disable all the cards and cause them to show their faces. Following this, lines 0181 to 0184 post a modal *yes/no* confirm dialog which blocks until the user presses one of the buttons. The button pressed determines the value returned from the showConfirmDialog() method which is stored in the local variable *replayReply*. If the user presses the *Yes* button the value returned will be the manifest value YES_OPTION and this results, on lines 0186, in a call of *startGame()* to restart the game. Otherwise the *gameState* attribute is updated to show that the game has well and truly *FINISHED*. The JOptionPane dialogs will be described in detail in the next chapter.

This concludes the initial development of the *MemoryGame* artifact. In the next chapter it will be further developed to allow the user to *drag* the matched pairs off the playing area and *drop* them into one of two storage areas so that the game can be played more naturally and competitively. An application-level menu bar system will also be added to allow the user to terminate the application or to choose between the two sets of cards.

Summary

- ♦ A clip window can be established within a Graphics context to constrain the area of the window effecting by the drawing operations.
- ♦ Specialized components should be designed and implemented without specific regard to the environment they will first be used within.
- ♦ A MediaTracker object can be used to monitor and synchronize the loading of images on a separate thread of control.
- ♦ A state/event table can be very useful in the implementation of complex behavioral logic.
- ♦ When the time an operation takes is indeterminately long the user should be informed of the progress (and given an opportunity to abandon it).
- ♦ Different cursors can be installed to inform the user of the capability or state of the component.

- The JOptionPane class can be used to obtain a number of simple modal dialogs.

Exercises

5.1 Reimplement the *TurnCard* class to support different turn styles, for example from left to right, from the inside out, etc.

5.2 Reimplement the *PlayingCardLoader* class so that two cards can only be matched if they have the same color.

5.3 Locate a suitable set of images and implement a third set of *cards* for the *MemoryGame*.

5.4 Produce an event/state table for the second version of the *MemoryGame* as described in the chapter.

5.5 Using the set of playing cards, amend the *MemoryGame* so that all four matching cards have to be turned over before they are removed.

5.6 Using the set of playing cards, implement a version of the card game 21 (Blackjack or Pontoon). The user will be allowed to turn over up to five cards unless the total score of the cards exceeds 21, in which case the player loses. The artifact should then turn over up to five cards at random and, assuming the artifact's score does not exceed 21, the winner is the player with the lowest score.

6

Main menus and drag and drop

It has become a requirement of almost all artifacts intended for use on desktop windowing systems that they be supplied with an application-level main menu system. Usually these are presented as a row of *menu items* on a *menu bar* at the top of the artifact's window, although some environments such as the Macintosh or OpenStep have a single main menu somewhere on the desktop which changes as the application focus is moved from window to window. A convention has evolved that dictates that the first (leftmost) menu is the *File* menu and its *menu pane* contains the controls to create, open, save and print the current workspace and also to exit from the application. The second menu, which may be omitted, is expected to be the *Edit* menu, containing the undo, redo, cut, copy and paste controls. Other standard menus, for example an *Options* menu, may also be presented and, by convention, the last menu will be the *Help* menu.

These conventions are very valuable to users, as they can recognize the presentation of a standard menu bar and have confidence that they will be able to operate at least some aspects of the artifact by deploying their existing knowledge of main menus. In the latter part of this chapter the JFC facilities and techniques for the construction and implementation of an application-level main menu will be introduced by the provision of a menu system for an upgraded version of the memory game.

Many of the items presented on a menu system cause dialogs to be presented to the user when they are activated. Many of these dialogs, such as *Yes/No* or *OK/Cancel*, are so standard that the dialogs available from the JOptionPane class can be used, for example the *Play Again*? dialog which was supplied as part of the *MemoryGame* artifact at the end of the previous chapter. However, some dialogs are more specialized or complex than those available from the JOptionPane class, and the techniques for the construction and use of specialized dialogs will be introduced at the end of this chapter.

The chapter will commence with a description of the Java facilities which can be used to implement *drag and drop* (*D&D*) capability. This capability has also become almost ubiquitous in modern artifacts, and hence typical users may expect it to be supplied.

6.1 Drag and drop

Drag and drop capability gives the user a very powerful and natural way of interacting with an application. However, the implementation of the capability is somewhat complex. Figure 6.1 provides a simplified, minimal overview of its implementation in Java. A D&D operation is started when a DragGestureListener is informed, via an event,

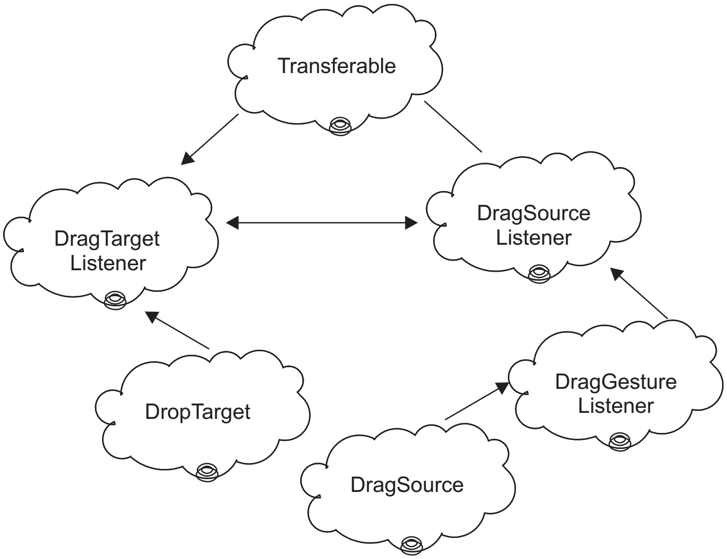


Figure 6.1 Drag and drop overview.

that the user has initiated a drag on the DragSource. It responds by constructing a Transferable instance, containing the object to be transferred, and associating it with a DragSourceListener. As the user drags the mouse pointer into and out of possible drop areas the DragSourceListener and DragTargetListener are informed of the status of the operation. When the user releases the mouse pointer over a possible drop area the DragTargetListener is notified via an event and can obtain the object to be transferred from the Transferable instance to the DropTarget.

In order to illustrate these considerations in detail a demonstration artifact called *DragCardDemo* will be developed. The appearance of this artifact is illustrated in Figure 6.2. On the right-hand side are four possible drag sources, all instances of the *DragCard*

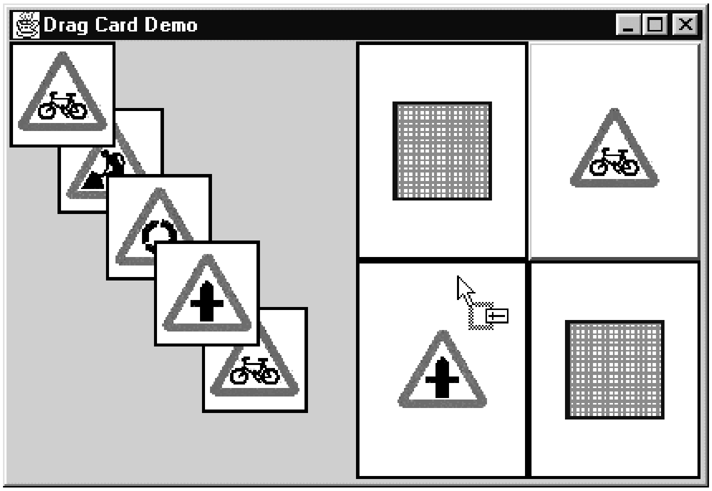


Figure 6.2 *DragCardDemo*: visual appearance.

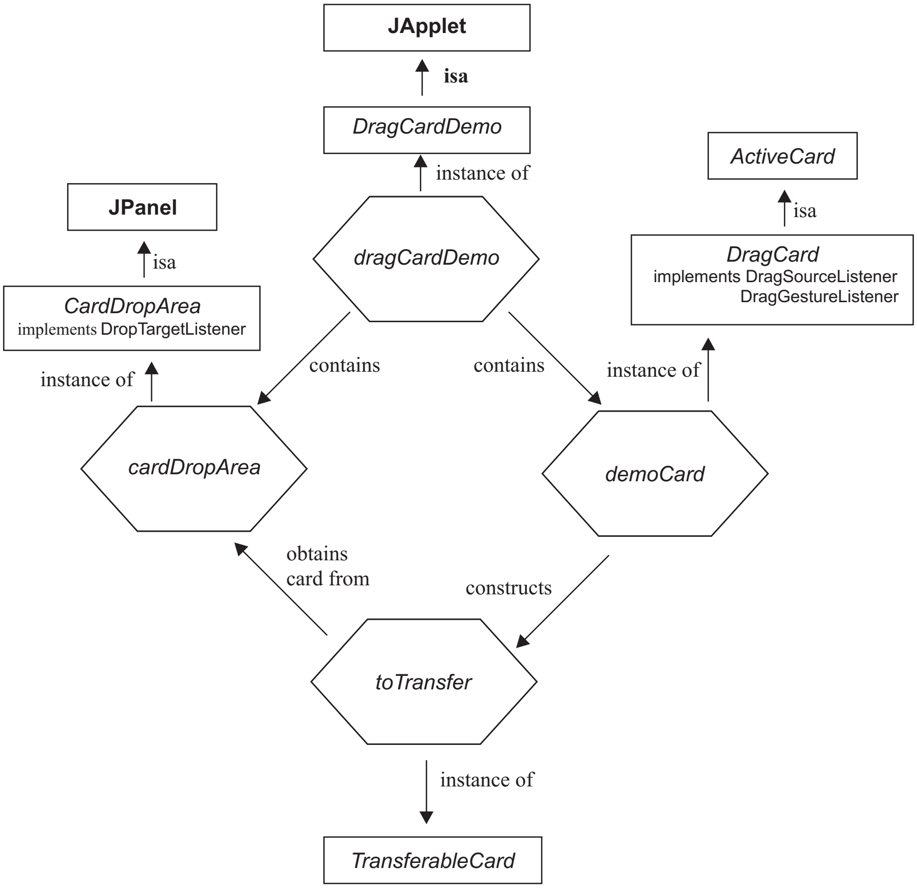


Figure 6.3 *DragCardDemo* context diagram.

class, which is an extension of the *ActiveCard* class. On the left-hand side is the drop target, an instance of the *CardDropArea* class which is an extension of the JPanel class. The image shows that two of the cards on the right have been turned, that five D&D operations have already completed and, from the shape of the cursor, that a sixth D&D operation is under way.

A context diagram for this artifact is given in Figure 6.3. It shows that the applet contains an instance of the *DragCard* class which is extended from the *ActiveCard* class. In terms of the overview shown in Figure 6.1 the *DragCard* class implements the DragSourceListener and DragGestureListener interfaces and also supplies the DragSource object. The applet also contains an instance of the *CardDropArea* class, which implements the DropTargetListener interface. It will obtain its cards, instances of the *DragCard* class, from the *toTransfer* object, an instance of the *TransferableCard* class which is created by the *demoCard* instance each time a D&D operation is initiated.

The class diagram for the *DragCard* class is given in Figure 6.4. The top of the diagram confirms the relationships and interfaces shown on the instance diagram in Figure 6.3. The boolean *draggable* attribute determines whether the card is currently capable of initiating a D&D operation and the *dragSource* attribute will supply the dragging

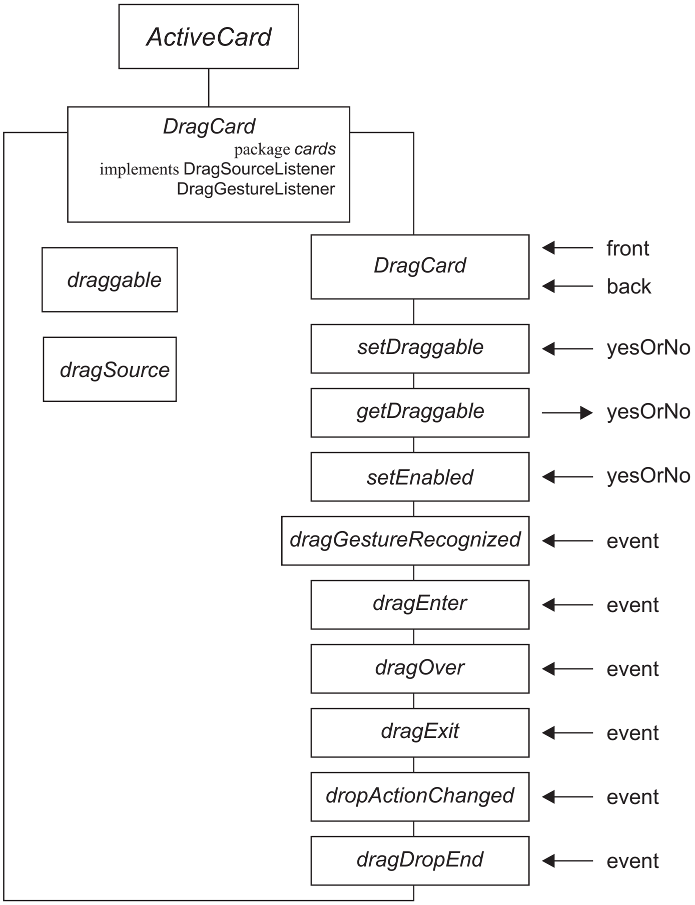


Figure 6.4 *DragCard* class diagram.

capability. The only constructor is comparable with the other *cards* constructors, requiring the *front* and *back* images.

The *setDraggable()* and *getDraggable()* methods are supplied to control the *draggable* attribute and the *setEnabled()* method has to be overridden as a disabled card should not be capable of being dragged from. The *dragGestureRecognized()* method is mandated by the DragGestureListener interface and will be called when the Java environment detects that the user has indicated he or she wants to drag. This is usually effected by the user holding down the mouse button and moving the pointer outside the extent of the card, an operation known as *swiping*. The remaining five methods are mandated by the DragSourceListener interface. The next three methods will be called when the mouse pointer enters a possible drop target, moves over the drop target and leaves it, respectively.

The *dropActionChanged()* method will be called if the user indicates a change of intention. For example, there is a convention that a D&D that originates from a text area will, by default, cause the highlighted text to be moved when it is dropped; however, if the CTRL key is held down the text will be copied. The last method, *dragDropEnd()*, will be called when the drop target accepts or rejects the drop. All of the mandated methods take an instance of the *DragSourceEvent* class as an argument.

The implementation of this class, as far as the end of its constructor, is as follows.

```
0010  package cards;
0011
0012  import javax.swing.*;
0013  import java.awt.*;
0014  import java.awt.event.*;
0015  import java.awt.dnd.*;
0016  import java.awt.datatransfer.*;
0017
0018
0019  public class DragCard extends ActiveCard
0020                       implements DragSourceListener,
0021                                  DragGestureListener {
0022
0023  private DragSource dragSource = null;
0024  private boolean    draggable  = true;
0025
0026     public DragCard( Image front,
0027                      Image back) {
0028        super( front, back);
0029        dragSource = DragSource.getDefaultDragSource();
0030        dragSource.createDefaultDragGestureRecognizer(
0031                     this, DnDConstants.ACTION_COPY, this);
0032     } // End DragCard constructor.
```

The *dragSource* attribute is declared on line 0023 as an instance of the DragSource class and the **boolean** *draggable* attribute is initialized to indicate that dragging is allowed. The constructor commences by indirecting to its super, *ActiveCard*, constructor passing on its arguments.

Object → java.awt.dnd.DragSource

```
public DragSource()
```

Constructs a DragSource instance; the default drag source is usually sufficient.

```
public static DragSource getDefaultDragSource()
```

Obtains the identity of the platform DragSource object.

```
public DragGestureRecognizer createDefaultDragGestureRecognizer(
                                                Component recognizer,
                                                int       dndActions,
                                       DragGestureListener listener)
```

Creates a DragGestureRecognizer for the *recognizer* component, allowing the *actions* specified, and the *listener* to be notified when the D&D operation is initiated. Actions, defined in DnDConstants, include ACTION_COPY, ACTION_MOVE, ACTION_COPY_OR_MOVE and ACTION_LINK.

```
public void startDrag( DragGestureEvent trigger,
                       Cursor           dragCursor,
                       Transferable     transferable,
                     DragSourceListener listener)
```

Initiate a drag operation with the object to be transferred in *transferable*, informing *listener* of the progress of the operation, using the cursor specified. The trigger can usually be obtained from the DragRecognizer.

On line 0029, the *dragSource* attribute is initialized to identify the system's DefaultDragSource. Once it is obtained an anonymous DefaultDragGestureRegognizer is created and associated with it. The arguments to the creation method are the Component, **this**, to associate with the recognizer; the allowed operations, copy only in this example; and the identity, **this**, of the DragGestureListener to inform of the progress of the operation. These two default objects are adequate for simple D&D operations; more complex operations, which may require a greater range of gestures or improved feedback, would require more specialized instances to be supplied. The *draggable* and enabled methods are implemented as follows.

EventListener → java.awt.dnd.DragGestureListener, interface

```
public void dragGestureRecognized( DragGestureEvent event)
```

Method to be called when the drag gesture is recognized.

EventListener → java.awt.dnd.DragSourceListener, interface

```
public void dragDropEnd( DragSourceDropEvent event)
public void dragEnter( DragSourceDropEvent event)
public void dragExit( DragSourceDropEvent event)
public void dragOver( DragSourceDropEvent event)
public void dragActionChanged( DragSourceDropEvent event)
```

Methods called to inform on the progress of the D&D operation.

```
0034     public void setDraggable( boolean yesOrNo) {
0035        draggable = yesOrNo;
0036     } // End setDraggable.
0037
0038     public boolean isDraggable() {
0039        return draggable;
0040     } // End isDraggable
0041
0042     public void setEnabled( boolean yesOrNo) {
0043        super.setEnabled(  yesOrNo);
0044        this.setDraggable( yesOrNo);
0045     } // End setEnabled.
```

The *setEnabled()* method also ensures that the *draggable* attribute is set by calling the *setDraggable()* method after calling the inherited, **super**, *setEnabled()* method. The five methods mandated by the DragSourceListener interface are implemented as follows.

```
   public void dragEnter( DragSourceDragEvent event) {
      event.getDragSourceContext().setCursor(
                         DragSource.DefaultCopyDrop);
   } // End dragEnter.

   public void dragOver( DragSourceDragEvent event) {
      // Do nothing.
   } // End dragOver.

   public void dragExit( DragSourceEvent event) {
      event.getDragSourceContext().setCursor( null);
   } // End dragExit.

   public void dropActionChanged(
                            DragSourceDragEvent event) {
      // Do nothing.
   } // End dropActionChanged.

   public void dragDropEnd( DragSourceDropEvent event) {
      // Do nothing.
   } // End dragDropEnd.

} // End class DragCard.
```

These methods will be called at the appropriate points during a D&D operation. For a simple example such as this, only the *dragEnter()* and *dragExit()* methods need to be supplied with bodies. The *dragEnter()* method ensures that the DefaultCopyDrop cursor, supplied by the DragSource class, will be shown while the mouse pointer is within the extent of a possible drop target. The *dragExit()* method restores the existing cursor as the pointer leaves the drop target. Both of these methods make use of the setCursor() method of the DragSourceContex object which is a helper instance automatically associated with the *dragSource* instance. The implementation of the *dragEnter()* method is a little naïve as it will always indicate that a copy operation is possible on any drop target, even when it subsequently turns out not to be possible. The upgrading of this method to indicate if the copy is, or is not possible, will be left as an end of chapter exercise after the appropriate inquiry methods have been introduced. The only other *DropCard* method is *dragGestureRecognized()* which is mandated by the DragGestureListener interface and is implemented as follows.

```
   public void dragGestureRecognized(
                                  DragGestureEvent event) {

   TransferableCard  toTransfer = null;

      if ( this.isDraggable() ) {
         toTransfer = new TransferableCard( this);
         dragSource.startDrag( event,
0056                                      DragSource.DefaultCopyDrop,
0057                                      toTransfer, this);
0058            } // End if.
0059         } // End dragGestureRecognized.
```

This method will be called when the Java environment recognizes that the user has indicated the initiation of a D&D operation by swiping the mouse pointer across the *DragCard* instance. It is protected by a guard, on line 0053, which will only initiate the operation if the *draggable* attribute is set. The *TransferableCard* class, which will be described shortly, contains the object to be dragged and is constructed on line 0054. The argument to the constructor, **this**, indicates that the card which is being interacted with is the object to be dragged.

Object → EventObject → java.awt.dnd.DragSourceEvent

```
public DragSourceEvent( DragSourceContext context)
```

A parent class for the DragSourceDragEvent and DragSourceDropEvent classes containing only the *context* that controls the operation.

```
public DragSourceContext getDragSourceContext()
```

Inquiry method on the single attribute.

Object → EventObject → DragSourceEvent → java.awt.dnd.DragSourceDragEvent

```
public DragSourceDragEvent( DragSourceContext context,
                            int               actions,
                            boolean           success)
```

DragSourceDragEvents are generally not constructed. The *context* controls the overall operation, the *actions* indicates the drop operation performed and *success* indicates the outcome of the operation.

```
public int     getDropAction()
public boolean getGestureModifiers()
```

Inquiry methods.

Object → EventObject → DragSourceEvent → java.awt.dnd.DragSourceDropEvent

```
public void startDrag( DragGestureEvent trigger,
                       Cursor           dragCursor,
                       Transferable     transferable,
                       DragSourceListener listener)
```

Initiate a drag operation with the object to be transferred in *transferable*, informing *listener* of the progress of the operation, using the cursor specified. The *trigger* can usually be obtained from the DragRecognizer.

Lines 0055 to 0057 initiate the drag by calling the *dragSource* startDrag() method, the arguments indicating the *event* that was passed as an argument to the *dragGestureRecognized()* method, that a DefaultCopyDrop operation is to be initiated, the object *toTransfer* and the DragSourceListener, **this**, which should be informed of the

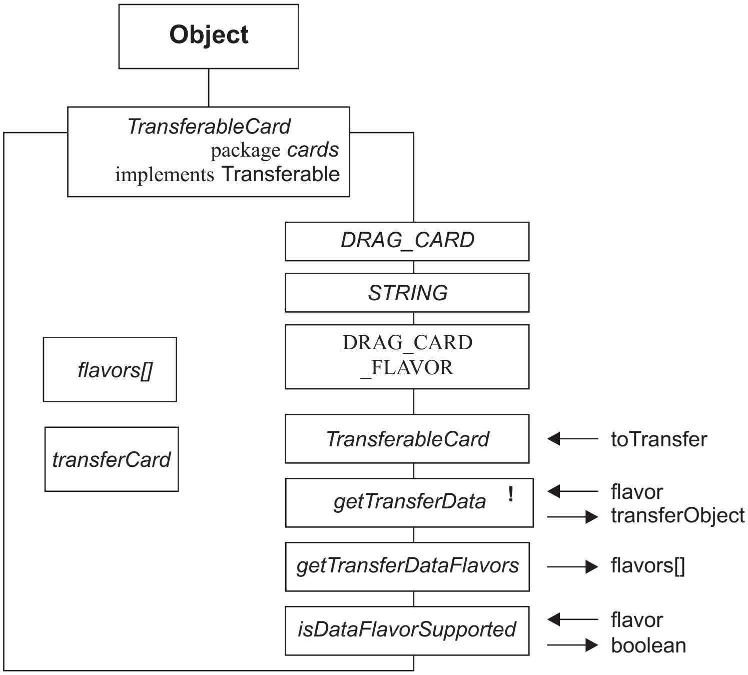


Figure 6.5 *TransferableCard* class diagram.

progress of the operation. When this method is called, as shown in Figure 6.2, it installs a D&D cursor and causes Java to monitor the D&D process constructing and dispatching *DragGestureEvents* as appropriate. Referring back to Figure 6.1, a *Transferable* instance containing the object to be dragged is required. In this example it is an instance of the *TransferableCard* class, whose class diagram is given in Figure 6.5.

The class diagram indicates that the *TransferableCard* class extends the Object class and implements the *Transferable* interface. Drag and drop protocols require that the object being transferred should be available in a number of different formats known as *flavors*. A *TransferableCard* instance is capable of supplying the card either as a *STRING* or as a *DRAG_CARD*, and these two flavors are defined by the two public manifest values shown on the diagram. The *DRAG_CARD_FLAVOR* is defined as a class-wide public resource and is a component part of the private class-wide *flavors[]* array. These two flavors are supported by the *getTransferFlavors()* and *isDataFlavorSupported()* methods, both of which are mandated by the Transferable interface as instance methods.

java.awt.datatransfer.Transferable, interface

```
public DataFlavor[] getTransferDataFlavors()
public boolean      isDataFlavorSupported( DataFlavor flavor)
```

Inquiry methods to determine the transfer flavors supported.

```
public Object getTransferData( DataFlavor flavor)
                         throws UnsupportedFlavorException, IOException
```

Method to obtain the Object to be transferred.

The implementation of the *TransferableCard* class commences as follows.

```
0010  package cards;
0011
0012  import java.awt.*;
0013  import java.awt.dnd.*;
0014  import java.awt.datatransfer.*;
0015  import java.io.*;
0016
0017  public class TransferableCard extends Object
0018                             implements Transferable {
0019
0020  public final static int DRAG_CARD  = 0;
0021  public final static int STRING     = 1;
0022
0023  private final static DataFlavor DRAG_CARD_FLAVOR =
0024                   new DataFlavor( Object.class, "DragCard");
0025
0026  private final static DataFlavor flavors[] = {
0027                                DRAG_CARD_FLAVOR,
0028                                DataFlavor.stringFlavor };
0029
0030  private DragCard transferCard = null;
```

Lines 0020 and 0021 declare the manifest names of the flavors which are supported and lines 0023 and 0024 the *DRAG_CARD_FLAVOR* itself. This is an instance of the DataFlavor class, which defines the mechanism and human-presentable name of the flavor. In this example the *DragCard* is being transferred as a Java Object, as opposed to, say, a stream, as indicated in the first argument to the constructor. The second argument may be used by a more complex artifact to supply a "*Paste As...*" menu, allowing the user to select the transfer format, for example HTML or formatted or unformatted text.

Lines 0026 to 0028 then declare and initialize the *flavors[]* array containing the two supported flavors as indicated by the manifests. The String data flavor is a standard flavor whose DataFlavor object is supplied by the DataFlavor class. The final declaration is the *DragCard* instance which will be transferred and which is initialized by the constructor as follows.

```
0032    public TransferableCard( DragCard toTransfer) {
0033       super();
0034       transferCard = new DragCard(
0035                                toTransfer.getFrontImage(),
0036                                toTransfer.getBackImage());
0037       transferCard.setName( toTransfer.getName());
0038    } // End TransferableCard constructor.
```

The constructor takes as an argument a *DragCard* and constructs a copy of it in the *transferCard* attribute.

The *transferCard* contains an identical copy of the *DragCard* supplied as an argument. The three defining features of a card are the front and back images, as supplied to the *DragCard* constructor on lines 0035 and 0036, and the card's name, as specified on line 0037. The effect of the construction is that the *TransferableCard* instance will contain a copy of the card which is being transferred in its *transferCard* attribute.

The *getTransferDataFlavors()* and *isDataFlavorSupported()* inquiry methods are both mandated by the Transferable interface and are implemented, without comment, as follows.

```
0057    public DataFlavor[] getTransferDataFlavors() {
0058       return flavors;
0059    } // End getTransferDataFlavors.
0060
0061    public boolean isDataFlavorSupported(
0062                                 DataFlavor flavorToCheck) {
0063
0064    boolean flavorFound   = false;
0065    boolean listExhausted = false;
0066    int     index         = 0;
0067
0068      while ( ! listExhausted &&
0069              ! flavorFound   ){
0070        if ( flavorToCheck.equals( flavors[ index])) {
0071           flavorFound = true;
0072        } else {
0073           index++;
0074           if ( index == flavors.length) {
0075              listExhausted = true;
0076           } // End if.
0077        } // End if.
0078      } // End while.
0079      return flavorFound;
0080    } // End isDataFlavorSupported.
0081
0082  } // End TransferableCard.
```

The only remaining method, *getTransferData()*, is mandated by the Transferable interface. It is responsible for supplying the object to be transferred in the *flavor* indicated by the argument, or throwing a UnsupportedFlavorException if the argument is not one of those in the *flavors[]* array.

```
0041    public Object getTransferData( DataFlavor flavorRequested)
0042                throws UnsupportedFlavorException, IOException {
0043
0044    Object transferObject = null;
0045
0046       if ( flavorRequested.equals( flavors[ DRAG_CARD])) {
0047          transferObject = transferCard;
0048       } else if( flavorRequested.equals( flavors[ STRING])) {
0049          transferObject = transferCard.getName();
0050       } else {
0051          throw new UnsupportedFlavorException( flavorRequested);
0052       } // End if.
0053       return transferObject;
0054    } // End getTransferData.
```

The substantive part of the method, on lines 0046 to 0052, is a selection structure which initializes the local *transferObject* variable to the identity of the *transferCard* or its name,

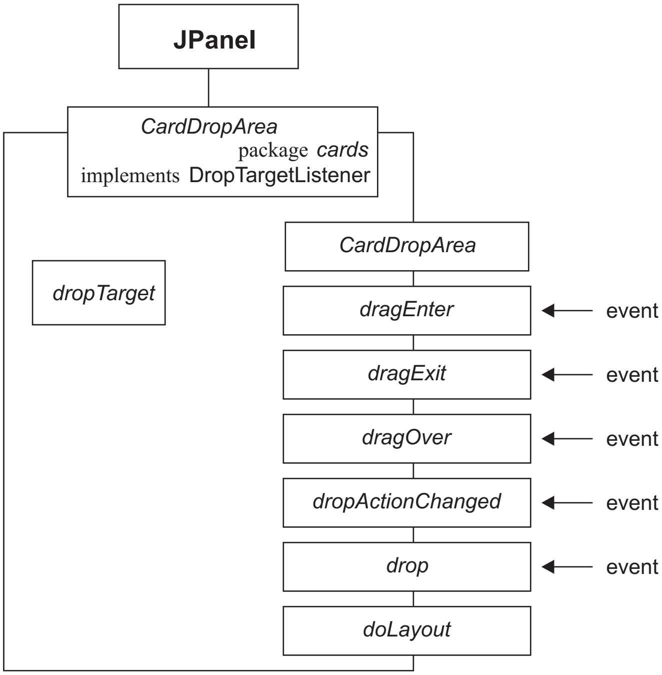


Figure 6.6 *CardDropArea* class diagram.

depending upon the *flavorRequested*. If the *flavorRequested* is not one of the two supported flavors then an UnsupportedFlavorException is thrown, on line 0051; otherwise the *transferObject* is returned on line 0053. The method must indicate that it might throw an IOException, as the run-time implementation of the transfer mechanism makes use of streams and an exception may be propagated from it. The effect of calling this method, if an exception is not thrown, is to make the data to be dragged available in the requested flavor.

With the drag source and transferable classes constructed, only the drop target class, *CardDropArea*, remains before the mechanism can be demonstrated, its class diagram is given in Figure 6.6. It indicates that the *CardDropArea* class extends the JPanel class, as it has to contain and lay out the cards dropped onto it, and also that it implements the *DropTargetListener* interface in order for it to be informed of the progress of the D&D operation. Its only attribute is a *DropTarget* which actually manages the dropping of the card.

java.awt.dnd.DropTragetListener, interface

```
public void dragEnter( DropTargetDragEvent event)
public void dragOver( DropTargetDragEvent event)
public void dropActionChanged( DropTargetDragEvent event)
public void dragExit( DropTargetDragEvent event)
public void drop( DropTargetDragEvent event)
```

Methods that are called to inform the listener of the progress of the D&D operation.

The first four methods *dragEnter()*, *dragExit()*, *dragOver()* and *dropActionChanged()* are all mandated by the *DropTargetListener* interface, all take an *event* argument of the

DropTargetDragEvent class and all are comparable to the methods of the same name mandated by the *DragSourceListener* interface, informing the drop target of the progress of the D&D operation. The fifth method, *drop()*, is also mandated by the *DropTargetListener* interface and has a *DropTargetDragEvent* argument. It will be called when the user releases an object onto the drop target. The only remaining method is *doLayout()*, which is required as the *CardDropArea* takes the responsibility for laying out the cards which have been dropped upon it. The implementation of this class, as far as the end of its constructor, is as follows.

```
0010  package cards;
0011
0012  import javax.swing.*;
0013  import java.awt.*;
0014  import java.io.*;
0015  import java.awt.dnd.*;
0016  import java.awt.datatransfer.*;
0017
0018
0019  public class CardDropArea extends JPanel
0020                         implements DropTargetListener {
0021
0022  private DropTarget dropTarget = null;
0023
0024     public CardDropArea() {
0025        super();
0026        this.setLayout( null);
0027        this.setMinimumSize( new Dimension( 100, 200));
0028        dropTarget = new DropTarget( this, this);
0029     } // End CardDropArea constructor.
```

The declaration of the class, on lines 0019 to 0020, is as expected from the class diagram. The constructor commences, on line 0025, with a call of its **super**, JPanel, constructor and then removes the default layout manager before establishing an arbitrary minimumSize. The final step, on line 0028, is to construct the DropTarget instance, the arguments to its constructor indicating the component that the objects are to be dropped onto and the DropTargetListener to be notified of the progress of the operation. In this example these are both **this** *CardDropArea* being constructed. The implementation of the first four methods mandated by the DropTargetListener interface is as follows.

```
0031     public void dragEnter( DropTargetDragEvent event) {
0032        event.acceptDrag( DnDConstants.ACTION_COPY);
0033     } // End dragEnter.
0034
0035     public void dragExit( DropTargetDragEvent event) {
0036        // Do Nothing.
0037     } // End dragExit.
0038
0039     public void dragOver( DropTargetDragEvent event) {
0040        // Do Nothing.
0041     } // End dragOver.
0042
0043     public void dropActionChanged(
0044                                 DropTargetDragEvent event) {
0045        // Do Nothing.
0046     } // End dropActionChanged.
```

These methods will be called at the appropriate points in the D&D operation. The only one that needs a body in a simple example such as this is the *dragEnter()* method, which is called when a dragging mouse pointer enters the drop target. In this example it is indicating that it will always accept the drop, even if it subsequently turns out that the object being transferred is not available in an acceptable flavor. The upgrading of this method will be left as an end of chapter exercise. The other three methods need not have a body supplied. The *drop()* method will be called when the object is dropped and is implemented as follows.

```
0049     public void drop( DropTargetDropEvent event) {
0050
0051     DragCard     received  = null;
0052     Transferable transfer  = null;
0053
0054        try {
0055           transfer = event.getTransferable();
0056           if ( transfer.isDataFlavorSupported(
0057                         TransferableCard.DRAG_CARD_FLAVOR)) {
0058
0059              event.acceptDrop(
0060                             DnDConstants.ACTION_COPY_OR_MOVE);
0061              received = (DragCard) transfer.getTransferData(
0062                         TransferableCard.DRAG_CARD_FLAVOR);
0063              received.showFront();
0064              received.setDraggable( false);
0065              this.add( received);
0066              this.validate();
0067           } else {
0068              event.rejectDrop();
0069           } // End if.
0070        } catch ( IOException exception) {
0071           event.rejectDrop();
0072        } catch ( UnsupportedFlavorException exception) {
0073           event.rejectDrop();
0074        } // End try/catch
0075     } // End drop.
```

The method commences by retrieving the Transferable instance from the DropTargetDropEvent passed as an argument. If the drag originated from a *DragCard* this will be an instance of the *TransferableCard* class. Once the *transfer* instance has been obtained, its *isDataFlavorSupported()* method is called to establish whether it can provide the object being dropped in the *TransferableCard.DRAG_CARD_FLAVOR*. Should this not be the case the **else** clause on lines 0067 and 0068 will reject the drop.

Object → EventObject → java.awt.dnd.DropTargetEvent

```
public DropTargetEvent( DragTargetContext context)
```

A parent class for the DropTargetDragEvent and DropTargetDropEvent classes containing only the *context* that controls the operation.

```
public DragTargetContext getDragTargetContext()
```

Inquiry method on the single attribute.

Object → EventObject → DropTargetEvent → java.awt.dnd.DropTargetDragEvent

```
public DropTargetDragEvent( DropTargetContext context,
                            Point             location,
                            int               dropAction
                            int               sourceActions)
```

The *context* is as for the DropTargetEvent, *location* is the position of the mouse pointer, *dropAction* is the user selected operation and *sourceActions* those supported by the source.

```
public Point getLocation()
public int   getDropAction()
public int   getSourceActions()
```

Inquiry methods on the constructor attributes.

```
public DataFlavor[] getCurrentDataFlavors()
public boolean      isDataFlavorSupported( DataFlavor flavor)
```

Convenience methods to determine the available flavors of the source.

```
public void acceptDrag( int dragOperation)
public void rejectDrag()
```

Methods to accept or reject the operation.

Object → EventObject → DropTargetEvent → java.awt.dnd.DropTargetDropEvent

```
public DropTargetDropEvent( DropTargetContext context,
                            Point             location,
                            int               dropAction
                            int               sourceActions)
```

The *context* is as for the DropTargetEvent, *location* is the position of the mouse pointer, *dropAction* is the user selected operation and *sourceActions* those supported by the source.

```
public Point getLocation()
public int   getDropAction()
public int   getSourceActions()
```

Inquiry methods on the constructor attributes.

```
public DataFlavor[] getCurrentDataFlavors()
public boolean      isDataFlavorSupported( DataFlavor flavor)
```

Convenience methods to determine the available flavors of the source.

```
public void         acceptDrop( int dropAction)
public Transferable getTransferable()
public void         dropComplete( boolean success)
public void         rejectDrop()
```

Methods to accept or reject the operation, together with one to obtain the Transferable object and to indicate success or otherwise.

If the object can be supplied in the required flavor, lines 0061 and 0062 obtain the Object from the transfer instance, casting it to its known class, *DragCard*, and referencing it by the local variable *received*. It is this method which might throw an IOException or an UnsupportedFlavorException, which are both handled by rejecting the drop, on lines 0070 to 0073. If the card is successfully *received*, lines 0063 and 0064 ensure that its face is showing and set its *draggable* attribute to prevent it being dragged out of the drop target. The card is then added to **this** *CardDropArea*, which is immediately validated to cause layout negotiations and a repaint to take place.

The remaining method, *doLayout()*, will be called during layout negotiations and must inform every card contained in the panel of its location and size, in order that they will appear in an appropriate place when they are subsequently repainted. The implementation of *doLayout()*, presented as follows without comment, positions the cards in a diagonal from top left to bottom right as shown in Figure 6.2.

```
0078     public void doLayout() {
0079
0080     Dimension containerSize = this.getSize();
0081     Dimension cardSize      = null;
0082     Component theCards[]    = this.getComponents();
0083
0084     int stepX      = 0;
0085     int stepY      = 0;
0086     int thisCardsX = 0;
0087     int thisCardsY = 0;
0088
0089        if ( theCards.length > 0){
0090           cardSize = theCards[ 0].getMinimumSize();
0091           stepX = (containerSize.width - cardSize.width)   /
0092                                                   theCards.length;
0093           stepY = (containerSize.height - cardSize.height) /
0094                                                   theCards.length;
0095
0096           for( int index =0;
0097                index < theCards.length;
0098                index++) {
0099              theCards[ index].setLocation( thisCardsX,
0100                                            thisCardsY);
0101              theCards[ index].setSize( cardSize);
0102              thisCardsX += stepX;
0103              thisCardsY += stepY;
0104           } // End for.
0105           this.repaint();
0106        } // End if.
0107     } // End doLayout.
0108
```

```
} // End class CardDropArea.
```

The implementation of the *DragCardDemo* artifact, which will not be presented in detail, constructs four instances of the *DragCard* class, attaches them to an ActionListener which turns them over whenever they are clicked, and installs an instance of the *CardDropArea* class alongside, into which they can be dropped. The user can turn any of the cards by clicking upon them or can initiate a drag by swiping across them. Once a drag has been initiated the cursor will change to show this to the user, as illustrated in Figure 6.2, and they can be dropped into the drop target area. As each card is dropped all cards will be re-laid out in an appropriate pattern.

Although the cards in the drop area are not connected to an ActionListener which would turn them they are still sensitive to mouse entry and mouse exit, and the repaint which is occasioned by these events will ensure that they rise to the top of the stack order as the mouse pointer pans across them. Having developed and demonstrated the D&D capability the *MemoryGame* can be re-engineered to make use of the capability.

6.2 The *MemoryGame* with D&D capability added

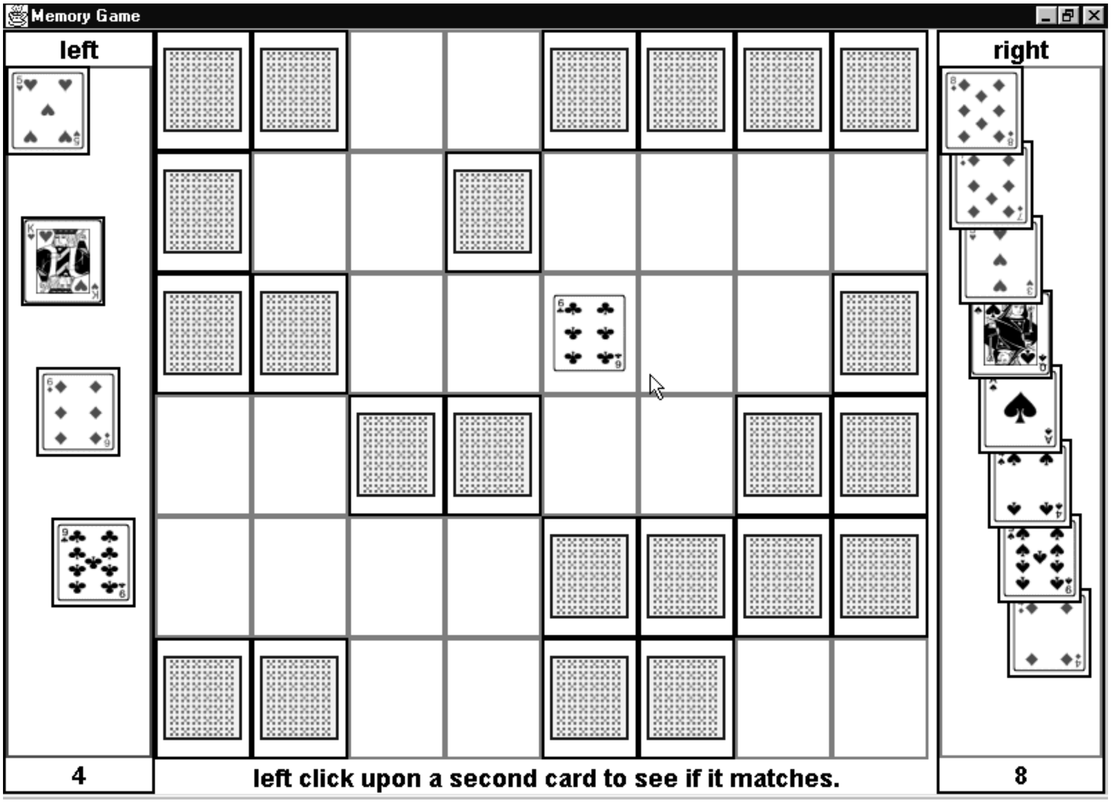
The appearance of the revised version of the *MemoryGame* artifact is shown in Figure 6.7. The most apparent difference is the inclusion of two panels to the left and right of the main area which contain a name label at the top, a drop target area in the middle and a score label at the bottom. The only other difference is the inclusion of a message area below the main playing area that suggests to one or other of the players what they should do next.

The illustration shows that at this stage in the game the *left* player has obtained 4 pairs and the *right* player 8. Currently left is playing, has turned over the six of clubs and is being asked to click upon a second card. If the card turned is also a six the player will be requested to drag one of the turned cards from the playing area into the player's drop area. As the left player is currently playing, only the left-hand drop area will accept a card dropped onto it, preventing the player from accidentally dropping the card onto the right-hand drop area. When the card has been dropped the two turned cards will be removed from the playing area and the right player will be invited to turn over his or her first card. If the second card turned over by left is not a six then both turned cards will flip onto their backs and the right player will be invited to turn over a card.

This description of the way in which the game is played may vary from other understandings; for example, if *left* were to turn over a matched pair, control of the game might be retained by that player. This consideration again emphasizes the importance of fully specifying the intended behavior of an artifact and not relying upon idiosyncratic understandings. The revised STD for this version of the game is given in Figure 6.8.

The STD differs by introducing an additional state called *drag off*, which is reached when a pair of cards in the *two cards turned* state match, and which can only be exited from by the dropping of a card onto the active drop target. If this is not the penultimate pair the players are switched and the game returns to the *no card turned* state. If it was the penultimate pair then the remaining pair of cards is credited to the current player and the game enters the terminal state. The transition from the *two card turned* to the *no card turned* state, which is followed if the cards do not match, is augmented by the action of switching the players.

In order to implement this version of the game the *MemoryGameArea* has been expanded from the previous version by the inclusion of a JLabel *message* at the bottom



left click upon a second card to see if it matches.

Figure 6.7 *MemoryGame* with D&D added.

and the provision of three methods called *setNoCardTurnedState()*, *setOneCardTurnedState()* and *setDragOffState()*, each of which takes a *playerName* String argument and prepends it to a standard phrase which is installed as the JLabel's text resource, causing it to be shown. The details of these changes will not be presented.

The *CardDropArea* will also require some minor improvements over the version presented in the previous section to make it suitable for this requirement. One change is to include a **boolean** attribute called *droppable*, accompanied by methods to set and obtain its value. The two methods mandated by the DropTargetListener interface which were provided with bodies, *dragEnter()* and *drop()* are supplied with guards which cause them to do nothing if the *droppable* attribute is not set. Hence a *CardDropArea* instance which has a **false** *droppable* attribute will not allow a card to be dropped upon it, nor will it indicate that it is capable of doing so by installing a cursor when a dragging mouse pointer enters it.

The only other change in the *CardDropArea* class is to provide a ChangeListener attribute which is initialized during construction from an argument which must be passed to the constructor. When the state of the instance is changed, by the dropping of a card upon it, it will dispatch a *ChangeEvent* to inform its listener of this. With these changes in place the *MemoryGameCardDropArea* class, which is used for the left- and right-hand parts of the game, can be designed and implemented. Its class diagram is given in Figure 6.9.

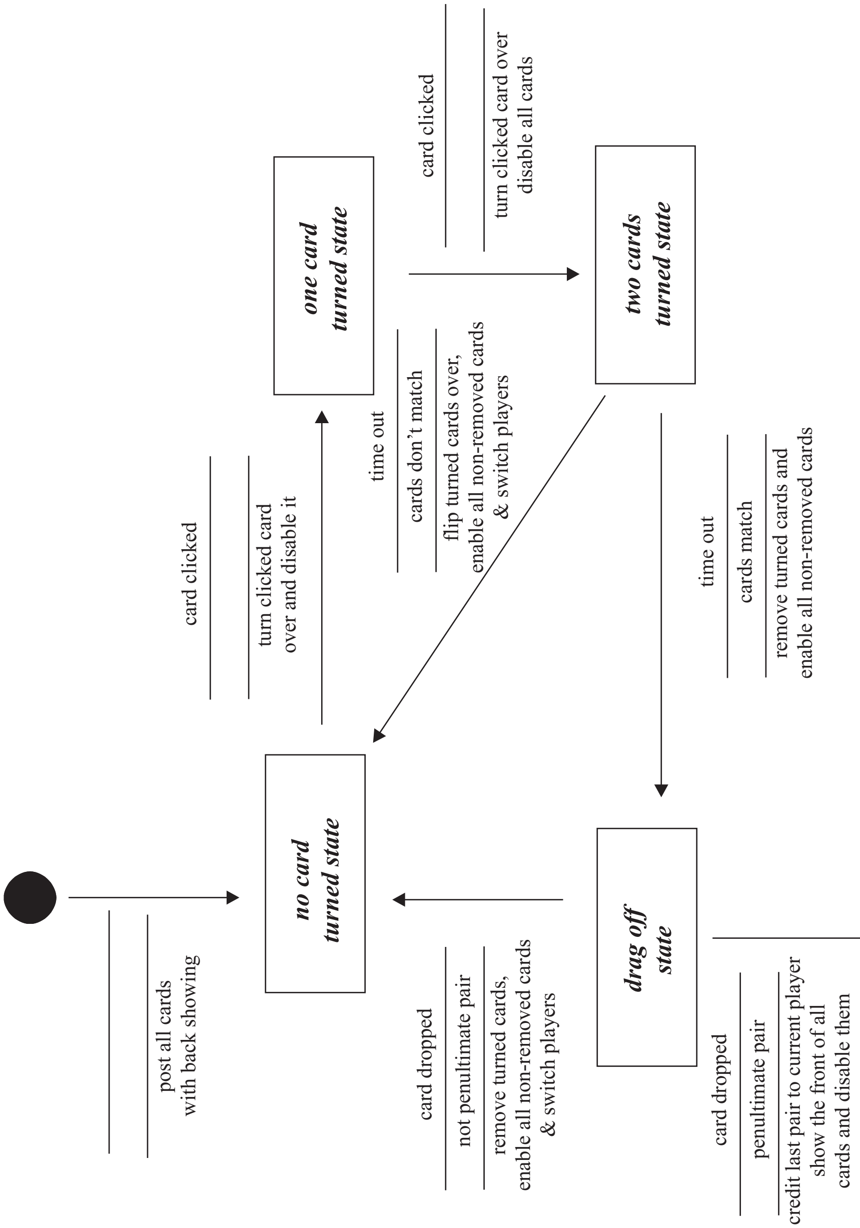


Figure 6.8 *MemoryGame*: revised STD.

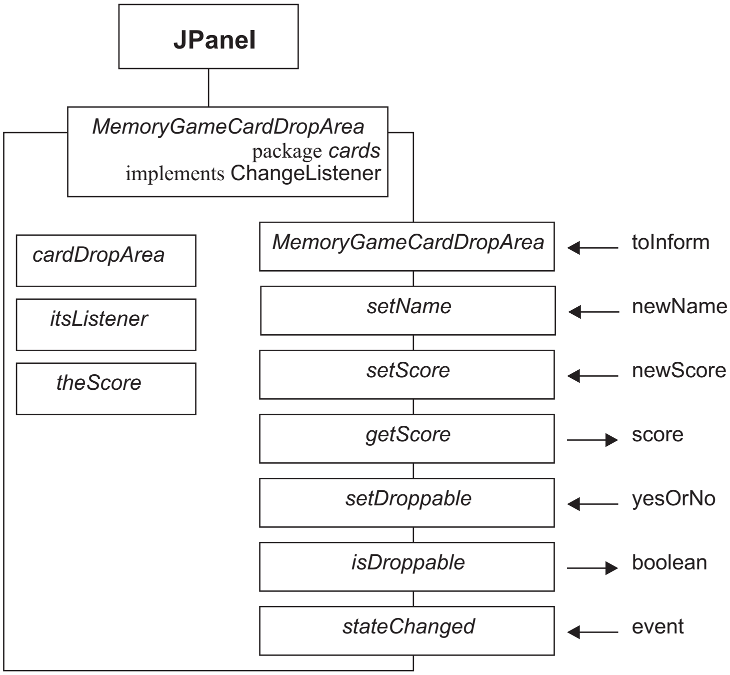


Figure 6.9 *MemoryGameCardDropArea* class diagram.

The diagram shows that the *MemoryGameCardDropArea* class is a member of the *cards* package and that it implements the ChangeListener interface in order that it can be registered as the destination of ChangeEvents dispatched from the encapsulated *CardDropArea* instance. Its constructor requires the identity of an ActionListener *toInform* when a card is dropped upon it. The name attribute is not shown on the class diagram and is a write-only attribute having a setting but not a getting method. The *theScore* attribute records the number of cards dropped upon it and has a setter and a getter. Although there are *setDroppable()* and *isDroppable()* methods there is no *droppable* attribute, as these will be implemented as wrapper methods on the *cardDropArea* attribute. The only other method is *stateChanged()*, which takes a ChangeEvent as an argument and which is mandated by the ChangeListener interface. The implementation of this class, as far as the start of its constructor, is as follows.

```
package cards;

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

public class MemoryGameCardDropArea extends JPanel
                                 implements ChangeListener{

private CardDropArea   cardDropArea = null;
```

```
0022  private ActionListener itsListener  = null;
0023  private JLabel         nameLabel    = null;
0024  private JLabel         scoreLabel   = null;
0025  private int            theScore     = 0;
0026
0027  private Border droppableBorder =
0028                              new LineBorder( Color.red, 2);
0029  private Border nonDroppableBorder =
0030                             new LineBorder( Color.gray, 2);
```

The class is declared, as expected from the class diagram, on lines 0018 and 0019. Lines 0021 to 0025 declare the three attributes shown on the class diagram as well as the two JLabel instances needed at the top and bottom of the component. The declarations are completed by two LineBorder instances that will be installed into the *cardDropArea* instance to indicate to the user whether it will or will not accept drops. The constructor is implemented as follows.

```
0032     protected MemoryGameCardDropArea(
0033                                   ActionListener toInform) {
0034        super();
0035        itsListener = toInform;
0036
0037        this.setLayout(      new BorderLayout());
0038        this.setMinimumSize( new Dimension( 100, 0));
0039        this.setBorder(      new LineBorder( Color.black, 2));
0040
0041        nameLabel = new JLabel();
0042        nameLabel.setHorizontalAlignment(
0043                                      SwingConstants.CENTER);
0044
0045        cardDropArea = new CardDropArea( this);
0046        cardDropArea.setBorder( droppableBorder);
0047
0048        scoreLabel = new JLabel( "0");
0049        scoreLabel.setHorizontalAlignment(
0050                                      SwingConstants.CENTER);
0051
0052        this.add( nameLabel,    BorderLayout.NORTH);
0053        this.add( cardDropArea, BorderLayout.CENTER);
0054        this.add( scoreLabel,   BorderLayout.SOUTH);
0055     } // End MemoryGameCardDropArea constructor.
```

The first step of the constructor, after it has called its **super** constructor, is to store the *toInform* argument in the *itsListener* attribute. Then, on lines 0037 to 0039, the component is configured to have a BorderLayout management policy, to request a minimum width of 100 pixels and to show itself with a two-pixel-wide black border.

The *nameLabel*, which will appear at the top of the component, is constructed and configured on lines 0041 to 0042. Following this, the *cardDropArea* is constructed and configured on lines 0045 and 0046. The argument to the *CardDropArea* constructor ensures that the ChangeEvents dispatched by it when a card is dropped upon it will be received by **this** component. Lines 0048 to 0050 construct and configure the *scoreLabel*, which will appear at the bottom of the component, before the constructor concludes by assembling the three contained components in their appropriate positions. The implementation of the state setting and getting methods is as follows.

```
0057     public void setName( String newName) {
0058        super.setName( newName);
0059        nameLabel.setText( newName);
0060     } // End setName.
0061
0062
0063     public void setScore( int newScore) {
0064        theScore = newScore;
0065        scoreLabel.setText(
0066                        (new Integer( theScore)).toString());
0067        if ( theScore == 0) {
0068           cardDropArea.removeAll();
0069           this.repaint();
0070        } // End if.
0071     } // End setScore.
0072
0073     public int getScore() {
0074        return theScore;
0075     } // End getScore
0076
0077
0078     public void setDroppable( boolean yesOrNo) {
0079        cardDropArea.setDroppable( yesOrNo);
0080        if ( yesOrNo) {
0081           cardDropArea.setBorder( droppableBorder);
0082        } else {
0083           cardDropArea.setBorder( nonDroppableBorder);
0084        } // End if.
0085     } // End setDroppable.
0086
0087     public boolean isDroppable() {
0088        return cardDropArea.isDroppable();
0089     } // End isDroppable.
```

The name attribute inherited from the AWT Component class is set on line 0058 before being installed as the text resource of the *nameLabel* to make it visible to the user. Likewise the *setScore()* method not only changes the value of the *theScore* attribute but also installs it as a String in the *scoreLabel*. If the score is ever set to zero, which should only happen if the component is being reset for another game, it also removes all the cards contained within the *cardDropArea* and repaints the component so that they will disappear. The *setDroppable()* method not only calls the *cardDropArea setDroppable()* method but also installs the appropriate border into that component to make its state obvious to the user. The only remaining method, *stateChanged()*, is implemented as follows.

```
0093     public void stateChanged( ChangeEvent event) {
0094        itsListener.actionPerformed(
0095              new ActionEvent( this,
0096                               ActionEvent.ACTION_PERFORMED,
0097                               "dropped"));
```

```
   } // End stateChanged.

} // End class MemoryGameCardDropArea.
```

The *stateChanged()* method will be called from the encapsulated *cardDropArea* component when the user changes its state by dropping a card onto it. It propagates this information as an ActionEvent whose actionCommand attribute contains *dropped* to *itsListener*. The consequences of the dispatch of this event will be explained in the revised *MemoryGame actionPerformed()* method which will be described shortly.

The revised *MemoryGame* will have to have all references to *ActiveCard* replaced with *DragCard*, as will the two *CardLoader* classes. It will also require a number of new attributes to support the two drop targets and the two players, as follows.

```
0045  private MemoryGameCardDropArea leftDropArea  = null;
0046  private MemoryGameCardDropArea rightDropArea = null;
0047
0048  private static final int LEFT_PLAYER   = 0;
0049  private static final int RIGHT_PLAYER  = 1;
0050  private int              currentPlayer = LEFT_PLAYER;
0051
0052  private String leftPlayerName    = "left";
0053  private String rightPlayerName   = "right";
0054  private String currentPlayerName = leftPlayerName;
```

The two *MemoryGameCardDropArea* instances are declared on lines 0045 and 0046. Lines 0048 to 0050 then declare the manifests and attributes that indicate the current player and this is complemented by the names of the players in the remaining declarations. An additional manifest game state value, called *DRAG_OFF*, will also be required, as suggested by the STD.

The *startGame()* method will require significant changes in order to construct an interface which includes the two drop targets. The revised implementation commences as follows.

```
     private void startGame() {

        if ( gameArea == null) {

        Box gameBox = Box.createHorizontalBox();

           gameArea = new MemoryGameArea(
                             cardLoader.getCards(),
                             cardLoader.getGridDimension());
           numberOfPairs =  cardLoader.getCards().length /2;
           this.getContentPane().removeAll();

           leftDropArea  = new MemoryGameCardDropArea(this);
           leftDropArea.setDroppable( true);
           rightDropArea = new MemoryGameCardDropArea(this);
           rightDropArea.setDroppable( false);

           gameBox.add( leftDropArea);
           gameBox.add( gameArea);
```

```
0096             gameBox.add( rightDropArea);
0097
0098             this.getContentPane().add( gameBox);
0099             this.validate();
0100             cardLoader = null;
0101          } // End if.
```

This part of the method will be executed when the game is played for the first time when the *gameArea* still has the value **null**. The three major components, the *gameArea*, the *leftDropArea()* and the *rightDropArea()* will be installed into a horizontal Box called *gameBox* which is constructed on line 0081. A Box is a very lightweight Container that will lay out its children in a horizontal (or vertical) alignment, affording each the minimum width (or height) it requests. Lines 0083 to 0086, which construct the *gameArea* and remove the opening screen, are unchanged from the previous version.

The method continues, on lines 0089 to 0092, by constructing and configuring the two *MemoryGameCardDropArea* instances, the argument to the constructors indicating that it is **this** component to which they should dispatch their ActionEvents. Once constructed the three components are added, in their left to right sequence, to the *gameBox*, which is then added to the applet's contentPane before being validated to ensure that it becomes visible. The method concludes as follows.

```
0103          gameArea.prepareGame();
0104          leftDropArea.setName(  leftPlayerName);
0105          leftDropArea.setScore( 0);
0106          rightDropArea.setName( rightPlayerName);
0107          rightDropArea.setScore( 0);
0108          numberOfPairsTurned = 0;
0109          gameArea.setNoCardTurnedState( currentPlayerName);
0110          gameState = NO_CARD_TURNED;
0111       } // End startGame.
```

This part of the method configures, or reconfigures, the interface for the start of a new game. It includes, on line 0109, a call to the *gameArea setNoCardTurnedState* method which will invite the current player to click upon a card to turn it. The behavior of the artifact is determined by the code in the revised *actionPerformed()* method which will be presented in its entirety, commencing as follows.

```
0119      public synchronized void actionPerformed(
0120                                         ActionEvent event) {
0121
0122      String command = event.getActionCommand();
0123      MemoryGameCardDropArea dropArea = null;
0124
0125
0126         if ( command == null) {
0127            command = new String("time-out");
0128         } // End if.
0129
0130         if ( command.equals( "loading") ){
0131             openingScreen.updateProgress(
0132                           cardLoader.getProportionLoaded());
0133
```

```
0134         } else if ( command.equals( "loaded")) {
0135
0136             if ( ! cardLoader.areCardsLoaded()) {
0137                this.showLoadFailure();
0138             } else {
0139                this.startGame();
0140             } // End if.
```

This part of the method differs from the previous version only in the declaration of an additional local variable, *dropArea*, of the *MemoryGameCardDropArea* class, on line 0123. The substantive part of the fragment deals with the events that are received up to the point where the game starts. The method continues as follows.

```
0142         } else if ( command.equals( "clicked")) {
0143
0144           if ( gameState == NO_CARD_TURNED) {
0145              gameArea.setAllCardsDisabled();
0146              firstCardTurned = (DragCard) event.getSource();
0147              firstCardTurned.turnOver();
0148           } else if ( gameState == ONE_CARD_TURNED) {
0149              gameArea.setAllCardsDisabled();
0150              secondCardTurned = (DragCard) event.getSource();
0151              secondCardTurned.turnOver();
0152           } // End if.
```

This fragment is unchanged from the corresponding part of the previous version and responds to the user clicking upon a card in order to turn it over. The *actionPerformed()* method continues as follows.

```
0151        } else if ( command.equals( "turned")) {
0152
0153           if ( gameState == NO_CARD_TURNED) {
0154              gameArea.setAllPlayableEnabled();
0155              gameArea.setOneCardTurnedState(
0156                                            currentPlayerName);
0157              gameState = ONE_CARD_TURNED;
0158           } else if ( gameState == ONE_CARD_TURNED) {
0159              gameState = TWO_CARD_TURNED;
0160              delay.start();
0161           } // End if.
```

This part of the method is activated when a card finishes turning. If it is the first of a pair of cards, on lines 0155 and 0156, the user will be invited to turn a second card. Otherwise the *delay* is started so that the user can contemplate the matching of the two cards before continuing. The method continues as follows.

```
0163        } else if ( command.equals( "time-out")) {
0164
0165           if ( firstCardTurned.getName().equals(
0166                             secondCardTurned.getName()) ){
0167              firstCardTurned.setDraggable(  true);
0168              secondCardTurned.setDraggable( true);
0169              gameState = DRAG_OFF;
0170              gameArea.setDragOffState( currentPlayerName);
0171           } else {
0172              firstCardTurned.showBack();
0173              secondCardTurned.showBack();
0174              this.switchPlayers();
0175              gameState = NO_CARD_TURNED;
0176              gameArea.setNoCardTurnedState(
0177                                       currentPlayerName);
0178              gameArea.setAllPlayableEnabled();
0179           } // End if.
```

This part of the method is activated when the *delay* timer times out. The first part of the selection, on lines 0167 to 0170, will be effected if a matched pair has been turned over. Lines 0167 and 0168 make both turned cards *draggable*, following which the *gameState* is updated and the current player is invited to drag off one of the cards. If the two cards do not match, lines 0172 and 0173 flip the cards back over, following which the players are switched by calling a method which will be described below, and the new current player is invited to turn over a first card. The method concludes as follows.

```
0181        } else if ( command.equals( "dropped")) {
0182              firstCardTurned.remove();
0183              secondCardTurned.remove();
0184              numberOfPairsTurned++;
0185              dropArea = (MemoryGameCardDropArea)
0186                                       event.getSource();
0187              dropArea.setScore( dropArea.getScore() +1);
0188           if ( numberOfPairsTurned == numberOfPairs-1) {
0189              this.setFinishedState();
0190           } else {
0191              this.switchPlayers();
0192              gameState = NO_CARD_TURNED;
0193              gameArea.setNoCardTurnedState(
0194                                      currentPlayerName);
0195              gameArea.setAllPlayableEnabled();
0196           } // End if.
0197        } // End if.
0198      } // End actionPerformed.
```

An ActionEvent containing the actionCommand *dropped* will be dispatched to this component when a player drops a card onto a *MemoryGameDropCardArea*. The first consequence of this is to *remove()* both turned cards and increment the *numberOfPairsTurned* attribute. Lines 0185 to 0187 then increment the *score* attribute of the drop target which was the source of the event. The remainder of this fragment will, if the penultimate pair has just been removed, call an updated *setFinishedState()* method which will credit the current player's score with the remaining pair of cards. Otherwise the players are switched, the game state set and the new current player invited to turn over a first card. The *switchPlayers()* method is implemented, without comment, as follows.

```
0232      public void switchPlayers() {
0233         if ( currentPlayer == LEFT_PLAYER) {
0234            currentPlayer = RIGHT_PLAYER;
0235            currentPlayerName = rightPlayerName;
0236            rightDropArea.setDroppable( true);
```

```
            leftDropArea.setDroppable(  false);
         } else {
            currentPlayer = LEFT_PLAYER;
            currentPlayerName = leftPlayerName;
            leftDropArea.setDroppable(  true);
            rightDropArea.setDroppable( false);
         } // End if.
      } // End switchPlayers.
```

This completes the implementation of the third revision of the *MemoryGame*, which supplies D&D capability. The remainder of this chapter will introduce a fourth version which will add an application-level main menu offering the user the opportunity to exit from the game, to choose which set of cards to use for a game and to indicate the names of the two players.

6.3 The *MemoryGameMenuBar*

The provision of an application-level main menu has become a ubiquitous part of virtually every artifact intended for use within a windowing environment. In this part of the chapter a main menu bar will be provided for the *MemoryGame* artifact. Figure 6.10 shows the simulated appearance of the menu system that will be supplied. The system consists of three main menus conventionally labeled *File*, *Options* and *Help,* the underlining of the initial capital letter indicating that there is a keyboard *mnemonic* that can be used to post the associated menu.

The *File* menu contains two menu buttons labeled *Finish...* and *Exit...*, the three dots following the label indicating to the user that a dialog will be posted when they are activated. They also have mnemonic indicators, which will only be effective when the menu is posted. The buttons also have *accelerators* associated with them, indicated by the *Ctrl+F* and *Ctrl+E* labels; these indicate that the dialogs can be posted (without having to first post the menu and activate the button) by pressing the indicated keys on the keyboard. The two elements on this menu are separated by a line, known as a *separator,* to assist the user in distinguishing between them.

The first entry on the *Options* menu is labeled *Names...*, which also indicates that a dialog will be posted when this is activated. The dialog allows the user to enter the names of the left- and right-hand players, which will subsequently be displayed in the drop areas and in the prompt at the bottom of the game area. The second entry on the *Options* menu

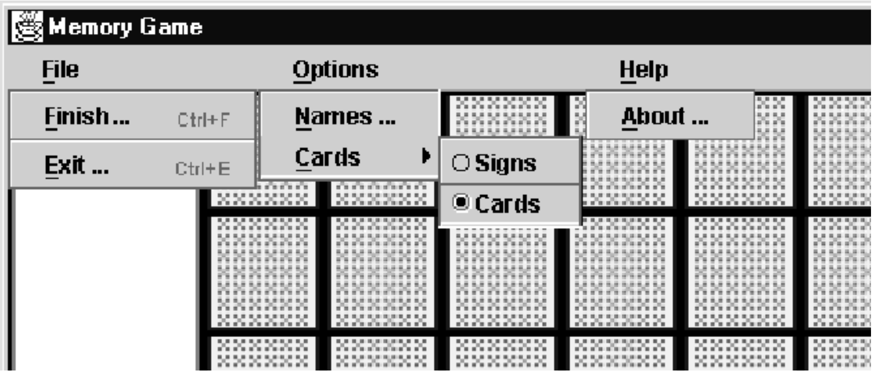


Figure 6.10 *MemoryGame*: simulated menu structure.

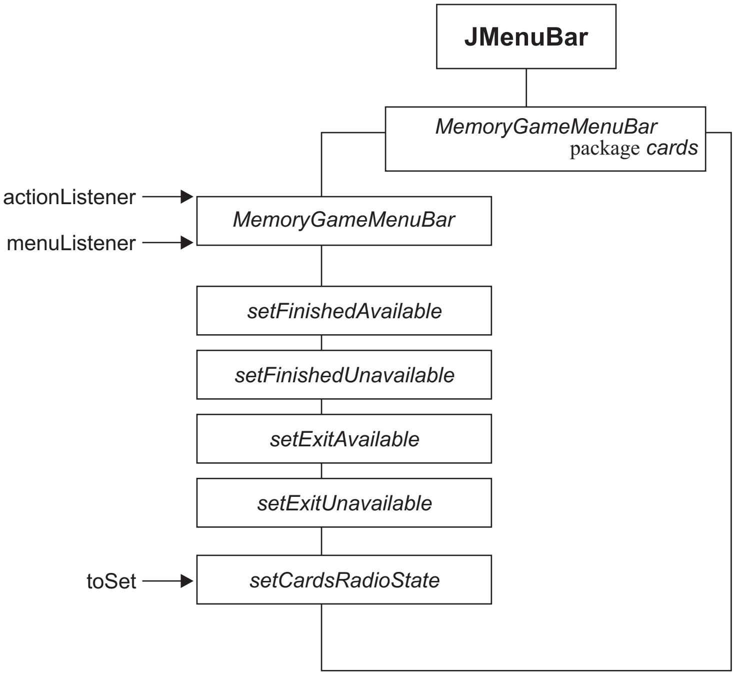


Figure 6.11 *MemoryGameMenuBar* class diagram.

is labeled *Cards*, and is followed by a right-facing arrow which indicates that when it is activated a further *cascading menu* will be posted. The cascading menu contains two radio menu buttons that allow the user to choose between the road *Signs* and playing *Cards* sets of images. The *Help* menu contains a single entry labeled *About...*, which leads to a dialog informing the user about who developed the artifact and its version number.

The menu system is provided by an instance of the *MemoryGameMenuBar* class whose class diagram is given in Figure 6.11. It shows that the constructor and all of the methods are protected, as this class is so tightly coupled with the *MemoryGame* class. The *MemoryGameMenuBar* class extends the JMenuBar class and so is a menu bar and can be installed as such. The constructor requires two listener arguments: an ActionListener which will be informed when any of the menu items are activated and a MenuListener which will be informed when a menu is posted or unposted.

The class has a number of attributes, corresponding to each of the elements on the menu system, declared as follows.

```
package cards;

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

class MemoryGameMenuBar extends JMenuBar {

```

```
private JMenu                fileMenu     = null;
private JMenuItem            exitButton   = null;
private JMenuItem            finishButton = null;

private JMenu                optionMenu   = null;
private JMenuItem            nameButton   = null;
private JMenu                cardsMenu    = null;
private ButtonGroup          cardsGroup   = null;
private JRadioButtonMenuItem signButton   = null;
private JRadioButtonMenuItem cardsButton  = null;

private JMenu                helpMenu     = null;
private JMenuItem            aboutButton  = null;
```

The JMenu elements will be used to supply the three buttons that exist on the main menu (*File*, *Options* and *Help*) and also the *Cards* button on the *Options* menu. These components will post a *menu pane* containing other components when they are activated. The JMenuItem elements will be used for the items which cause a dialog to be posted. The JRadioButtonMenuItem elements supply the two radio button elements on the *Cards* menu and will be associated with a ButtonGroup to cause them to behave as a single group.

At various points in the playing of a game the finished and exit options will not be available. The finished option should only be available when the game is being played and the exit option when a game has finished. Accordingly, the first four methods are supplied to set, or reset, the sensitivity of these items. Likewise it is essential to ensure that the settings of the radio button items always reflect the state of the artifact, and the *setCardsRadioState()* method is supplied to effect this.

The constructor will be presented in several stages, corresponding to the three menus. The first part of its implementation is as follows.

```
protected MemoryGameMenuBar( ActionListener sendTo,
                   MenuListener   sendMenuEventsTo) {
   super();

   fileMenu = new JMenu( "File");
   fileMenu.setName( "file");
   fileMenu.setMnemonic( 'F');
   fileMenu.addMenuListener( sendMenuEventsTo);

   finishButton = new JMenuItem(   "Finish");
   finishButton.setActionCommand(  "finish");
   finishButton.addActionListener( sendTo);
   finishButton.setMnemonic( 'F');
   finishButton.setAccelerator(
                   KeyStroke.getKeyStroke(
                       KeyEvent.VK_F, Event.CTRL_MASK));
   fileMenu.add( finishButton);

   fileMenu.addSeparator();

   exitButton = new JMenuItem(   "Exit");
   exitButton.setActionCommand(  "exit");
```

```
0055        exitButton.addActionListener( sendTo);
0056        exitButton.setMnemonic( 'E');
0057        exitButton.setAccelerator(
0058                      KeyStroke.getKeyStroke(
0059                            KeyEvent.VK_E, Event.CTRL_MASK));
0060        fileMenu.add( exitButton);
0061
0062        this.add( fileMenu);
```

Once the **super** constructor has been called the *File* menu pane is prepared on lines 0037 to 0040. It is constructed with the argument indicating the label it is to display, has a name attribute established, so that its events can be differentiated from others, has a mnemonic established and has the menuListener registered with it. This last stage will ensure that notifications are dispatched when the menu pane is posted and unposted.

JComponent → AbstractButton → javax.swing.JMenuItem

```
JMenuItem( String text)
JMenuItem( Icon   icon)
JMenuItem( String text, Icon icon)
JMenuItem( String text, int  mnemonic)
```

Constructors allowing various combinations of *text*, *icon* and *mnemonic* to be specified.

```
public void      setAccelerator( KeyStroke keyStroke)
public KeyStroke getAccelerator()
```

Setter and getter on the keyStroke attribute. The actionListener and mnemonic attributes are inherited from AbstractButton.

JComponent → AbstractButton → JMenuItem → javax.swing.JMenu

```
JMenu()
JMenu( String label,)
JMenu( String label, boolean tearOff)
```

Various constructors, a *tearOff* menu can be dragged away from the menu bar and left continually available.

```
public JMenuItem add(    JMenuItem item)
public JMenuItem insert( JMenuItem item, int position)
public void      addSeparator()
public void      remove( JMenuItem item)
public void      remove( int       position)
public JMenuItem getItem( int position)
public int       getItemCount()
public boolean   isTearOff()
public void      addMenuListener( MenuListener listener)
```

Methods to maintain the items on the menu. The menu will dispatch MenuEvents to its menuListener whenever it is posted or unposted.

The remaining parts of this fragment populate the *File* menu pane with the items that it contains. The *Finish* button is constructed, configured and added to the *fileMenu* on lines 0042 to 0049. The JMenuItem is constructed, has an actionCommand established, has the *sendTo* ActionListener registered with it, has a mnemonic and accelerator established and is finally added to the *fileMenu*. As it is the first element to be added to the menu it will always appear at the top.

JComponent → javax.swing.JMenuBar

```
public JMenuBar()
```

The only constructor.

```
public JMenu add( JMenu menuToAdd)
public JMenu getMenu( int index)
public int   getMenuCount()
public JMenu getHelpMenu()
public void  setHelpMenu( JMenu theHelpMenu)
```

Methods to maintain the menus on the main menu bar; the help menu is always placed at the right.

Line 0051 inserts a separator into the menu before lines 0053 to 0060 construct, configure and add the *exitButton* to the *fileMenu* in an essentially identical manner. As the *fileMenu* is now complete, on line 0062, it is added to **this** JMenuBar and, as it is the first element added, will always appear at the left. When either of the JMenuItems are activated by the user an ActionEvent, containing the appropriate actionCommand, will be dispatched to the *sendTo* ActionListener. The four methods, which control the sensitivity of these two buttons, are implemented as follows.

```
0112     protected void setFinishAvailable(){
0113        finishButton.setEnabled( true);
0114     } // End setFinishAvailable.
0115
0116     protected void setFinishUnavailable(){
0117        finishButton.setEnabled( false);
0118     } // End setFinishUnavailable.
0119
0120     protected void setExitAvailable(){
0121        exitButton.setEnabled( true);
0122     } // End setExitAvailable.
0123
0124     protected void setExitUnavailable(){
0125        exitButton.setEnabled( false);
0126     } // End setExitUnavailable.
```

The next part of the constructor, as follows, constructs and configures the *Options* menu.

```
0065        optionMenu = new JMenu(   "Options");
0066        optionMenu.setName( "options");
0067        optionMenu.setMnemonic( 'O');
0068        optionMenu.addMenuListener( sendMenuEventsTo);
```

```
0069
0070         nameButton= new JMenuItem(   "Names ... ");
0071         nameButton.setActionCommand(  "names");
0072         nameButton.addActionListener( sendTo);
0073         nameButton.setMnemonic( 'N');
0074         optionMenu.add( nameButton);
0075
0076
0077         cardsGroup = new ButtonGroup();
0078         cardsMenu  = new JMenu(   "Cards");
0079         cardsMenu.setMnemonic( 'C');
0080
0081         signButton = new JRadioButtonMenuItem( "Signs");
0082         signButton.setActionCommand(  "signs");
0083         signButton.addActionListener( sendTo);
0084         cardsGroup.add( signButton);
0085         cardsMenu.add(  signButton);
0086
0087         cardsButton = new JRadioButtonMenuItem( "Cards");
0088         cardsButton.setActionCommand(  "cards");
0089         cardsButton.addActionListener( sendTo);
0090         cardsGroup.add( cardsButton);
0091         cardsMenu.add(  cardsButton);
0092
0093         optionMenu.add( cardsMenu);
0094
0095         this.add( optionMenu);
```

The first part of this fragment, between lines 0065 and 0074, is comparable to the previous fragment. It constructs and configures the *optionMenu* and then constructs, configures and adds the *nameButton* to it. On line 0077 the ButtonGroup is prepared, and on lines 0078 to 0079 the *cardsMenu* is constructed and configured. Lines 0081 to 0084 construct and configure the first JRadioButtonMenuItem, including associating it with the *cardsGroup*, and it is added to the cascading *cardsMenu* on line 0085.

JComponent → AbstractButton → JMenuItem → javax.swing.JRadioButtonMenuItem

```
JRadioButtonMenuItem( Icon icon)
JRadioButtonMenuItem( Icon icon,   boolean selected)
JRadioButtonMenuItem( String text)
JRadioButtonMenuItem( String text, boolean selected)
JRadioButtonMenuItem( String text, Icon icon)
JRadioButtonMenuItem( String text, Icon icon, boolean selected)
```

Constructors allowing various combinations of text and icon to be specified. By default a JRadioButtonMenuItem is not selected unless *selected* is specified **true**.

All other methods, most significantly actionCommand and actionListener, are inherited. A set of JRadioButtonMenuItems that have been added to the same ButtonGroup will allow only one of them to be set.

A JRadioButtonMenuItem instance will dispatch an ActionEvent when it is selected; this differs from JRadioButton instances which dispatch ItemEvents. Lines 0087 to 0091 construct, configure and add the second menu radio button to the *cardsMenu*. Both of the JRadioButtonMenuItem instances are constructed without specifying which should be selected by default. The *setCardsRadioState()* method, described below, will be called before the menu is posted for the first time to ensure that one of them is set. Once the two menu radio buttons have been added to the *optionMenu*, on line 0095 the *optionMenu* is added to **this** menu bar where it will appear alongside the *fileMenu*. The implementation of the *setCardsRadioState()* method is as follows.

```
0129     protected void setCardsRadioState( int toSet) {
0130        if ( toSet == MemoryGame.SIGN_CARDS) {
0131           signButton.setSelected(  true);
0132           cardsButton.setSelected( false);
0133        } else {
0134           signButton.setSelected(  false);
0135           cardsButton.setSelected( true);
0136        } // End if.
0137     } // End setCardsRadioState.
```

The manifest value *SIGN_CARDS* is supplied by the upgraded *MemoryGame* class and is used in conjunction with the *toSet* argument to decide which pattern of radio button selections to provide. The final part of the *MemoryGameMenuBar* constructor supplies the *Help* menu and is implemented as follows.

```
0098        helpMenu = new JMenu( "Help");
0099        helpMenu.setMnemonic( 'H');
0100
0101        aboutButton = new JMenuItem(   "About ... ");
0102        aboutButton.setActionCommand(  "about");
0103        aboutButton.addActionListener( sendTo);
0104        aboutButton.setMnemonic( 'A');
0105        helpMenu.add( aboutButton);
0106
0107        this.setHelpMenu( helpMenu);
0108        this.add( helpMenu);
0109     } // End MemoryGameMenuBar constructor.
```

The only significant difference in this fragment is the call of the JMenuBar setHelpMenu() method on line 0107. This will ensure that the nominated menu is always posted in the rightmost position and in some environments, for example X/Motif, will be posted at the extreme right-hand side of the artifact's window.

The upgraded *MemoryGame* class can now declare an attribute called *menuBar* of the *MemoryGameMenuBar* class. The *menuBar* is constructed during the *startGame()* method and placed into the interface by calling the JApplet setJMenuBar() method, as follows.

```
     // New attribute declarations.
0058 private MemoryGameMenuBar menuBar = null;
0059
0060 protected static final int SIGN_CARDS      = 0;
0061 protected static final int PLAYING_CARDS   = 1;
```

```
0062 private int currentCards = SIGN_CARDS;
              // Within the setMenu() method.

0102          if ( menuBar == null) {
0103             menuBar = new MemoryGameMenuBar( this, this);
0104             this.setJMenuBar( menuBar);
0105          } // End if.
0112        menuBar.setCardsRadioState( currentCards);
```

The *currentCards* attribute will always indicate the set of cards that are in use. The *startGame()* method may be called several times but the *menuBar* need only be constructed and added once and so its constructor, on line 0103, is guarded. The menu radio buttons are set as appropriate on line 0112, before the user sees the menu for the first time. In order for **this** *MemoryGame* instance to be supplied as the second argument to the *menuBar* constructor on line 0103 the class has to declare that it implements the *MenuListener* interface and supplies the mandated methods as follows.

```
0454      public void menuCanceled( MenuEvent event) {
0455         // Do nothing.
0456      } // End menuCanceled
0457
0458      public void menuDeselected( MenuEvent event) {
0459         // Do nothing.
0460      } // End  menuCanceled
0461
0462      public void menuSelected( MenuEvent event) {
0463
0464      Component source = (Component) event.getSource();
0465      String    name   = source.getName();
0466
0467         if ( name.equals( "file")) {
0468            if ( gameState != FINISHED) {
0469               menuBar.setFinishAvailable();
0470            } else {
0471               menuBar.setFinishUnavailable();
0472            } // End if.
0473
0474            if ( gameState == FINISHED    ||
0475               numberOfPairsTurned == 0 ){
0476               menuBar.setExitAvailable();
0477            } else {
0478               menuBar.setExitUnavailable();
0479            } // End if.
0480         } // End if.
0481      } // End menuSelected
```

javax.swing.event.MenuListener, interface

```
public void menuSelected(   MenuEvent event)
public void menuDeselected( MenuEvent event)
public void menuCanceled(   MenuEvent event)
```

Methods called as the user interacts with the menu.

Object → EventObject → javax.swing.event.MenuEvent

```
public MenuEvent( Object source)
```

The only constructor; the getSource() method is inherited from EventObject.

The *menuCanceled()* and *menuDeselected()* methods will be called when a menu is abandoned without any action being effected or when it is un-posted after an action has been effected, respectively. In this implementation they are supplied as dummy methods. The *menuSelected()* method is called whenever the menu is posted and commences, on line 0467, with a guard that ensures that the *File* menu is being posted. The substantive part of the method consists of two two-way selections which ensure that the availability of the *Finished* and *Exit* buttons is set as appropriate to the state of the game. The effect is that every time the *File* menu is posted the two items that it contains are configured appropriately before they become visible to the user.

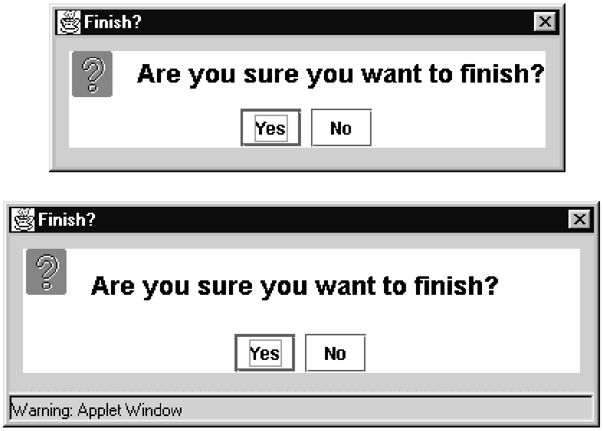
When the *Finished* menu item is activated it will dispatch an ActionEvent whose actionCommand contains *finished* to an upgraded *actionPerformed()* method which intercepts it and calls a new method called *doFinish()*, as follows.

```
              // New branch in actionPerformed()
0210          } else if ( command.equals( "finish")) {
0211             this.doFinish();
0229       private void doFinish() {
0230
0231       int    finishReply   = 0;
0232       String finishMessage = "Are you sure you want to finish?";
0233       String finishTitle   = "Finish?";
0234
0235          finishReply = JOptionPane.showConfirmDialog(
0236                                            this, finishMessage,
0237                                            finishTitle,
0238                                            JOptionPane.YES_NO_OPTION);
0239          if ( finishReply == JOptionPane.YES_OPTION) {
0240             this.setFinishedState();
0241          } // End if.
0242       } // End doFinish.
```

The *doFinish()* method posts a standard modal *Yes/No* confirmation dialog asking users whether they are sure they want to finish the game. If the user replies *Yes*, on line 0240, it calls the *MemoryGame setFinishedState()* which, as before, will post a dialog asking whether the user wants to play again. The appearance of this dialog is illustrated in Figure 6.12. The lower illustration shows how the dialog may appear when posted from a browser. It includes a warning at the bottom informing the user that it is an applet window. The reason for this is to prevent a malicious applet loaded across the Internet from masquerading as locally produced dialog, for example one that invites the user to type in a system password.

The *Exit* button on the *File* menu will, when activated, dispatch an ActionEvent containing the actionCommand *exit*, which will be intercepted in the *actionCommand()* method and causes the *doExit()* method, as follows, to be called.

```
0244       private void doExit() {
0245
```



Are you sure you want to finish?

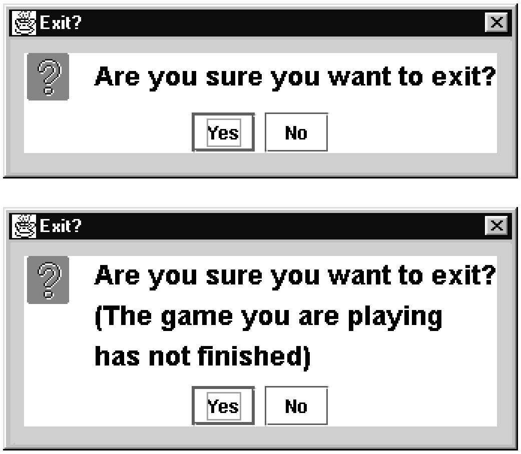
Are you sure you want to finish?

Figure 6.12 *MemoryGame* finish dialog.

```
0246      int    exitReply      = 0;
0247      String exitMessage    = "Are you sure you want to exit?";
0248      String warningMessage = "\n(The game you are playing" +
0249                               "\nhas not finished)";
0250      String fullMessage    = null;
0251      String exitTitle      = "Exit?";
0252
0253        if ( gameState != FINISHED) {
0254           fullMessage = exitMessage + warningMessage;
0255        } else {
0256           fullMessage = exitMessage;
0257        } // End if.
0258
0259        exitReply = JOptionPane.showConfirmDialog(
0260                                      this, fullMessage,
0261                                      exitTitle,
0262                                      JOptionPane.YES_NO_OPTION);
0263        if ( exitReply == JOptionPane.YES_OPTION) {
0264           this.setVisible( false);
0265           System.exit( 0);
0266        } // End if.
0267      } // End doExit.
```

The appearance of this dialog is illustrated in Figure 6.13. For reasons that will be explained below it is possible for this dialog to be posted when the game has not finished, despite the guard on the posting of the *File* menu which disables the *Exit* menu item. Accordingly, it has two possible messages which are produced in the selection structure on lines 0253 to 0257.

If the user presses the *Yes* button lines 0264 and 0265 will be executed. Line 0264 will cause the artifact to disappear from a browser, or from the desktop if it is executing as an application, and line 0265 will halt the process entirely. When the artifact is executing as an application, the window frame controls, as shown for example in Figure 6.7, will allow the user to maximize, restore, iconify, deiconify or close the window. This last possibility,



Exit?

Are you sure you want to exit?

Exit?

Are you sure you want to exit? (The game you are playing has not finished)

Figure 6.13 *MemoryGame* exit dialog.

effected by pressing the button with a cross in it in Figure 6.7, will normally close the window and terminate the application. It is not acceptable for this to happen automatically without confirmation at any stage in the game. To prevent the window from closing automatically the WindowEvent dispatched to the artifact can be intercepted, and the closing of the window canceled, if the *MemoryGame* is upgraded to implement the WindowListener interface.

java.awt.event.WindowListener

```
public void windowOpened(      WindowEvent event)
public void windowActivated(   WindowEvent event)
public void windowDeactivated( WindowEvent event)
public void windowClosed(      WindowEvent event)
public void windowClosing(     WindowEvent event)
public void windowIconified(   WindowEvent event)
public void windowDeiconified( WindowEvent event)
```

Methods called as the window is opened, closed, activated and iconified.

The WindowListener interface mandates a total of seven methods, called windowOpened(), windowActivated(), windowIconified(), windowDeiconified(), window Deactivated(), windowClosing() and windowClosed(), which are called at appropriate times in the window's life cycle. In order to intercept the notification that the window is about to close, the *MemoryGame* class will have to implement the WindowListener interface, but need only supply a body for the windowClosing() method, as follows. The other six methods can be supplied as empty methods.

```
public void windowClosing( WindowEvent event) {
   this.actionPerformed(
           new ActionEvent( this,
                          ActionEvent.ACTION_PERFORMED,
                          "exit"));
} // End windowClosing.
```

The method dispatches an ActionEvent whose actionCommand attribute contains *exit* to the *actionPerformed()* method, where it will cause the *doExit()* method to post an exit dialog, possibly with the warning that the game has not finished. If the user says *Yes* the closing of the window and the termination of the game will be effected, as described above. If the user says *No* the dialog will be unposted and the game will remain on the desktop. In order for this to happen the *MemoryGame* instance has to be specified as the WindowListener resource of the *MemoryGame*'s *frame* in its *main()* method, as on line 0562 in the following fragment.

```
0554    public static void main( String args[]) {
0555
0556        JFrame     frame   = new JFrame( "Memory Game");
0557        MemoryGame theDemo = new MemoryGame();
0558
0559          theDemo.init();
0560          frame.getContentPane().add(
0561                         theDemo, BorderLayout.CENTER);
0562          frame.addWindowListener( theDemo);
0563          frame.setSize( 800, 450);
0564          frame.setVisible( true);
0565     } // end main.
```

The dialog posted from the *Options* menu *Names* item will be described in the following section. The remainder of this section will describe the changes required in the *MemoryGame* class to respond appropriately to the user selecting one of the two radio buttons on the *Options* menu *Cards* cascading menu. When either of these buttons is selected it will dispatch an ActionEvent containing the actionCommand *signs* or *cards* to the *MemoryGame actionPerformed()* method, which detects and responds to them as follows.

```
0222        } else if ( command.equals( "signs") ||
0223                    command.equals( "cards") ){
0224           this.loadNewCards( command);
```

The *loadNewCards()* method, whose implementation follows, on lines 0320 to 0325, will terminate without doing anything if it is asked to load the set of cards which is already in use. Otherwise, the remainder of the method catenates an appropriate message and asks users, by means of a JOptionPane *Yes/No* dialog, if they really do want to change the cards. If they reply *Yes* then the *doLoadNewCards()* method will be called; otherwise the dialog is unposted and the game can continue with the current cards.

```
0310      private void loadNewCards( String toLoad) {
0311
0312      int    loadReply  = 0;
0313      String loadMessage1  = "Are you sure you want to load\n";
0314      String signMessage   = "the road sign cards?\n";
0315      String cardMessage   = "the playing cards?\n";
0316      String loadMessage2  =
0316                       "(This will finish the current game.)";
0317      String fullMessage   = null;
0318      String loadTitle     = "New Cards?";
0319
0320         if ( ( toLoad.equals( "signs") &&
0321                currentCards == SIGN_CARDS)    ||
0322              ( toLoad.equals( "cards") &&
0323                currentCards == PLAYING_CARDS) ){
0324            return;
0325         } // End if.
0326
0327         if ( toLoad.equals( "signs")) {
0328            fullMessage = loadMessage1 + signMessage +
0329                          loadMessage2;
0330         } else {
0331            fullMessage = loadMessage1 + cardMessage +
0332                          loadMessage2;
0333         } // End if.
0334
0335        loadReply = JOptionPane.showConfirmDialog(
0336                                      this, fullMessage,
0337                                      loadTitle,
0338                                      JOptionPane.YES_NO_OPTION);
0339        if ( loadReply == JOptionPane.YES_OPTION) {
0340           this.doLoadNewCards( toLoad);
0341        } // End if.
0342      } // End loadNewCards.
```

The String naming the set of cards to load is passed as an argument to the *doLoadNewCards()* method so that it can be informed of which set of cards need to be loaded. The method is supported by and manipulates a number of instance attributes that are declared as follows.

```
0060  protected static final int SIGN_CARDS      = 0;
0061  protected static final int PLAYING_CARDS   = 1;
0062  private                int currentCards    = SIGN_CARDS;
0063  private CardLoader         signCardLoader = null;
0064  private CardLoader         playCardLoader = null;
0065  private CardLoader         cardLoader      = null;
```

The two manifest values, declared on lines 0060 and 0061, are protected, as their identity is used by the *setCardsRadioState()* method of the *MemoryGameMenuBar* class, as has already been described. The *currentCards* attribute will always indicate the set of cards which is currently in use and is initialized to load the road sign cards by default. The first two *CardLoader* instances will be initialized to each of the card loader classes, *SignCardLoader* or *PlayingCardLoader*, as described in the previous chapter. The final attribute, *cardLoader* will be maintained to indicate which *cardLoader* should currently be used. The *doLoadNewCards()* method consists of two almost identical parts, one for the road sign cards and one for the playing cards. Only the road sign card part, as follows, will be described.

```
0345      private void doLoadNewCards( String toLoad) {
0346
0347         if ( toLoad.equals( "signs")) {
0348            if ( signCardLoader == null) {
0349               gameState = INITIAL;
0350               signCardLoader = new SignCardLoader( this);
0351               cardLoader = signCardLoader;
0352               try {
0353                  signCardLoader.addActionListener( this);
0354               } catch ( TooManyListenersException exception) {
0355                  // do nothing.
0356               } // end try catch.
0357               currentCards = SIGN_CARDS;
0358               cardLoader.start();
0359               this.getContentPane().removeAll();
0360               gameArea = null;
0361               openingScreen = new MemoryGameOpeningScreen();
0362               this.getContentPane().add( openingScreen);
0363               this.validate();
0364            } else {
0365               gameState = INITIAL;
0366               gameArea = null;
0367               cardLoader = signCardLoader;
0368               currentCards = SIGN_CARDS;
0369               this.startGame();
0370            } // End if.
0371
0372         } else {
                // Essentially identical for the playing cards!

0397         } // End if.
0398      } // End doLoadNewCards.
```

There are two possibilities to be considered when this method is called to load the road sign cards (either they have already been loaded or they have not), and this fragment is structured accordingly. The restarting of the game when the cards have previously been loaded is simpler in structure and will be described first. On lines 0365 to 0369 the *gameState* is reset to *INITIAL*, the *gameArea* is set to **null** and the *cardLoader* and *currentCards* attributes are set as appropriate. Having prepared these attributes the *startGame()* method is called and, as the *gameArea* is **null**, the method will, as has already been explained, construct a new *MemoryGameArea*, installing the cards obtained from *cardLoader*. Once constructed, the interface, including the drop areas, is assembled and the *gameState* set to *NO_CARD_TURNED*, following which the user can start interacting with the game.

The steps to be taken if the road sign cards have not already been loaded are more complex. On lines 0350 and 0351 the *signCardLoader* is constructed as an instance of the *SignCardLoader* class and its identity recorded in *cardLoader* for when the game is subsequently started. Having constructed the loader it has **this** *MemoryGame* instance installed as its action listener before the loading process is started on line 0358. This will ensure that the ActionEvents whose actionCommands contain *loading* and *loaded* will be received by the *MemoryGame actionPerformed()* method as the loading proceeds and concludes.

The remaining part of this branch, on lines 0359 to 0363, removes any existing components from the *contentPane* and installs the *openingScreen MemoryGameOpeningScreen* instance. The effect of this will be that the *loading* messages from the loader will update the appearance of the progress bar, as described in detail in the previous chapter. When the *loaded* message is received the *startGame()* method will be called which will install the cards from the *cardLoader* into a fully configured user interface.

The provision of this method allows the *MemoryGame init()* method to be considerably simplified, as follows.

```
0070       public void init() {
0071          this.setResources();
0072          delay = new Timer( 1000, this);
0073          delay.setRepeats( false);
0074          this.doLoadNewCards( "signs");
0075       } // End init.
```

The final step of the *init()* method, on line 0074, is to call the *doLoadNewCards()* method with a request to load the default (road sign) cards. The effect is identical to that described in detail in the previous chapter, posting and updating an opening screen before commencing the game. If the user subsequently asks to load the playing card set, and confirms this intention on the subsequent dialog, the opening screen will appear again and the game will restart. Following this the switch between the two sets of cards will still require confirmation via the dialog, but will thereafter be immediate, as both sets of cards have been loaded and are available.

6.4 Specialized dialogs

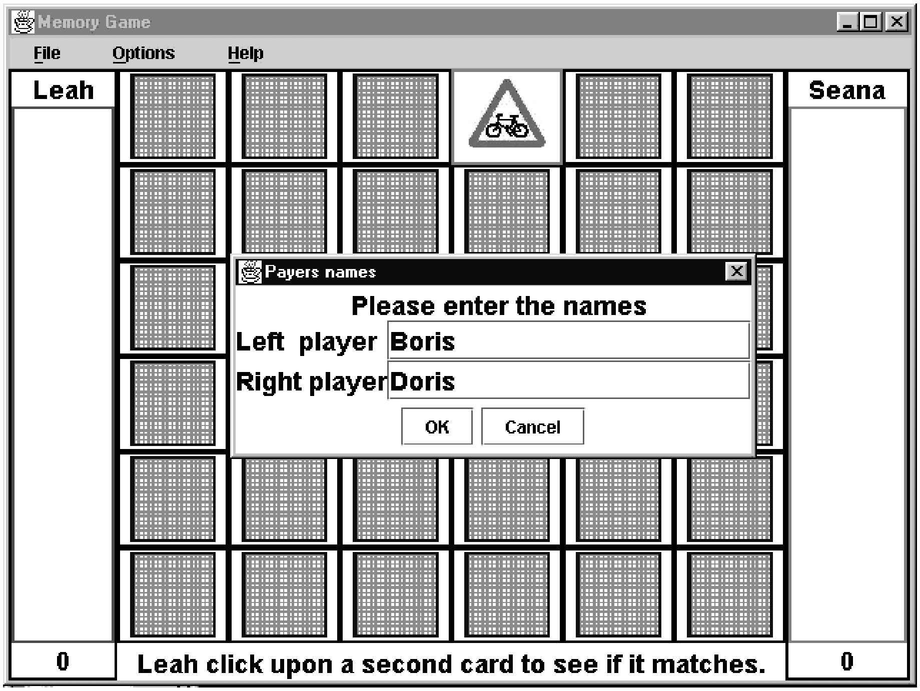
The requirement for the dialog which is posted from the *Names* item on the *Options* menu is to allow the user to input the names of the two players. This requirement cannot be satisfied by the use of any of the JOptionPane standard dialogs and will require the implementation of a specialized dialog class called *MemoryGameNameDialog*. The appearance of the dialog is illustrated in Figure 6.14; it consists of a prompt at the top, two labeled text fields in the middle and *OK* and *Cancel* buttons at the bottom. The user interacts with the dialog by typing in the names of the two users and then pressing the *OK* button. Figure 6.14 also suggests that the dialog has already been used to change the default names (*left* and *right*) to *Leah* and *Seana* and that it is currently Leah's turn.

The class diagram for the *MemoryGameNameDialog* class is given in Figure 6.15. It shows that it is a member of the *cards* package, extends the JDialog class and implements the ActionListener interface, as it has to listen to the events generated by the JButtons it contains. The constructor requires the identity of the ActionListener that it should inform when the user closes the dialog by pressing the *OK* button; it also requires the identity of the parent component that it should post itself centered within.

The two protected inquiry methods retrieve the *leftName* and *rightName* that the user has typed in. The other two methods must be declared public as they override, or implement, methods that have already been declared public. The *setVisible()* method allows the dialog to be posted or unposted and the *actionPerformed()* method is mandated by the ActionListener interface.

The layout diagram of the dialog is given in Figure 6.16 and is implemented by the substantive parts of the constructor which construct, configure and assemble the various components illustrated. The implementation of the constructor, omitting much of the detail, is as follows.

```
protected MemoryGameNameDialog(ActionListener toInform,
                               Component      itsParent){
   super();
```



Leah

Seana

Payers names

Please enter the names

Left player Boris

Right player Doris

0

Leah click upon a second card to see if it matches.

0

Figure 6.14 *MemoryGameNameDialog*: visual appearance.

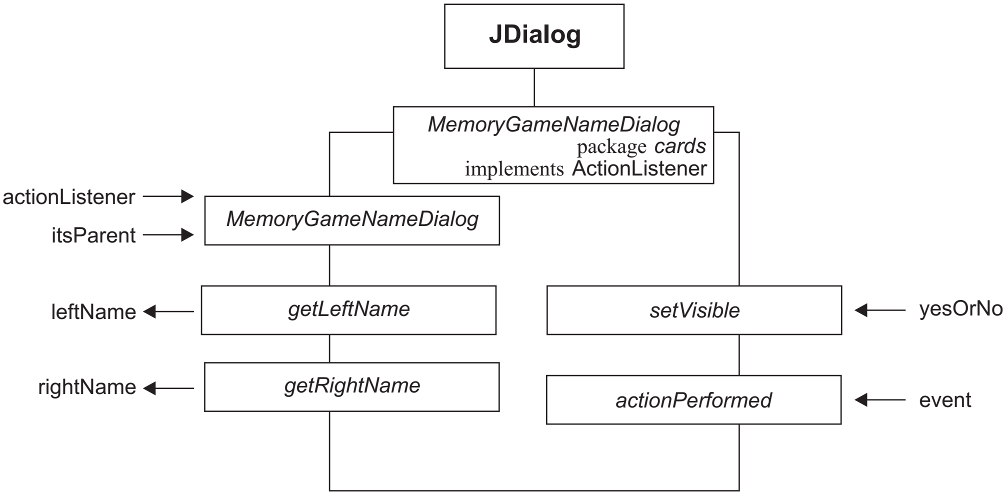


Figure 6.15 *MemoryGameNameDialog*: class diagram.

```
          this.setTitle( "Payers names");
          itsListener = toInform;
          centerWithin = itsParent;

          dialogPanel = new JPanel( new BorderLayout());
--
```

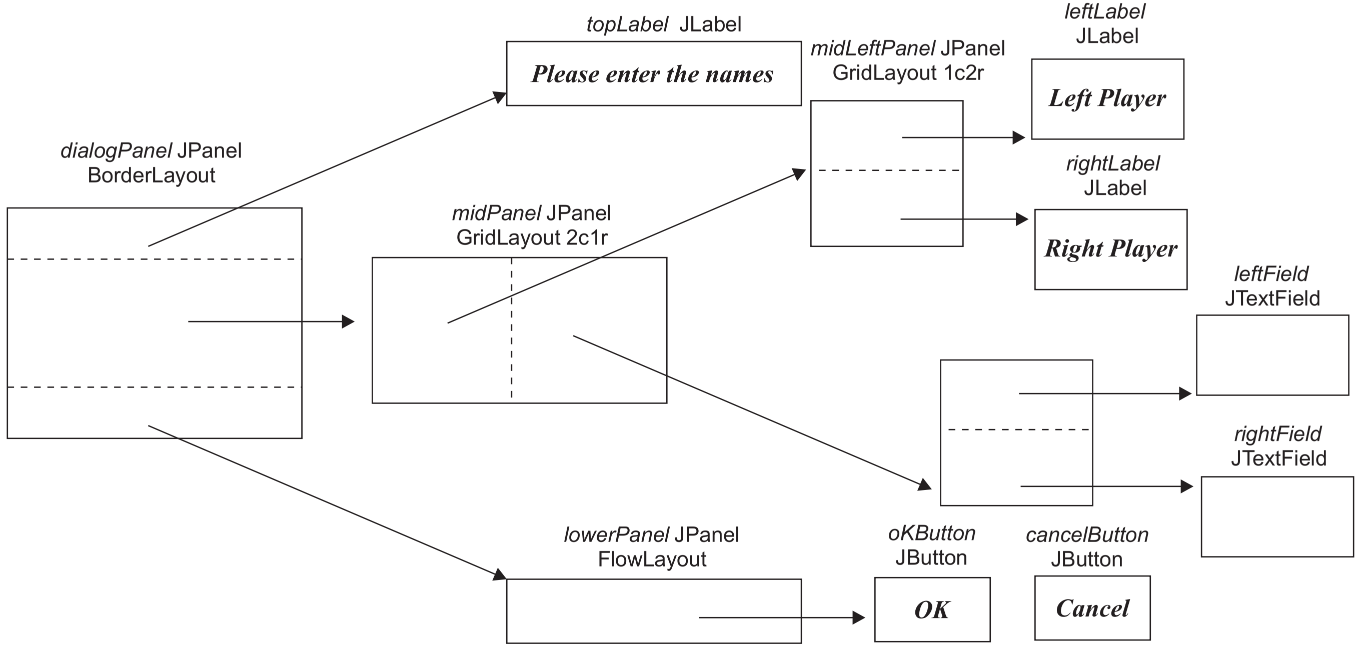


Figure 6.16 *MemoryGameNameDialog*: layout diagram.

```
0079         dialogPanel.add( topLabel,   BorderLayout.NORTH);
0080         dialogPanel.add( midPanel,   BorderLayout.CENTER);
0081         dialogPanel.add( lowerPanel, BorderLayout.SOUTH);
0082         this.getContentPane().add( dialogPanel);
0083         this.pack();
0084      } // End MemoryGameNameDialog constructor.
```

The constructor commences by calling its super, JDialog, constructor, setting a title for the dialog frame and then stores the arguments in their respective instance attributes. Line 0048 starts the construction and configuration of the components shown on the layout diagram and the conclusion of this process is shown on lines 0079 to 0081. A JDialog provides a top-level window, complete with a frame, and just like an applet top-level window it contains a number of panes. On line 0082 the assembled interface is added to the dialog's contentPane and the dialog is then packed in order to force layout negotiations and establish a size for it.

Object → Component → Container → Window → Dialog → javax.swing.JDialog

```
public JDialog()
public JDialog( Dialog owner)
public JDialog( Dialog owner, boolean modal)
public JDialog( Dialog owner, String title)
public JDialog( Dialog owner, String title, boolean modal)
public JDialog( boolean modal)
public JDialog( String title)
public JDialog( String title, boolean modal)
```

Constructors allowing various combinations of *owner*, *title* and *modal* to be specified. A JDialog can be associated with an existing Dialog and will be unposted if it is unposted. By default a dialog is non-modal unless modal is specified **true**; the title is displayed as a part of the frame declorations.

```
public Container getContentPane()
```

Obtains the identity of the contentPane. Other significant methods, including getters and setters for title, visibility, size and location, are inherited. The pack() method, inherited from Dialog, causes layout negotiations to take place.

During the configuration of the components contained within the dialog **this** instance of the *MemoryGameNameDialog* class being constructed was specified as the ActionListener resource of the two JButtons, which were also given appropriate actionCommand resources. The two JTextFields did not have an ActionListener specified, so the *actionPerformed()* method, which follows, can only be called as a consequence of the user pressing one of the buttons.

```
0111      public void actionPerformed( ActionEvent event) {
0112
0113      String command = event.getActionCommand();
0114
0115         this.setVisible( false);
0116         if ( command.equals( "ok")) {
0117            itsListener.actionPerformed(
0118              new ActionEvent( this,
0119                                 ActionEvent.ACTION_PERFORMED,
0120                                "namechange"));
0121        } // End if.
0122     } // End actionPerformed.
```

The first step of the method, on line 0115, is to call the *setVisible()* method with the argument **false**. The consequence of calling this method is that the dialog will be unposted from the desktop. The second step will dispatch an ActionEvent whose actionCommand resource contains *namechange* to the registered listener. The consequences of the receipt of this event by the *MemoryGame actionPerformed()* method will be described shortly. The other public method is the overriding *setVisible()* method, implemented as follows.

```
0085     public void setVisible( boolean yesOrNo) {
0086
0087     Point     itsParentLocation  = null;
0088     Dimension itsParentSize      = null;
0089     Point     itsLocation        = null;
0090     Dimension itsSize            = null;
0091
0092        if ( yesOrNo) {
0093           itsParentLocation =
0094                   centerWithin.getLocationOnScreen();
0095           itsParentSize     = centerWithin.getSize();
0096           itsSize           = this.getSize();
0097           itsLocation       = new Point();
0098
0099           itsLocation.x = itsParentLocation.x +
0100                           itsParentSize.width/2 -
0101                           itsSize.width/2;
0102           itsLocation.y = itsParentLocation.y +
0103                           itsParentSize.height/2 -
0104                           itsSize.height/2;
0105           this.setLocation( itsLocation);
0106        } // End if.
0107        super.setVisible( yesOrNo);
0108     } // End setVisible.
```

The substantive part of the method, between lines 0092 and 0106, is executed only if the argument is **true**; that is if the request is to post the dialog onto the screen. The purpose of the fragment is to position the dialog so that it is centered within the extent of the component *centerWithin*, which was supplied as an argument to the *MemoryGameNameDialog* constructor. The considerations involved in the calculation are illustrated in Figure 6.17. The calculations compute and set the location of the top left of the dialog, shown as ?x?, ?y? on the diagram, from the other known locations and dimensions.

Once the dialog has been positioned, and outside the scope of the **if** structure, the inherited setVisible() method is called which will actually cause the dialog to be posted or unposted. The two remaining methods retrieve and return the contents of the two JTextFields, as follows. The trim() clause in the **return** expressions will ensure that should the user simply press the space bar without entering any characters an empty string will be returned.

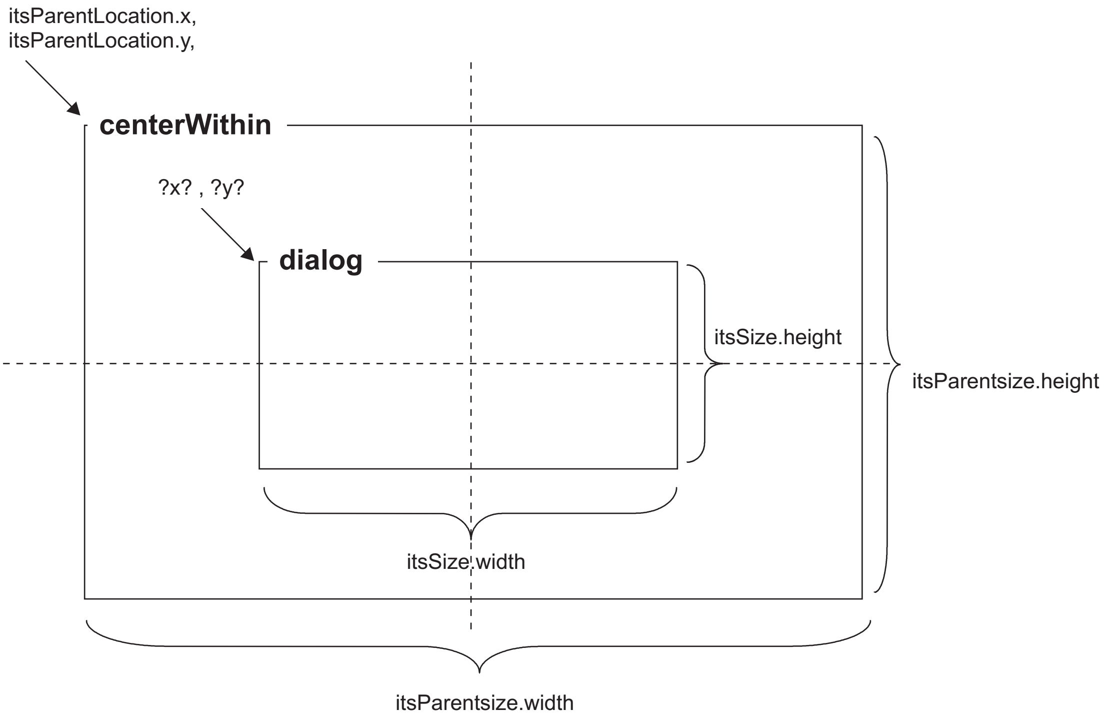


Figure 6.17 *MemoryGameNameDialog*: positioning considerations.

```
protected String getLeftName() {
   return leftField.getText().trim();
} // End getLeftName.

protected String getRightName() {
   return rightField.getText().trim();
} // End getRightName.
```

Within the *actionPerformed()* method of the *MemoryGame* an actionCommand containing *names* will be received when the user activates the *Names* item on the *Options* menu and should result in the dialog being posted. Subsequently, should the user press the *OK* button, an actionCommand containing *namechanged* will be received. These are detected and handled as follows.

```
  } else if ( command.equals( "names")) {
     this.postNameDialog();

  } else if ( command.equals( "namechange")) {
     this.changeNames();
```

```
private void postNameDialog() {
   if ( nameDialog == null) {
      nameDialog = new MemoryGameNameDialog( this, this);
   } // End if.
   nameDialog.setVisible( true);
} // End postNameDialog.
```

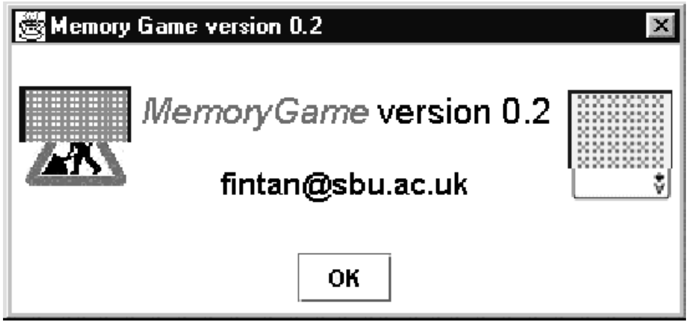
This is an example of lazy dialog creation. The dialog will only be created if the user asks for it, which can cause a delay in the time it takes to appear. One alternative would be to create all dialogs as the application is initialized, but this would impact upon the time taken for the application to appear. A third possibility is to start the application quickly and then create the dialogs on a separate low priority thread of control. (This may give some synchronization problems if the user asks for a dialog which has not yet been created.)

In practice, all three of these methods are used. Dialogs which are not expected to be commonly used are created on demand and may even be destroyed after use. Dialogs which are expected to be heavily used are created on initialization and other dialogs are created on a subsidiary thread.

The *nameDialog* is an instance attribute of the *MemoryGameNameDialog* class and is constructed by this method the first time it is called. The two arguments to the constructor ensure that it is this instance of the *MemoryGame* class to which ActionEvents from the dialog are dispatched and within whose window the dialog is centered as it is posted. The only other step in the method is to post the dialog by calling its *setVisible()* method with the argument **true**. The implementation of the *changeNames()* method is implemented as follows.

```
private void changeNames() {

String  newLeftName  = nameDialog.getLeftName();
String  newRightName = nameDialog.getRightName();
boolean nameChanged  = false;

   if (   newLeftName.length() > 0            &&
        ! newLeftName.equals( leftPlayerName) ){
      leftPlayerName = newLeftName;
      leftDropArea.setName( leftPlayerName);
      nameChanged = true;
   } // End if.

   if (   newRightName.length() > 0            &&
        ! newRightName.equals( rightPlayerName) ){
      rightPlayerName = newRightName;
      rightDropArea.setName( rightPlayerName);
      nameChanged = true;
   } // End if.

   if ( nameChanged) {
      this.doLayout();
   } // End if.
} // End changeNames.
```

The method is implemented by retrieving the *newLeftName* and *newRightName* from the *nameDialog* and, if the String obtained is not empty and if it differs from the existing name, each is then recorded in the corresponding attribute and used as an argument to the appropriate *MemoryGameDropArea setName()* method. The final step in the method,



MemoryGame version 0.2

fintan@sbu.ac.uk

Figure 6.18 *MemoryGameAboutDialog*.

on lines 0303 to 0305, is to lay out the entire interface if either of the names has changed. The effect is that as the dialog disappears the names shown at the top left and right will change, as can be inferred from Figure 6.14.

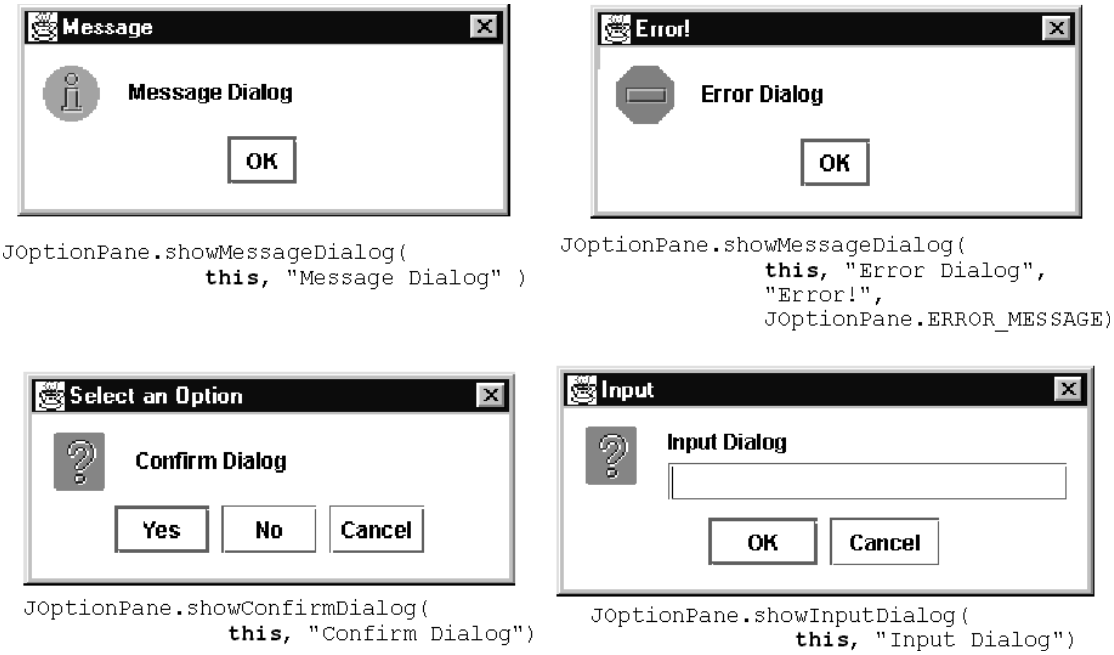
The *About...* option on the *Help* menu also leads to the posting of a specialized dialog. It is expected that this dialog will give information about the producer, developer and version number of the artifact it is describing. Figure 6.18 illustrates the *MemoryGameAboutDialog*, which contains two *TurnCard* instances that are continually turning while the dialog is visible, positioned each side of a JLabel instance displaying HTML text that gives details of the artifact. The design and implementation of this dialog will be left as an end of chapter exercise.

This chapter, and the previous chapter, have introduce the simple modal dialogs that are supplied by the JOptionPane class. As has been explained and demonstrated these dialogs will block the flow of control that they are posted from until the user has responded to and dismissed them. This differs from the specialized dialogs, modal or non-modal, which do not block the current flow of control but rely upon the dispatch of events to inform the main flow of control when the user dismisses, or otherwise interacts with, the dialog. Figure 6.19 illustrates the four most common and useful dialogs which can be obtained from the JOptionPane class. The legend below each of the images shows the Java instruction to post the dialog. The *message* and *error* dialogs do not return any information, the *confirm* dialog will return a manifest integer indicating which button was used to dismiss the dialog and the *input* dialog will return the contents of the input area as a string. More details of the JOptionPane dialogs can be found in the Java documentation.

This chapter has introduced three essential considerations for the provision of an artifact that meets a user's expectations: the implementation of D&D capability, the provision of an application-level main menu and the provision of specialized dialogs. It has upgraded the *MemoryGame* so that it might be expected to meet a user's minimal expectations. However, this has been accomplished by a significant increase in the complexity of the implementation of the artifact and should start to give some indication of why the development of GUIs is such a resource-intensive undertaking.

Summary

- Drag and drop operations have a highly complex software architecture but provide the user with direct manipulation capability which is highly valued.



```
JOptionPane.showMessageDialog(
                   this, "Message Dialog" )
```

```
JOptionPane.showMessageDialog(
                   this, "Error Dialog",
                   "Error!",
                   JOptionPane.ERROR_MESSAGE)
```

```
JOptionPane.showConfirmDialog(
                   this, "Confirm Dialog")
```

```
JOptionPane.showInputDialog(
                   this, "Input Dialog")
```

Figure 6.19 JOptionPane dialogs.

- An application-level main menu has become ubiquitous and conventions have been established for the names, positions and contents of menus.
- A main menu can be supplied as an instance of JMenuBar and installed with the JApplet setJMenuBar() method.
- The main menu is populated with JMenu instances which post a pull-down menu when they are activated and dispatch MenuEvents to a MenuListener as the user interacts with them.
- The JMenu instances can be populated with JMenuItems, JMenuRadioItems, JCheckBoxMenuItems, or cascading menus can be supplied by adding further JMenu instances.
- The labels of JMenuItems which cause a dialog to be posted should be followed by three dots (...).
- The labels of JMenuItems which cascade a further menu will be automatically supplied with a right-facing arrow.
- The *Help* menu should be nominated with the JMenuBar setHelpMenu() method and will always be positioned as the last menu on the right.
- A WindowListener can be registered with a JFrame and will dispatch WindowEvents which can be intercepted to coordinate window operations.
- The JOptionPane class supplies a number of simple modal dialogs.
- Specialized dialogs can be provided by extending the JDialog class and can be modal or non-modal.

Exercises

6.1 Design and implement a simple *TextDropArea* that will only accept objects in the *STRING* flavor and display the string dropped into it within a JTextField.

6.2 The notification of the progress of the D&D operation was implemented in a naïve manner in the chapter. By implementing the appropriate listener methods and using the event inquiry methods the operation can be curtailed if the event source is not capable of supplying an object in an acceptable flavor. Implement this capability and demonstrate it by dragging text from a text component into an upgraded *CardDropArea*.

6.3 Provide an upgraded event/action table for the completed version of the *MemoryGame*.

6.4 Implement different rules for the *MemoryGame* allowing a player who has just turned over a pair of cards to continue playing until they fail to find a pair.

6.5 Revisit the 21 card game from Exercise 5.6 and implement a *PlayingCardPack* object to play the game with. A *PlayingCardPack* instance should contain a shuffled deck of all 52 playing cards and allow the user to drag the top card off the deck and drop it onto a *21DropArea* which would count its score according to the rules of the game.

6.6 Revisit some of the artifacts from previous chapters and exercises and retrofit them with suitable application-level main menu bars.

6.7 Investigate the menu structure of a complex commercial artifact with which you are familiar, such as a word processor, and attempt to draw a map of all the menus, cascading menus and dialogs that it contains. Once you have completed the map shade in all the parts that you commonly use. What does this tell you about the interface complexity of the artifact?

7

Usability Engineering

7.1 Introduction

The term *usability engineering* refers to a body of knowledge, skills and techniques which attempts to maximize a user's ability to comfortably interact with an artifact while minimizing the costs of developing and maintaining it. Many of the associated concepts have already been implicitly introduced throughout the previous chapters. This chapter will consolidate many of these ideas and introduce additional considerations, particularly *usability investigations*. These are the processes whereby a product, either an initial prototype or a finished version, is placed in front of end-users and their interactions with it monitored in some way.

Although the focus of this chapter is on usability engineering it will also, further, consolidate the design and implementation of specialized user interface components. A *TimeInput* component will be developed in this chapter and subsequently used as the example for suggested practical usability investigations. As with the other specialized components already introduced, its engineering will be sufficient for the purposes of this book but not totally complete.

7.2 Know the user, know the task, know the environment!

This is one of the most fundamental slogans of usability engineering. 'Know the user' refers to the different levels of ability and competence which different individuals might have. When designing an interactive component or an artifact's interface the capabilities of the people who will eventually use it should be borne in mind. For example, many of the specialized interactive components presented in this book are intended for *walk-up users*. That is, no specific competences, knowledge or skills are assumed, beyond an initial familiarity with GUIs, and consequently any individual should be able to use them effectively without any explicit preparation or instruction.

More specialized artifacts may be targeted towards particular groups of users. For example, an interface intended to be used by people who do not have fine muscle control must take this into account, or an interface intended to be used by air traffic controllers, who can be assumed to have intensive training before using it for real, will have quite different requirements.

'Know the task' refers to the consideration that an artifact is used to satisfy some requirement in the real world. The nature of the task being performed will determine the design and implementation of the interface. Ideally, the interface should be so task-oriented that users do not actually perceive it and can only see the task that they are performing. For example, an artifact that allows a user to plan the layout of a kitchen by choosing and positioning its different parts (sink units, cupboards, ovens, refrigerators etc.) should be constructed so that the user can concentrate upon the simulated appearance of the kitchen rather than upon the techniques needed to choose and position the parts. A corollary of this phrase is that if the only tool you have is a hammer then everything looks like a nail! An interface is a tool to perform some task, and the characteristics of the tool influence the perception of the task. Ideally, the tool should be so well constructed that it effectively becomes an extension of whoever is wielding it and is adaptive to their developing skill.

'Know the environment' refers to the consideration that a user interacting with an artifact is not isolated from the rest of the world. For example, office productivity applications, word processors, spreadsheets and specialized applications etc., are intended to be used within an office environment. In such an environment interruptions are very common (for example having to answer the phone while in the middle of doing something). The interface should be designed with this consideration in mind and should not require users to have to hold in their human memory any essential information. Instead, the appearance of the interface should clearly indicate the state of the interrupted task, allowing the user to immediately continue after attending to the interruption. Alternatively, the air traffic control example can (hopefully!) assume that controllers will never be distracted from their task and will never have to operate it to the point where their concentration deteriorates.

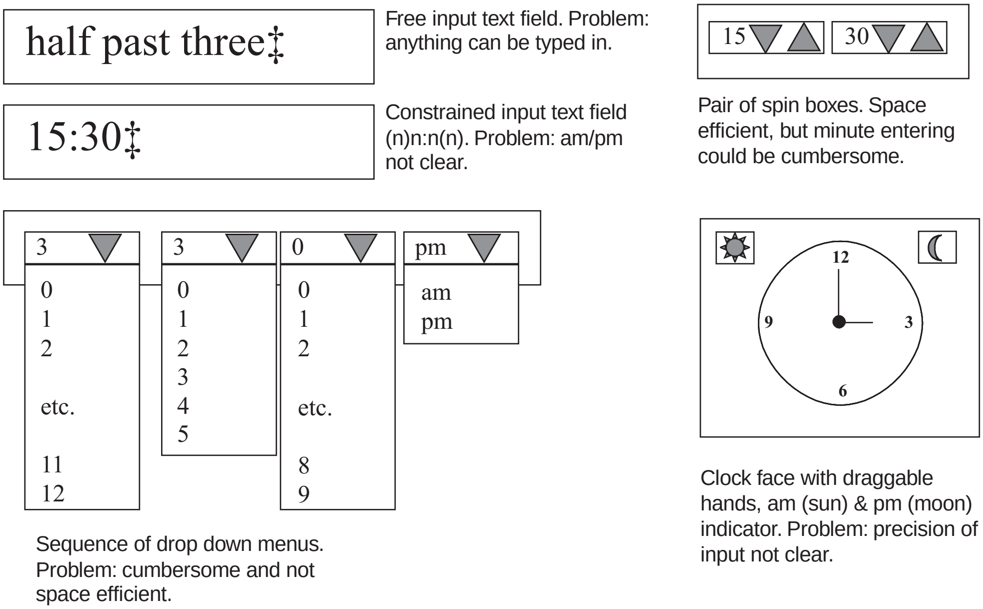
These three fundamental principles are necessary for effective user interface design, yet in themselves they are not sufficient. Many other considerations will also have to be brought to the task of designing a particular artifact's interface. A skilled usability engineer will have a wide repertoire of additional considerations and techniques and will also have the knowledge of which of them are the most applicable in a particular circumstance.

Some of these techniques are proactive, allowing a proposed interface to be evaluated at the initial design or prototype stage. Examples include the use of checklists, heuristics and style guides. Other techniques are reactive and can only be used when the interface has been developed and deployed. Examples include formal experimentation and user satisfaction surveys. Some of these techniques will be introduced throughout this chapter, and more guidance will be given at the end.

One final point that might be worth making at this stage in the chapter is that the engineering requirement to maximize usability while minimizing life cycle costs directly implies that proactive techniques are likely to be most effective. The earlier in the production process that usability considerations are brought to bear, the larger their impact on the finished product and the cheaper their costs. A design change made to an initial STD is far cheaper than having to recall a deployed product that has failed. This in turn implies that the distinction between a usability engineer and a software engineer, at least for those parts of a product that the user will have a direct perception of, is an artificial one. Effective software engineers have, as a matter of definition, to be effective usability engineers and, ideally, vice versa.

7.3 An illustrative task

To illustrate usability engineering the task of entering the time of day, accurate to the nearest minute, into a system via a specialized component will be considered. As this is a common requirement used within a large number of different interfaces it will be specified for walk-up users performing a variety of tasks in a variety of environments. It might



Free input text field. Problem: anything can be typed in.

Constrained input text field (n)n:n(n). Problem: am/pm not clear.

Pair of spin boxes. Space efficient, but minute entering could be cumbersome.

Sequence of drop down menus. Problem: cumbersome and not space efficient.

Clock face with draggable hands, am (sun) & pm (moon) indicator. Problem: precision of input not clear.

Figure 7.1 Time input: initial design sketches.

seem that this very general description of the usability specification simplifies the requirement. However, it is actually one of the most complex to engineer, as no simplifying assumptions can be made about the user, the task or the environment. Figure 7.1 shows five initial interface design sketches that were produced by a student brainstorming this specification.

The first possibility, shown at the top left, is a free input text box. The user would indicate the time by typing it in. The major problem with this idea is that the user might input anything, as illustrated, and then have to have any 'errors' corrected. The interface design principle of preventing errors clearly indicates that this idea is totally unacceptable.

The second possibility, shown below the first, is a constrained text input box similar to the constrained numeric input fields developed in Chapter 4. The comment is suggesting that the user will enter an integer value in the range 0 to 23, a colon (which could be automatically supplied) and then an integer value in the range 0 to 59. One major problem with this idea is that the user will have to be informed of the format required, and this compromises its walk-up capability, even if an instructional message such as '*Enter the time in a twenty-four hour format (e.g. 15:30)*' is supplied. Not only would this require screen space to display, but it can be predicted that many users would not attend to it. (Just consider the number of times someone has pushed at a door that is clearly labeled *pull*!) Although this is an improvement upon the first design it still does not seem optimal.

The third design, shown at the bottom left, contains a sequence of pull-down menus allowing the user to enter the hours value, the number of tens of minutes, the number of minutes, and whether it is morning or evening. This has the advantage of allowing the user not to have to think in twenty-four hour clock format, which is assumed to be more cognitively complex and so more error-prone than the twelve-hour clock format. It can also be operated entirely with the mouse, so not requiring users to relocate their attention from the mouse to the keyboard. However, the design takes up a large amount of screen space and might require the user to perform four different mouse operations to enter a time. As it seems cumbersome and non-intuitive it should not be favored.

The fourth design, shown at the top right, consists of two spin boxes with values constrained to the ranges 0 to 23 and 0 to 59. This has the major advantage of requiring very little screen space, and so would be a very suitable choice for use on personal organizers or mobile phones. If the spin boxes are editable it could also be operated entirely from the keyboard by an experienced user, using the <TAB> key to move between the two boxes. However it requires a twenty-four hour clock format, although a third spin box spinning between the strings *am* and *pm* would solve this at the expense of more screen space. Also, it might require a lengthy mouse interaction to spin the minute box, in the worst case scenario from 0 to 30 (although an inattentive, or naïve, user might spin from 00 to 59 in the worst case). Overall this design was deemed suitable for further consideration.

The final design, shown at the bottom right, consists of a representation of a conventional analog clock face. The user interacts with it by dragging the hands around the face and indicating morning or afternoon by clicking upon the sun or moon icon. Although in its minimum usable size this design will take up more space than the spin box design, it will make better use of the available space if it has to be accorded a larger area. It is also very intuitive, particularly for older users to whom digital clocks may still seem new-fangled, making effective use of the user's everyday experience. However, this version of the design has a major problem with precision, as the provision of sixty tick marks to explicitly indicate the required settings would clutter the interface and increase the minimum usable size.

The two designs on the right are deemed to be effective. Given that the requirements specification does not indicate that the component has to be engineered for use with limited screen space, the final design was chosen as the basis for further development due to its effective use of space and predicted familiarity. In order to overcome the precision problem a text feedback of the precise setting was introduced below the clock face.

7.4 The *TimeInput* component: detailed usability design

The state transition diagram for the *TimeInput* component's behavior is shown in Figure 7.2. It shows that the component has a single state, which indicates that it might be easy to use, although the total of eight transitions, each with preconditions, might suggest a degree of complexity. However, the preconditions are relatively simple and this would also suggest ease of use.

Starting with the transition at the top left, this is followed when a mouse down event occurs and the mouse pointer is outside both the morning and evening (*mevening*) icons. The consequence of following the transition is that one of the hands will be *grabbed*, this will become apparent to the user as the grabbed hand will be shown in a more intense color. If the pointer is within the radius of the smaller, hour, hand it will be grabbed; otherwise the larger, minute, hand will be grabbed. Whichever hand is grabbed it will be moved to point towards the direction of the mouse pointer. As this action has changed the time indicated by the clock face, the time shown at the bottom of the interface will be updated.

The second transition, to the right, is followed when a hand is grabbed and the mouse pointer is dragged. It results in the grabbed hand following the mouse pointer as it moves, and the time shown at the bottom continually updates to show the precise time indicated by the position of the hands. The hands move with a discrete, rather than a continuous

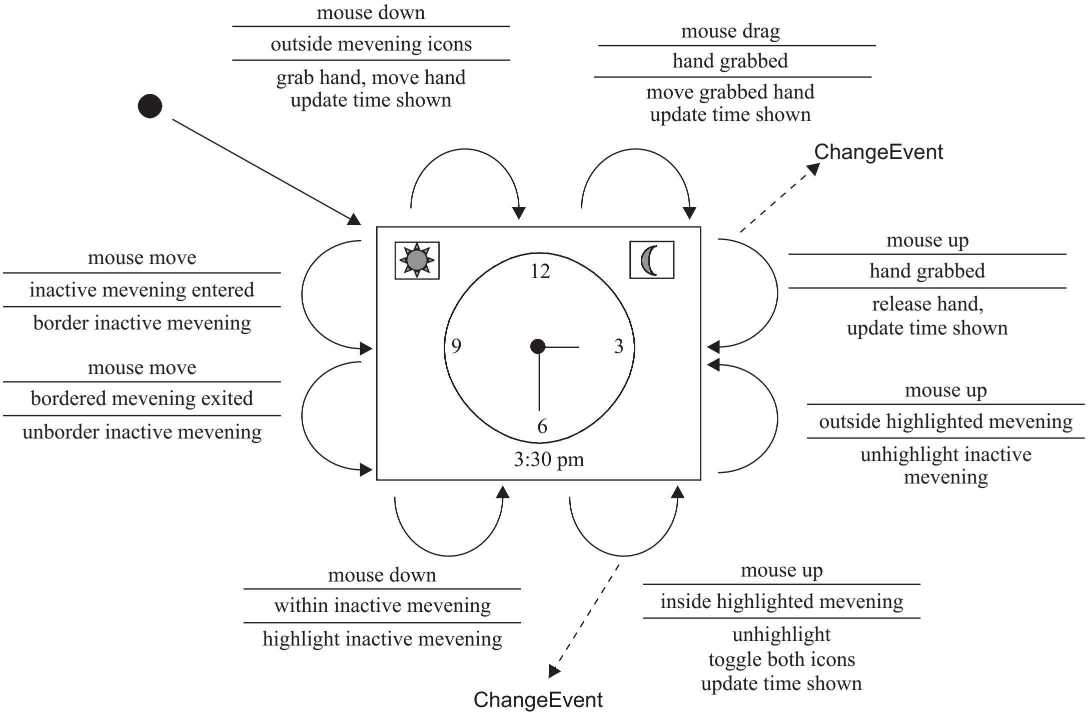


Figure 7.2 *TimeInput*: state transition diagram.

motion; this is most apparent with the hour hand, which jumps from hour to hour and so can only be in one of 12 locations.

The third transition, shown at the top right, is followed when a mouse up event occurs and a hand is grabbed. This results in the grabbed hand being released, the time being updated to ensure that it reflects the final location of the hand and also in a ChangeEvent being dispatched from the component to indicate to its listeners that the user has changed the time.

Taken together these three transitions allow the user to manipulate the location of the clock hands in order to precisely indicate a time within a 12 hour period. It is predicted to be almost immediately apparent to a user that the hands can be manipulated in this way. The interface is a screen metaphor for an analog clock face and, although it is used in the real world to show the time, the user can readily reverse this understanding and use it to indicate a time.

However, the metaphor is not totally in accord with the real world. With a real clock the movement of the hands is usually continual, rather than discrete. Also, when the minute hand passes across the 12 o'clock position with a clockwise rotation, the hour hand will move to point directly to the next hour; and likewise to the previous when it passes with an anticlockwise rotation. This is an example of a metaphor breaking down, which occurs when the metaphorical behavior is different from the user's expectations brought from the real word. This is usually inevitable, as perfect congruence between the behavior of an artifact and the thing in the real world that it is basing its metaphor upon is very difficult, if not impossible, to achieve.

In this example a closer congruence could be achieved by tying the two hands together in some way. That is, the location of the hour hand should not be constrained to 12 discrete locations but should point to intermediate positions, depending upon the location of the minute hand. Likewise, as the hour hand is moved the minute hand should rotate continually, making a complete circle each time the hour hand moves across one twelfth of a circle. However this would complicate the behavior of the clock and make it more difficult to indicate the hour intended. The consequences of breaking this metaphor will be returned to later in the chapter.

One of the most famous user interface metaphors is the wastebasket icon which appeared on the Apple Macintosh desktop. A user could drag resources on to the icon and they would be removed from the system and placed in the wastebasket. They remained in the wastebasket and could be retrieved from it if the user decided that they were still needed. A control on the wastebasket allowed it to be emptied, after which its contents could no longer be retrieved. So far, this description of the wastebasket metaphor is totally in accord with the real-word understanding of an office wastebasket.

However, if the icon representing the floppy disk was dragged onto the wastebasket it caused the disk to eject. The contents of the disk could not be retrieved unless the disk was reinserted into the drive. This behavior is not in accord with that of an office wastebasket. Although the connection between the real-world action of throwing a disk away and removing it from a computer was not maintained it did not seem to cause Macintosh users undue confusion.

The remaining transitions are concerned with the user interacting with the morning and evening (*mevening*) icons to indicate which 12 hour period is intended. Each of these icons has two appearances: active, when it is shown in color, and inactive, when it is shown grayed out. At all times one of the icons will be shown as active and the other as inactive. The user can toggle between the two possibilities by interacting with the inactive icon.

The upper transition shown at the left of the diagram is followed when the mouse pointer moves into the extent of the inactive *mevening* icon and results in the icon being bordered, using the same color as an ungrabbed hand. The transition below it unborders the icon if the pointer leaves it while it is in this state. Taken together these two transitions allow the user to infer that the icon will do something if it is clicked upon and will be easily discovered by users as they explore the interface.

The left-hand transition at the bottom of the diagram is followed when a mouse down event occurs while the pointer is within the inactive *mevening* icon, which must therefore be highlighted. This transition results in the icon being further highlighted by the icon being bordered with the color used for a grabbed hand. This will indicate to the user that the icon has button-like behavior and can be activated by releasing the mouse pointer within the icon. This is shown in the right-hand transition at the bottom of the diagram which, when followed, totally unhighlights the icon, toggles the images shown on both icons, updates the time shown and, as the user has changed the indicated time, dispatches an event. The final transition is followed when the user releases the mouse button while it is outside a highlighted *mevening* icon, and this just cancels the highlight.

Hence the user can only interact with a *mevening* icon while it is in an inactive state, and by using its button-like behavior can change the time indicated from am to pm, or vice versa. It is predictable at this stage in the description that because the *mevening* icons look somewhat like standard buttons some users will attempt to use them in that way, and some may not discover that they are effectively a toggle.

7.5 The *TimeInput* component: class design

The class diagram for the *TimeInput* component is shown in Figure 7.3. The first three encapsulated attributes record the time shown by the component, and each has an associated inquiry method with a fourth inquiry method, *getTimeAsString()*, returning their composite value. The other encapsulated attribute is a list of *listeners* which, with the *addChangeListener()* and *removeChangeListener()* methods, identifies instances of this class as *ChangeEvent* sources.

The default constructor will initialize an instance to show the current system time. The alternative constructor initializes the instance to show the time specified in its arguments, using a 24 hour *hoursNow* argument to include morning or afternoon. The reasons why this class needs an *addNotify()* method will be explained below. The *paint()* method renders the component as illustrated in Figure 7.4.

The two protected methods, *processMouseEvent()* and *processMouseMotionEvent()* indicate that the component will take responsibility for the internal handling of mouse events.

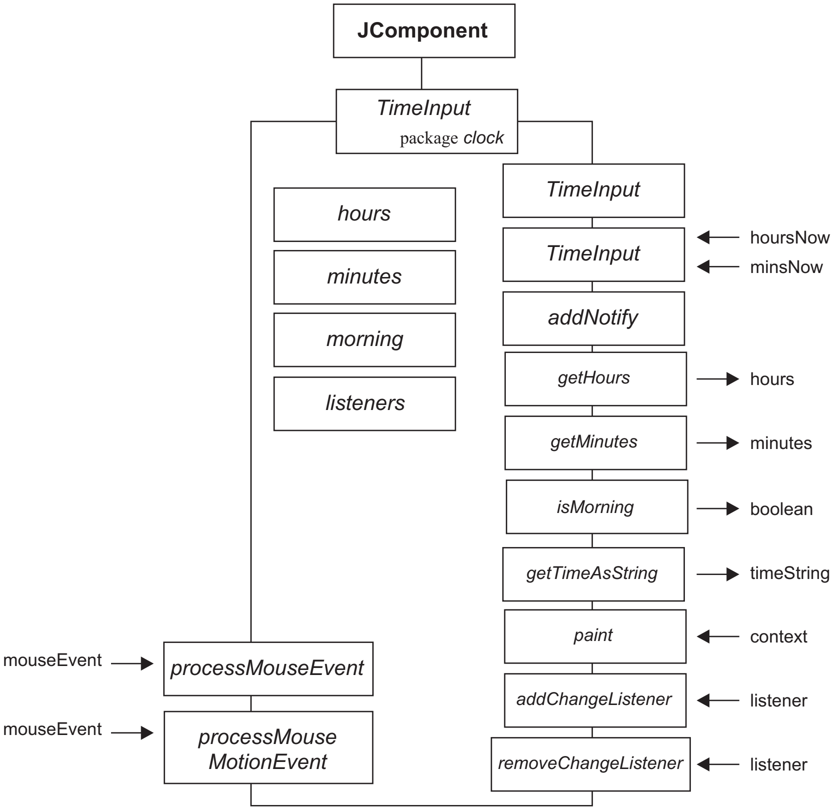


Figure 7.3 *TimeInput*: class diagram.

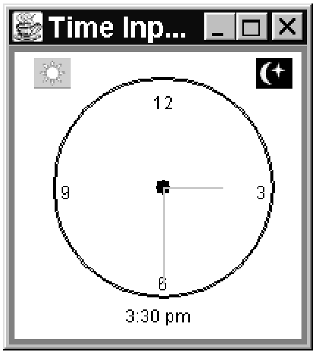


Figure 7.4 *TimeInput*: appearance.

7.6 The *TimeInput* component: implementation

The implementation of the *TimeInput* class, as far as the end of its class-wide constants, is as follows.

```
0011  package clock;
0012
0013  import java.awt.*;
0014  import java.awt.event.*;
0015  import java.applet.*;
0016  import java.util.*;
0017  import javax.swing.*;
0018  import javax.swing.border.*;
0019  import javax.swing.event.*;
0020
0021
0022  public class TimeInput extends JComponent {
0023
0024  private static final int CLOCK_WIDTH      = 180;
0025  private static final int CLOCK_HEIGHT     = 180;
0026  private static final int BIG_HAND_LENF    = 65;
0027  private static final int SMALL_HAND_LENF  = 40;
0028  private static final int MEVENING_OFFSET  = 2;
```

The first four constant declarations, on lines 0024 to 0027, define the size of the clock face and the lengths of the two hands that will be shown upon it. The last declaration, *MEVENING_OFFSET*, defines the size of the gap between the edge of the clock face and the morning and evening icons, as can be seen on Figure 7.4. The class definition continues with the instance attribute declarations, as follows.

```
0030  private int     hours    = 0;
0031  private int     mins     = 0;
0032  private boolean morning  = true;
0033
0034  private Image clockBackground = null;
0035  private Image activeMorning   = null;
0036  private Image inactiveMorning = null;
0037  private Image activeEvening   = null;
```

```
0038  private Image inactiveEvening = null;
0039
0040  private Rectangle clockRect   = null;
0041  private Rectangle morningRect = null;
0042  private Rectangle eveningRect = null;
0043
0044  private Color inactiveHandColor = Color.pink;
0045  private Color activeHandColor   = Color.red;
0046
0047  private boolean grabbed         = false;
0048  private boolean bigHandGrabbed  = false;
0049  private boolean selecting       = false;
0050
0051  private boolean morningEntered  = false;
0052  private boolean eveningEntered  = false;
0053  private boolean morningPrimed   = false;
0054  private boolean eveningPrimed   = false;
0055
0056  private EventListenerList listenerList = null;
```

The first three of these declarations are the first three attributes shown on the class diagram. The next five are the Images that are needed to display the state of the clock. These are the background image containing the circle and the numerals, and then two images for each of the morning and evening icons. The three Rectangle attributes which follow, on lines 0040 to 0042, record the area within the extent of the component where the three Images have been placed.

The two Color declarations, on lines 0044 and 0045, are used to indicate the grabbed hand and also for the highlighting of the icon. The choice of pink and red is arbitrary and a more robustly engineered component would have methods to support the attributes. The three **boolean** attributes, on lines 0047 to 0049, will be used to indicate the state of the user's interaction: if they have currently *grabbed* one of the hands and, if so, if it is the *bigHandGrabbed* or if they are *selecting* morning or evening. The four flags, which follow, record the situation when an icon is being selected.

The final attribute, *listenerList*, declared on line 0056, will be used to contain the list of ChangeListeners for this component. The ChangeListener class is a part of the JFC and the techniques used to maintain the list of listeners and to dispatch events to them differ from those used for AWT event classes, such as ActionEvents, which have already been described.

The implementation of the two constructors is as follows.

```
0059     public TimeInput() {
0060        this((new GregorianCalendar()).get(Calendar.HOUR_OF_DAY),
0061             (new GregorianCalendar()).get(Calendar.MINUTE));
0062     } // End TimeInput default constructor.
0063
0064
0065     public TimeInput( int hoursNow, int minsNow) {
0066        super();
0067        if ( hoursNow < 0 || hoursNow > 23) {
0068             hoursNow = 0;
0069        } // End if.
0070         if ( minsNow < 0 || minsNow > 59) {
0071             minsNow = 0;
0072         } // End if.
0073
0074         hours = hoursNow;
0075         mins  = minsNow;
0076
0077         if ( hours >= 12) {
0078            hours  -= 12;
0079            morning = false;
0080         } else {
0081            morning = true;
0082         } // End if.
0083
0084         this.setOpaque( true);
0085         listenerList = new EventListenerList();
0086         this.enableEvents( AWTEvent.MOUSE_EVENT_MASK        |
0087                            AWTEvent.MOUSE_MOTION_EVENT_MASK);
0088      } // End TimeInput constructor.
```

The default constructor indirects to the alternative constructor, passing as arguments the hours and minutes obtained from the system clock by means of GregorianCalendar instances. The alternative constructor commences by calling its **super**, JComponent, constructor and continues on lines 0067 to 0072 with checks to ensure that the values of the arguments are acceptable, correcting them to default values if they are not.

Having validated the arguments the instance attributes *hours* and *mins* are initialized from the arguments on lines 0074 and 0075. The **if/else** structure on lines 0077 to 0082 then constrains the value of hours to the range 0 to 12 and sets the value of the third attribute *morning* as appropriate.

The final steps in the constructor, on lines 0084 to 0087, ensure that the component is opaque, construct the *listenerList* and enable both categories of mouse events upon itself. The construction of the JFC EventListenerList, on line 0085, differs from the AWT equivalent that would have left the list with a **null** value to indicate that it is empty.

The four inquiry methods are implemented, without comment, as follows. An example of the String returned from the *getTimeAsString()* method can be seen below the clock face in Figure 7.4.

```
0149      public int getHours() {
0150         return hours;
0151      } // End getHours.
0152
0153      public int getMinutes() {
0154         return mins;
0155      } // End getHours.
0156
0157      public boolean isMorning() {
0158         return morning;
0159      } // End isMorning.
0160
0161      public String getTimeAsString() {
0162
String theTime = hours + ":" + mins;

   if ( morning) {
      return theTime + " am";
   } else {
      if ( hours == 0) {
         return "12:" + mins + " pm";
      } else {
         return theTime + " pm";
      } // End if.
   } // End if.
} // End getTimeAsString.
```

The *addNotify()* method is responsible for initializing the remaining instance attributes and commences as follows.

```
public void addNotify() {

Dimension    preferred = null;
Insets       insets    = this.getInsets();
MediaTracker tracker   = new MediaTracker( this);
Class        classDescriptor = this.getClass();
URL          resourceLocation = null;

   resourceLocation = classDescriptor.
                               getResource( "ama.gif");
   activeMorning   = Toolkit.getDefaultToolkit().
                           getImage( resourceLocation);

   resourceLocation = classDescriptor.
                               getResource( "ami.gif");
   inactiveMorning = Toolkit.getDefaultToolkit().
                           getImage( resourceLocation);

   resourceLocation = classDescriptor.
                               getResource( "pma.gif");
   activeEvening   = Toolkit.getDefaultToolkit().
                           getImage( resourceLocation);

   resourceLocation = classDescriptor.
                               getResource( "pmi.gif");
   inactiveMorning = Toolkit.getDefaultToolkit().
                           getImage( resourceLocation);

   tracker.addImage( activeMorning,   0);
   tracker.addImage( inactiveMorning, 1);
   tracker.addImage( activeEvening,   2);
   tracker.addImage( inactiveEvening, 3);

   try {
      tracker.waitForAll();
   } catch ( InterruptedException exception) {
     // Do nothing.
0127        } // End try/catch.
0128
0129        if ( tracker.isErrorAny()) {
0130            System.err.println( "TimeInput inages not loaded");
0131            System.exit( -1);
0132        } // End if.
```

The first part of the method, from lines 0098 to 0116, initiates the loading of the four images which will be used for the morning and evening icons. The technique used to do this differs from that used to load the *MemoryGame* images, as described in previous chapters. The *MemoryGame* images were application specific, whereas these images are intimately bound to the class. Every *TimeInput* instance will use these same four images, whereas a *MemoryGame* instance may make use of any suitable set of images.

The process commences, on line 0095, by obtaining the *classDescriptor*, an instance of the Class class, for the TimeInput class. An instance of the Class class contains a complete description of the class, including where it was obtained from. This knowledge is made use of, on lines 0098 and 0099, to obtain a *U*niform *R*esource *L*ocator, called *resourceLocation*, for the `ama.gif` image, which must be stored in the same location as the *TimeInput.class* file. A URL is effectively the Internet address of a resource, and it is this address which is passed to the Toolkit getImage() method on lines 0100 and 0101. The use of the URL class will be more fully explained in the next chapter; all that needs to be noted here is that this will initiate the asynchronous loading of the `ama.gif` image, in the same way as the technique used in the *MemoryGame* to load its images asynchronously.

Lines 0103 to 0116 initiate the loading of the other three images and the remainder of the method synchronizes the threads, using a MediaTracker instance as has previously been explained. The *addNotify()* method continues as follows.

```
0134        morningRect        = new Rectangle();
0135        morningRect.width  = activeMorning.getWidth(  this);
0136        morningRect.height = activeMorning.getHeight( this);
0137
0138        eveningRect        = new Rectangle();
0139        eveningRect.width  = morningRect.width;
0140        eveningRect.height = morningRect.height;
0141
0142        clockRect          = new Rectangle();
0143        clockRect.width    = CLOCK_WIDTH;
0144        clockRect.height   = CLOCK_HEIGHT;
0145
0146        preferred        = new Dimension();
0147        preferred.width  = CLOCK_WIDTH +
0148                           insets.left + insets.right;
0149        preferred.height = CLOCK_HEIGHT +
0150                           insets.top + insets.bottom;
0151                           (this.getFontMetrics(
0152                                 this.getFont())).getHeight();
0153        this.setMinimumSize(   preferred);
0154        this.setPreferredSize( preferred);
0155        this.setMaximumSize(   preferred);
0156
0157        super.addNotify();
```

```
0158     } // End addNotify.
```

The *morningRect*, *eveningRect* and *clockRect* attributes are all instances of the Rectangle class. At this stage, on lines 0121 to 0131, they can be constructed and the width and height attributes of all three of them can be initialized. The other two Rectangle attributes, x and y, can only be initialized as the interface is laid out in the *paint()* method, when the precise location of the images within the component is known.

The last substantive step of the *addNotify()* method, on lines 0133 to 0142, is to establish a preferred, minimum and maximum size for the component to take part in layout negotiations. All three of these attributes are set to the same value, which is established, on lines 0134 to 0139, in the local Dimension variable, *preferred*. The width is decided from the width of the clock face plus the left and right border *insets*. The height is decided from the height of the clock face, the top and bottom border *insets* and also the height required for the text representation of the time. It is this latter part of the component's height which can only be known at this stage, causing an *addNotify()* method to be required. These considerations are illustrated in Figure 7.5. The very final step of the overriding *addNotify()* method, on line 0144, is to call its **super**, JComponent, addNotify() method. A failure to include the chaining call of the **super** addNotify() method is a common production bug which results in the component never becoming visible.

These layout considerations are also essential to the implementation of the *paint()* method, which commences as follows.

```
0209     public void paint( Graphics context) {
0210
0211     int bigHandAngle    = 0;
0212     int smallHandAngle  = 0;
0213     int windowWidth     = this.getWidth();
0214     int windowHeight    = this.getHeight();
```

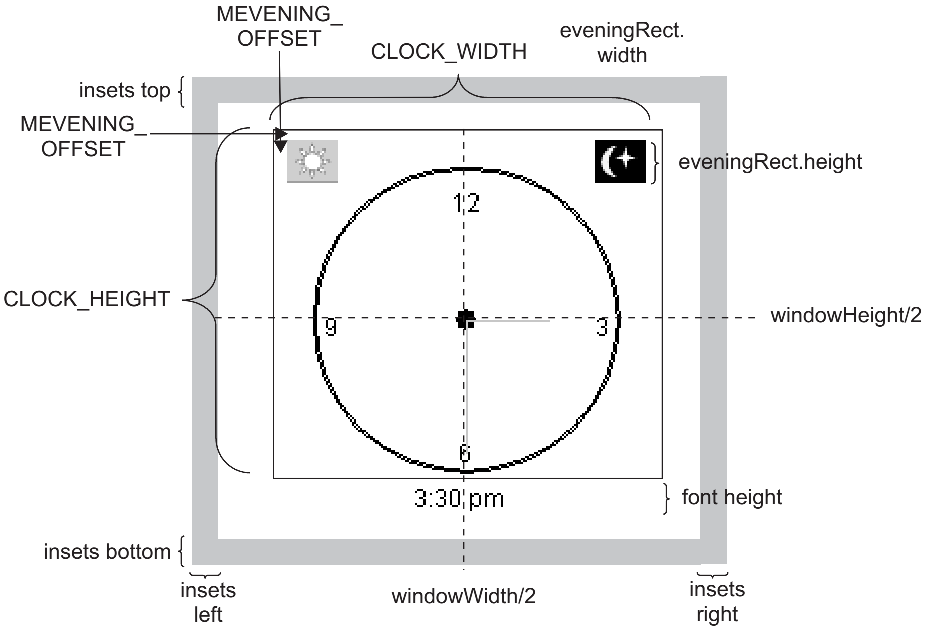


Figure 7.5 *TimeInput*: layout considerations.

```
0215     Insets      insets  = this.getInsets();
0216     FontMetrics metrics = this.getFontMetrics(
0217                                  this.getFont());
0218     String      theTime = this.getTimeAsString();
0219
0220        if ( clockBackground == null) {
0221           this.prepareBackgroundImage();
0222        } // End if.
```

The method commences on line 0220 with a check to ensure that the *clockBackground* image has been prepared and indirects to *prepareBackgroundImage()* if not. This method constructs the image which contains those parts of the clock face which never change, the circle, the boss in the middle of the circle and the numerals around the outside. The method is implemented as follows.

```
0173     private void prepareBackgroundImage() {
0174
0175     Graphics    clockGraphics = null;
0176     FontMetrics metrics       = this.getFontMetrics(
0177                                         this.getFont());
0178
0179        clockBackground = this.createImage(
0180                                CLOCK_WIDTH, CLOCK_HEIGHT);
0181        clockGraphics   = clockBackground.getGraphics();
0182        clockGraphics.setColor( this.getBackground());
0183        clockGraphics.fillRect( 0, 0,
0184                                CLOCK_WIDTH, CLOCK_HEIGHT);
0185        clockGraphics.setColor( this.getForeground());
0186        clockGraphics.drawOval( 15, 15,
0187                                 CLOCK_WIDTH  -30,
0188                                 CLOCK_HEIGHT -30);
0189        clockGraphics.drawOval( 16, 16,
0190                                 CLOCK_WIDTH  -32,
0191                                 CLOCK_HEIGHT -32);
0192
0193        clockGraphics.fillOval( (CLOCK_WIDTH/2)  -5,
0194                                 (CLOCK_HEIGHT/2) -5,
0195                                 10, 10);
0196
0197        clockGraphics.drawString( "12",
0198                    (CLOCK_WIDTH -
0199                       metrics.stringWidth( "12"))/2,
0200                            20 + metrics.getHeight());
0201        clockGraphics.drawString( "3",
0202                     CLOCK_WIDTH -
0203                        metrics.stringWidth( "3") -20,
0204                        (CLOCK_HEIGHT +
0205                              metrics.getHeight())/2);
0206        clockGraphics.drawString( "6",
0207                     (CLOCK_WIDTH -
0208                         metrics.stringWidth( "6"))/2,
0209                                    CLOCK_HEIGHT - 20);
0210        clockGraphics.drawString( "9",
0211                                  20,
0212                           (CLOCK_HEIGHT +
0213                                 metrics.getHeight()) /2);
0214     } // End prepareBackgroundImage.
```

The first step of this method, on lines 0179 and 0180, is to create the Image instance, *backgroundImage*, using the Component createImage() method. The arguments to this method are the required width and height of the image. Having constructed the image, its getGraphics() method can be called to obtain a Graphics instance, called *clockGraphics*, that can be used to draw onto the image.

Lines 0182 to 0185 then ensure that the entire extent of the *clockBackground* Image is cleared to the current background color and that the *clockGraphics* is restored to draw in the current foreground color. Lines 0186 to 0191 then draw two concentric circles, using drawOval(), within the extent of the image to provide the outer circle of the clock face. The central boss is then produced, by drawing a filled oval, on lines 0193 to 0195. The remaining parts of the method render the numerals onto the image, at appropriate locations, using the *clockGraphics* drawString() method.

This is another part of the component's construction which would require re-engineering to bring it up to production quality. The positioning of the numerals is a little haphazard and could easily be broken by the specification of a larger font. A change of the background color should result in the background, and possibly the foreground, color of the clock face changing. A resizing of the component should result in a resizing of the clock face to make best use of the available space, although there is a minimium size below which the component would be unusable.

Having ensured that the *clockBackground* image has been prepared the *paint()* method can present the state of the component to the user. This will take place in a number of stages, as illustrated in Figure 7.6.

The first of these stages is to clear the entire window and draw the border, which is accomplished, without comment, as follows.

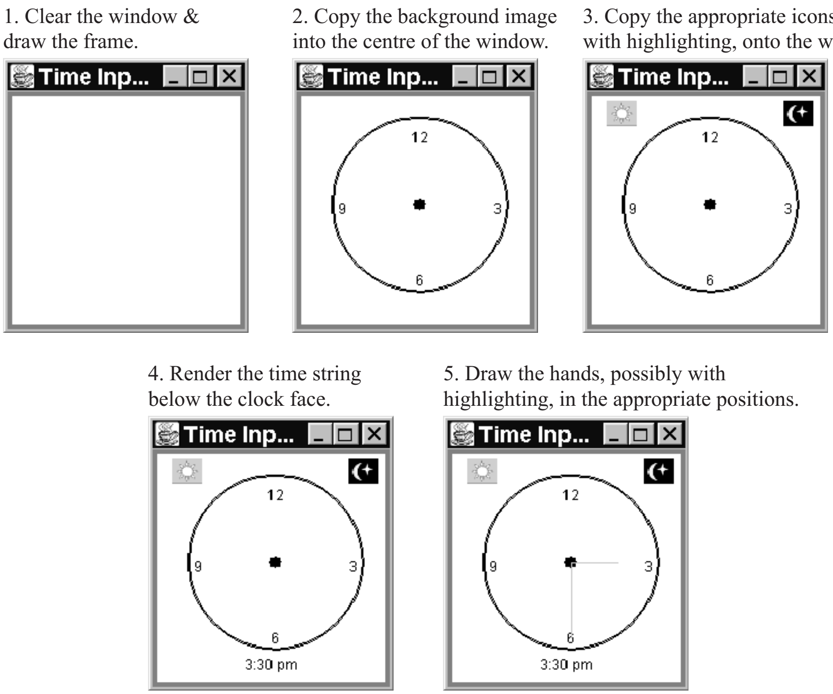
```
0224        context.setColor( this.getBackground());
0225        context.fillRect( 0, 0,
0226                          windowWidth, windowHeight);
0227        this.paintBorder( context);
```

The next stage is to draw the *clockBackground* Image in the center of the window. The *x* and *y* location of the upper left-hand corner of the position of the image is also recorded in the x and y attributes of the *clockRect* Rectangle attribute. The other two attributes of this object, the width and height, were initialized in the *addNotify()* method, as they will never change.

```
0237        clockRect.x =  (windowWidth - CLOCK_WIDTH)  /2;
0238        clockRect.y =  (windowHeight- CLOCK_HEIGHT) /2;
0239        context.drawImage( clockBackground,
0240                           clockRect.x, clockRect.y, this);
```

The third stage involves the positioning and possible highlighting of the two icons. The positioning and drawing of the icons is accomplished as follows.

```
0243        morningRect.x  = clockRect.x + MEVENING_OFFSET;
0244        morningRect.y  = clockRect.y + MEVENING_OFFSET;
```


1. Clear the window & draw the frame.

2. Copy the background image into the centre of the window.

3. Copy the appropriate icons, possibly with highlighting, onto the window.

4. Render the time string below the clock face.

5. Draw the hands, possibly with highlighting, in the appropriate positions.

Figure 7.6 *TextInput*: *paint()* details.

```
0245
0246        eveningRect.x  = clockRect.x +
0247                              CLOCK_WIDTH - MEVENING_OFFSET
0248                                        - morningRect.width;
0249        eveningRect.y  = clockRect.y + MEVENING_OFFSET;
0250
0251        if ( morning) {
0252           context.drawImage( activeMorning, morningRect.x,
0253                                        morningRect.y, this);
0254           context.drawImage( inactiveEvening, eveningRect.x,
0255                                        eveningRect.y, this);
0256        } else {
0257           context.drawImage( inactiveMorning,  morningRect.x,
0258                                        morningRect.y, this);
0259           context.drawImage( activeEvening, eveningRect.x,
0260                                        eveningRect.y, this);
0261        } // End if.
```

The first step in this stage, on lines 0243 to 0249, is to compute the upper left positions of both icons relative to the upper left location of the clock face which has just been stored in *clockRect*, and store the details in the *morningRect* and *eveningRect* Rectangle attributes.

Having done this, the selection structure, on lines 0251 to 0261, can draw the appropriate pair of images in the pre-computed locations. The **boolean** attribute *morning* will be **true** for an am time and **false** for a pm time, and its value will determine which pair of icons is used. This stage of the *paint()* method continues, as follows, with the possible highlighting of one of the icons.

```
0263        if ( morning) {
0264           if ( eveningPrimed) {
0265              context.setColor( activeHandColor);
0266              context.drawRect( eveningRect.x, eveningRect.y,
0267                        eveningRect.width, eveningRect.height);
0268           } else if ( eveningEntered) {
0269              context.setColor( inactiveHandColor);
0270              context.drawRect( eveningRect.x, eveningRect.y,
0271                        eveningRect.width, eveningRect.height);
0272           } // End if.
0273
0274        } else {
0275          if ( morningPrimed) {
0276              context.setColor( activeHandColor);
0277              context.drawRect( morningRect.x, morningRect.y,
0278                        morningRect.width, morningRect.height);
0279          } else if ( morningEntered) {
0280              context.setColor( inactiveHandColor);
0281              context.drawRect( morningRect.x, morningRect.y,
0282                        morningRect.width, morningRect.height);
0283          } // End if.
0284        }// End if.
```

If the *morning* attribute is **true** then only the evening icon could be active, and vice versa; accordingly this fragment contains a two-way selection. The four attributes *eveningPrimed*, *eveningEntered*, *morningPrimed* and *morningEntered* are maintained by the event-handling routines to indicate the state of the user's interaction with the icons. It is possible that none of these attributes will be **true**, in which case no highlighting will take place. Otherwise a highlight using the *activeHandColor* will be drawn around the appropriate icon if it is primed, or a highlight using the *inactiveHandColor* if it has only been entered. The extent of the highlight is determined by the composite value of the *morningRect* or *eveningRect* attribute.

The fourth stage shown in Figure 7.6 is to render the time string immediately below the clock face, and is accomplished as follows.

```
0286        context.setColor( this.getForeground());
0287        context.drawString( theTime,
0288               (windowWidth - metrics.stringWidth( theTime)) /2,
0289                            clockRect.y + clockRect.height );
```

The value in the local String variable *theTime* has been initialized from a call of *getTimeAsString()* when it was declared on line 0218. The final stage in the *paint()* method is to draw the hands, and is accomplished as follows.

```
0291        context.setColor( inactiveHandColor);
0292        context.translate( clockRect.x + CLOCK_WIDTH/2,
0293                           clockRect.y + CLOCK_HEIGHT/2);
0294
0295        smallHandAngle = ((hours +21)  %12) * 30;
0296        bigHandAngle   = ((mins  +105) %60) * 6;
0297
0298        if ( grabbed && bigHandGrabbed) {
0299           this.paintHand( context,
0300                           smallHandAngle, SMALL_HAND_LENF);
0301           context.setColor( activeHandColor);
0302           this.paintHand( context,
0303                           bigHandAngle, BIG_HAND_LENF);
0304        } else if ( grabbed && smallHandGrabbed) {
0305           this.paintHand( context,
0306                           bigHandAngle, BIG_HAND_LENF);
0307           context.setColor( activeHandColor);
0308           this.paintHand( context,
0309                           smallHandAngle, SMALL_HAND_LENF);
0310        } else {
0311           this.paintHand( context,
0312                           bigHandAngle,   BIG_HAND_LENF);
0313           this.paintHand( context,
0314                           smallHandAngle, SMALL_HAND_LENF);
0315        } // End if.
0316     } // End paint.
```

By default the *origin*, i.e the (0,0) location, of a Graphics instance is the upper left of the drawable area; with x values increasing to the right, y values increasing downwards and the zero degree angle pointing to 3 o'clock on the clock face. Together this set of conventions and information is known as a *frame of reference*.

For convenience lines 0292 and 0293 use the Graphics translate() method to move the origin to the center of the clock face. The two instance attributes *hours* and *mins* contain the time to be shown, in the ranges 0 to 11 and 0 to 59 respectively. On lines 0295 to 0296 the two local variables *smallHandAngle* and *bigHandAngle* are initialized from these attributes to represent the direction that the clock hands should be drawn in in the Graphics frame of reference.

There are three possibilities for the drawing of the hands. Either the big hand has been grabbed and should be drawn in the active color, or the small hand has been grabbed and should be drawn in the active color, or neither of the hands has been grabbed and both hands should be drawn in the inactive color. The outer three-way selection structure on lines 0298 and 0315 implements this decision and its consequences, each making two calls of a private method called *paintHand(),* implemented as follows.

```
0321     private void paintHand( Graphics context,
0322                             int angle, int lenf) {
0323
0324     double cosAngle = Math.cos( Math.toRadians( angle));
0325     double sinAngle = Math.sin( Math.toRadians( angle));
0326
0327     int deltaX  = (int) ((lenf * cosAngle) );
0328     int deltaY  = (int) ((lenf * sinAngle));
0329
0330         context.drawLine( 0, 0, deltaX, deltaY);
0331     } // End paintHand.
```

The three arguments to this method are the Graphics *context*, which has already had its origin translated and the color to draw in installed, the *angle* in degrees to draw the hand and the length (*lenf*) of the hand to draw. Lines 0324 to 0328 compute the *deltaX* and *deltaY* displacements of the end of the hand, having first to convert the angle from degrees to radians. Having initialized these variables the method's single step, on line 0330, is to draw the hand from the translated origin (0,0) to the displacements just calculated.

This completes the *paint()* method, which, although it is complex, will be called repeatedly as the user interacts with the component. One important side-effect of the method is to maintain in the attributes *morningRect* and *eveningRect* the locations of the two icons. The significance of this side-effect will become clear when the two event-handling routines, *processMouseEvent()* and *processMouseMotionEvent()* are described. The *processMouseMotionEvent()* method is slightly simpler and commences as follows.

```
0407     protected void processMouseMotionEvent( MouseEvent event) {
0408
0409     int     mouseX        = event.getX();
0410     int     mouseY        = event.getY();
0411     int     dX            = 0;
0412     int     dY            = 0;
0413     int     angleOfMouse  = 0;
0414     boolean repaintNeeded = false;
0415
0416        if ( grabbed                                  &&
0417             event.getID() == MouseEvent.MOUSE_DRAGGED ){
0418
0419           dX = mouseX - ( clockRect.x + (CLOCK_WIDTH/2));
0420           dY = ( clockRect.y + (CLOCK_HEIGHT/2)) - mouseY;
0421           angleOfMouse = (((int) Math.toDegrees(
0422                             Math.atan2( dX, dY))) + 270) % 360;
0423
0424           if ( bigHandGrabbed) {
0425             mins = ((angleOfMouse / 6) + 15) % 60;
0426           } else {
0427             hours = ((angleOfMouse / 30) +3) % 12;
0428           } // End if.
0429           repaintNeeded = true;
```

This part of the method is only activated, as shown on lines 0416 and 0417, if one of the clock hands has already been *grabbed* and the event indicates that the user is dragging the mouse pointer. The first step, on lines 0419 to 0422, is to compute the angle of the mouse pointer, in *angleOfMouse*, relative to the center of the clock face. This involves some complex trigonometrical considerations which will not be described in detail. The next step, on lines 0424 to 0428, updates either the *mins* or the *hours* attribute, depending upon which hand has been grabbed, from the *angleOfMouse* value just computed. The final term on lines 0425 and 0427, % 12 or % 60, will constrain the position of the hands to indicate one of 12 hours or one of 60 minutes, as appropriate. As the user has dragged one of the clock hands to a new location the *repaintNeeded* flag is set to cause the component to redisplay itself. The method continues as follows.

```
0431        } else {
0432           if ( !morning &&
0433                 morningRect.contains( mouseX, mouseY)) {
0434              if ( !morningEntered) {
0435                 morningEntered = true;
0436                 repaintNeeded  = true;
0437              } // End if.
0438           } else if ( morningEntered) {
0439              morningEntered = false;
0440              repaintNeeded  = true;
0441           } // End if.
0442
0443           if ( morning &&
0444                eveningRect.contains( mouseX, mouseY)) {
0445              if ( !eveningEntered) {
0446                 eveningEntered = true;
0447                 repaintNeeded  = true;
0448              } // End if.
0449           } else if( eveningEntered) {
0450              eveningEntered = false;
0451              repaintNeeded  = true;
0452           } // End if.
0453        } // End if.
```

This part of the method is concerned with the mouse pointer entering or leaving one of the icons. At any instance only one of the icons, determined by the value of *morning*, can be active, and so is implemented as a sequential two-way selection with comparable branches.

The first branch on lines 0432 to 0441 is concerned with the situation where the morning icon is active (*!morning*). If the mouse pointer is within the extent delineated by the *morningRect* attribute and the *morningEntered* attribute does not already indicate this, lines 0435 and 0436 effect the transition of entering the icon. This is accomplished by setting the *morningEntered* attribute and also the *repaintNeeded* flag so that the state change will subsequently become apparent to the user by an *inactiveColor* highlight around the icon. Otherwise, on lines 0438 to 0441, the pointer is not within the extent of the *morningRect* attribute and if the value of *morningEntered* indicates that it was, the value of *morningEntered* is cleared and the flag set to repaint.

Lines 0443 to 0453 are essentially identical and are concerned with the mouse pointer entering and leaving the extent of the *eveningRect*. The method concludes, as follows, by scheduling a repaint() if the *repaintNeeded* flag indicates that one is required and also invoking the **super**, JComponent, processMouseMotionEvent() handler.

```
0455        if ( repaintNeeded) {
0456           this.repaint();
0457        } // End if.
0458
0459        super.processMouseMotionEvent( event);
0460     } // End processMouseMotionEvent.
```

The other event-handling method, *processMouseEvent()*, is concerned with responding to the user's interactions with the mouse button, and commences as follows.

```
0334     protected void processMouseEvent( MouseEvent event) {
0335
0336     int mouseX = event.getX();
0337     int mouseY = event.getY();
0338
0339     int dX                 = 0;
```

```
    int dY                 = 0;
    int distanceFromCenter = 0;
    int angleOfMouse       = 0;

       switch ( event.getID()) {

       case MouseEvent.MOUSE_PRESSED:

          if ( morningRect.contains( mouseX, mouseY)) {
             if ( !morning) {
                morningPrimed = true;
             } // End if.

          } else if ( eveningRect.contains( mouseX, mouseY)) {
             if ( morning) {
                eveningPrimed = true;
             } // End if.
```

This part of the method is concerned with responding to mouse down events that occur within the extent of an active icon and result in the value of *morningPrimed* or *eveningPrimed* being set **true**. The consequence of this is that the icon will subsequently be shown to the user with an *activeColor* highlight. This part of the method continues with responding to a mouse down event outside the extent of an active icon and results in one of the hands being grabbed.

```
          } else {

             grabbed = true;
             dX = Math.abs(( clockRect.x + (CLOCK_WIDTH/2))
                                                  - mouseX);
             dY = Math.abs(( clockRect.y + (CLOCK_HEIGHT/2))
                                                   - mouseY);
             distanceFromCenter = (int) (
                           Math.sqrt((dX * dX) + (dY * dY)));

             dX = mouseX - ( clockRect.x + (CLOCK_WIDTH/2));
             dY = ( clockRect.y + (CLOCK_HEIGHT/2)) - mouseY;
             angleOfMouse = (((int) Math.toDegrees(
                               Math.atan2( dX, dY)))
                                       + 270) % 360;

             if ( distanceFromCenter < SMALL_HAND_LENF) {
                bigHandGrabbed = false;
                hours = ((angleOfMouse / 30) +3) % 12
             } else {
                bigHandGrabbed = true;
                mins = ((angleOfMouse / 6) + 15) % 60;
             } // End if.
          } // End if.
          break;
```

This fragment commences, after setting the *grabbed* flag, by computing the *distanceFromCentre*, using Pythagoras, and *angleOfMouse* using trigonometry as before.

The value of *distanceFromCentre* is then used to decide whether the big hand or the little hand has been grabbed and the value of *hours* or *mins* updated as appropriate. The consequence of this is that one of the hands will jump to the location indicated by the mouse and be shown in the *activeColor*. The final term on lines 0377 and 0380, % 12 or % 60, will constrain the position of the hands to indicate one of 12 hours or one of 60 minutes, as appropriate.

The *processMouseEvent()* method continues, as follows, with the part which responds to the user releasing a mouse button.

```
0385        case MouseEvent.MOUSE_RELEASED:
0386
0387           if ( morningPrimed) {
0388              if ( morningRect.contains( mouseX, mouseY)) {
0389                 morning = true;
0390              } // End if.
0391              morningPrimed = false;
0392
0393           } else if ( eveningPrimed) {
0394              if ( eveningRect.contains( mouseX, mouseY)) {
0395                 morning = false;
0396              } // End if.
0397              eveningPrimed = false;
0398           } else {
0399              grabbed = false;
0400           } // End if.
0401           this.notifyListeners();
0402           break;
0403
0404        } // End switch.
0405        this.repaint();
0406        super.processMouseEvent( event);
0407     } // End processMouseEvent.
```

The first part of this fragment responds to the mouse button being released when the *morningPrimed* or *eveningPrimed* attributes are set. If these events occur within the appropriate *morningRect* or *eveningRect* Rectangles the value of the *morning* attribute is set, on line 0389 or 0395, as required. Otherwise, releasing the button indicates that the dragging of the hand had finished and so, on line 0399, the *grabbed* flag is set **false**. Whenever the mouse button is released any listeners registered with the component are informed via a call to *notifyListeners()* before the component is repainted and the **super** processMouseEvent() method called.

The remaining methods in the *TimeInput* class are concerned with maintaining the list of listeners and informing any listeners that a change has occurred. This is implemented by making use of the appropriate JFC techniques, as the ChangeListener class is a part of the JFC. The *addChangeListener()* and *removeChangeListener()* methods are implemented as follows.

```
0466     public void addChangeListener( ChangeListener listener) {
0467        listenerList.add( ChangeListener.class, listener);
0468     } // End addChangeListener.
0469
```

```
0470     public void removeChangeListener( ChangeListener listener) {
0471        listenerList.remove( ChangeListener.class, listener);
0472     } // End removeChangeListener.
```

These methods are wrappers on the *listenerList* add() and remove() methods. An EventListenerList instance can contain a heterogeneous list, that is one which contains listeners of different classes. The add() method adds two entries in sequence to the list. The first entry is the class identity of the listener class which is about to be added and is specified by the first argument to the add() method on line 0467. The expression ChangeListener.class is the class identity of the second argument, *listener,* which itself is the actual argument to the call. The same two arguments are required by the *listenerList* remove() method call, on line 0471. The *notifyListeners()* method is implemented as follows.

```
0475     private void notifyListeners() {
0476 
0477     Object[] listeners = listenerList.getListenerList();
0478 
0479        for (int index =  listeners.length-2;
0480                 index >= 0;
0481                 index -= 2) {
0482          if ( listeners[ index] == ChangeListener.class) {
0483             ( (ChangeListener) listeners[ index+1]).
0484                      stateChanged( new ChangeEvent( this));
0485          } // End if.
0486        } // End for.
0487     } // End notifyListeners.
0488 
0489  } // End TimeInput.
```

The method commences, on line 0477, by obtaining the contents of the *listenerList* as an array of Object. The loop between lines 0479 and 0486 then iterates across all pairs of entries in the list and if, on line 0482, the identity of the ChangeListener.class is detected then it is known that a ChangeListener is stored in the next location. Accordingly, lines 0483 and 0484, invoke the stateChanged() method of the ChangeListener stored in the next location of the array *listeners*. The argument to the call of stateChanged() is an instance of the ChangeEvent class which contains a single significant attribute which is the identity of **this** *TimeInput* component.

This completes the implementation of the *TimeInput* component. The following code fragment contains the *init()* and *stateChanged()* methods of the *TimeInputDemo* artifact which was used to obtain the image shown in Figure 7.4.

```
0020  public class TimeInputDemo extends    JApplet
0021                              implements ChangeListener {
0022 
0023     public void init() {
0024 
0025     TimeInput demo = new TimeInput();
0026 
0027        demo.setBackground( Color.white);
0028        demo.setForeground( Color.black);
0029        demo.setFont( new Font( "Dialog", Font.PLAIN, 12));
0030         demo.setBorder( new LineBorder( Color.gray, 4));
0031         demo.addChangeListener( this);
0032         this.getContentPane().add( demo);
0033      } // End init.
0034
0035
0036      public void stateChanged( ChangeEvent event) {
0037         System.out.println( ((TimeInput)
0038                      event.getSource()).getTimeAsString());
0039      } // End stateChanged.
```

The *stateChanged()* method is mandated as the *TimeInputDemo* class states that it implements the ChangeListener interface and must do so in order to be registered as the *TimeInput demo* instance's listener, on line 0031. When the *TimeInput* component becomes visible to the user, who drags and releases one of the hands, or clicks upon an inactive icon, the consequences of the receipt of the *ChangeEvent* will be the output on the terminal of the same string as shown on the clock face.

7.7 Proactive evaluation of the *TimeInput* component

As has been pointed out in the previous section the software engineering of the *TimeInput* component is adequate, but capable of significant improvement. However, the key engineering question is not how well the component is constructed but whether the people who have to use the component will find it to be *effective*, *efficient* and *enjoyable*? These are the three fundamental criteria which are required of an interface by the ISO 9241 usability specification. An interface must be effective in that it allows the user to perform the task that it is intended to allow, in this case the entering of a time of day. It must be efficient in that it allows the user to perform the task without undue effort or stress. It must be enjoyable in that the user will have an overall neutral or positive attitude towards it.

There are essentially two approaches towards ensuring that an interface, or a component part of the interface, satisfies the ISO 9241 requirements: proactive and reactive evaluation. Proactive techniques can be used at the earliest parts of the design stage and can therefore have a persuasive influence upon the production of the component. Reactive techniques can only be used when an artifact, or a prototype of the artifact, is available, so any deficiencies revealed would be more difficult and expensive to correct.

There are a number of proactive techniques available, including having a usability expert review and advise upon the proposed design, adhering to look and feel guidelines for the appearance and behavior of the interface, applying usability checklists to the design, and the use of general heuristics in the production of the design. Of these possibilities the last is the most general-purpose and also the most applicable by untrained practitioners. Accordingly it is the only one which will be discussed in detail in this section. The Appendix contains details of several references which provide much more detailed information concerning the processes discussed in the rest of this chapter.

The best-known set of heuristics are those produced by Jakob Nielsen, which are given in Table 7.1. The first heuristic (Use a simple and natural dialogue) implies that a GUI component should be clearly laid out and uncluttered with a clear mapping between the model as presented on the interface, the underlying application model and the user's cognitive model. The *TimeInput* artifact, as shown in Figure 7.4, attempts to achieve an appropriate layout. The use of a border delineates this component from any other

Table 7.1 Nielsen's heuristics.

1.	Use a simple and natural dialogue.
2.	Speak the user's language.
3.	Minimize the user's memory load.
4.	Achieve consistency.
5.	Provide feedback.
6.	Provide clearly marked exits.
7.	Provide shortcuts for skilled users.
8.	Provide clear error messages.
9.	Prevent errors from happening.
10.	Provide appropriate help and clear documentation.

components on an interface and most of the enclosed space is empty, drawing the user's attention to the clock hands and the icons.

The second heuristic (Speak the user's language) implies for a text-based dialog that the terms used should be related to the task that the user is performing and not to the computer technology that is implementing the system. For example a word processing artifact should refer to the thing that the user is producing as a *document* and not as a *unicode file*. For a GUI component this implies that the representation should be appropriate to the task and the representation of a clock face attempts to achieve this.

The third heuristic (Minimize the user's memory load) implies that information should be supplied to the user rather than making the user remember it. For a GUI artifact this has its clearest expression in the use of menus from which the user can make a choice, compared with a traditional command line system which requires the user to have to remember both the command and its syntax. For the *TimeInput* component the current time, at least within a 12 hour period, is immediately apparent from the position of the hands, removing some memory load from the user.

The fourth heuristic (Achieve consistency) includes consistency within an artifact, between different artifacts and between the artifact and the real world. The possibility of consistency between artifacts might be violated by the *TimeInput* component. The utility to set the system clock provided by Microsoft Windows uses a spin box component positioned below an animated clock face which displays the time accurate to the nearest second. Hence there are two inconsistencies between these two components: the accuracy of the clock and its interactivity. One other *TimeInput* component inconsistency is between it and a real-world clock. It can be predicted that some users will attempt to adjust the hour when they have the minute hand grabbed by dragging it across the 12 o'clock position.

The fifth heuristic (Provide feedback) requires that a clear indication of the state of the system is supplied in a timely manner. The text display of the time shown below the clock face on the *TimeInput* component provides this.

The sixth heuristic (Provide clearly marked exits) refers not only to exiting from an artifact as a whole but also to exiting from its constituent parts. In the context of a GUI this would imply that most, if not all, dialogs which are provided should have a *Cancel* button to allow users to back out of a dialog that they did not intend to post. It also implies that an *undo* capability should be supplied in order that users can restore the state of the artifact if they accidentally trigger a command.

The seventh heuristic (Provide shortcuts for skilled users) could be afforded by the *TimeInput* component by including an editable specialized input component, in place of the simple feedback label, that allows a skilled user to enter a time directly from the keyboard.

The ninth heuristic (Prevent errors from happening), if totally effected, precludes consideration of the eighth heuristic (Provide clear error messages). For the *TimeInput* component it is not possible for the user to indicate an invalid time: the only possibility is for users to indicate 'am' when they intended 'pm'. The consequence of this error, if it can be detected by the artifact making use of the *TimeInput* component, should be clearly stated to the user on a modal dialog.

The tenth heuristic (Provide appropriate help and clear documentation) is possibly the most controversial. One view is that if the user needs help then the interface is already faulty, and in these circumstances – *help doesn't* (help)! For the *TimeInput* component there is no clear indication to users that they can grab and drag the hand to enter a time. One possibility might be to supply a default message in a tool tip attribute indicating this capability to the user. In its absence it is hoped that users will discover this capability, but it can also be predicted that some users will only discover a part of it. They might interact with the component by clicking on the clock face in the approximate location, successively refining the location until the exact required time is shown, rather than discovering that the hands can be dragged. One solution to this, which also illustrates the concept of supplying context-sensitive help, is to build knowledge of this possibility into the component and have a message appear informing the user that dragging is likely to be more efficient if it detects multiple clicking.

The list of heuristics, just discussed, provides a high-level set of principles for the design of a user interface and the interaction that users will have with it. It does not provide any specific guidelines. For example, the commonest form of color blindness is red–green confusion, affecting 6% of males, so these color combinations should be avoided. Another example is that humans are only capable of simultaneously attending to approximately seven (seven plus or minus two) cognitive chunks of information, so an interface should never require the user to have to attend to more than about five different pieces of information.

Detailed guidance on issues such as this can be found in Ben Schneiderman's book, details of which are contained in the Appendix. There is insufficient space in this book to do justice to the importance of these specific guidelines or to the necessary supporting explanations of human perception, learning capability, cognition and memory that allow their applicability to be intelligently evaluated. However, some of these points have already been briefly expounded, as the principles have been employed, in the boxed comments.

The use of state transition diagrams to design the user's interaction with an artifact can also be regarded as a proactive technique. The complexity of an interface needs to be limited in order to be assured that a user will be able to operate it. The degree of difficulty can be inferred from the number of different states, the number of transitions and the complexity of any pre-conditions associated with them. In order to use an interface effectively the user will have to form a cognitive model of how it operates, and the complexity of that model will be related to the complexity of the STD.

If the complexity is within normal human capabilities, seven plus or minus two cognitive chunks, it can be predicted that, providing the means of operating the interface is clearly indicated, an average user will have little difficulty in operating it. If it is not possible to redesign an interface to limit its complexity then it suggests that training will be required before a user will be capable of operating it effectively.

This, however, leads to the question of exactly what one cognitive chunk is, and there is no simple answer to this question. Each state could be operationally identified as a single cognitive chunk and each transition tentatively so identified also. However, if a transition leading from one state to another state, or even back to the same state, is accompanied by another transition that reverses the state change, then it can be argued that these together constitute less than two cognitive chunks. Likewise. a pre-condition can be identified as a cognitive chunk, but if a number of transitions have identical, or substantively similar, pre-conditions then the total number of cognitive chunks will be less than the number of transitions.

This concept of a cognitive chunk is also compounded by another desirable attribute of a good user interface, that of learnability. If an interface has a high degree of internal, or external, consistency then users will be able to predict the behavior of parts of the interface that they have not yet explored. Effectively, a part of an interface that behaves in a comparable manner to another part of the interface has a lower degree of cognitive complexity than would otherwise be expected. As users become more familiar with an interface they are able to *compile* a sequence of actions into a single operation that eventually becomes a single cognitive chunk.

This compiling of actions sometimes has unfortunate consequences. Very few people have ever accidentally formatted a hard disk (although the author must admit to being one of them) as this function is performed so rarely the actions to effect it are never compiled into a single cognitive operation. However, the majority of regular computer users have probably at some stage accidentally deleted or overwritten a single file. These functions are performed so frequently that the sequence of actions becomes a single cognitive operation, and once the first action in the sequence is initiated the remaining actions are performed as if the user is on auto-pilot.

7.8 Reactive evaluation of the *TimeInput* component

Although proactive evaluation of an interface, particularly when performed by a usability expert, can remove, or at least identify, many of the most obvious faults, reactive evaluation when real end-users use the finished interface to perform actual tasks cannot be avoided. It will happen eventually when the product is released by the development team to its users. So it is obviously far better that any faults that will become apparent in this situation are found during the development rather than the maintenance phase of the production process.

The history of computing is already littered with many examples of systems being released to their users and then failing dramatically. For example: one British bank had a very well-established system for its tellers which was operated by text-only terminals connected by dedicated lines to a cluster of mini- and mainframe computers. This system was to be replaced by one which employed a GUI on standard windowing PCs, communicating by secure TCP/IP links. The roll-out of the new system had to be halted at a very early stage when the performance of the tellers on the new system became unacceptable. Eventually the PCs were required to operate as if they were text-only terminals running terminal emulation software communicating with the original back-end computers. There were various reasons for the failure, including faults in the interface and the responsiveness of the PCs, but what is certain is that the problems would have been detected before roll-out if real bank tellers has been required to attempt to operate the new system under realistic, but simulated, conditions.

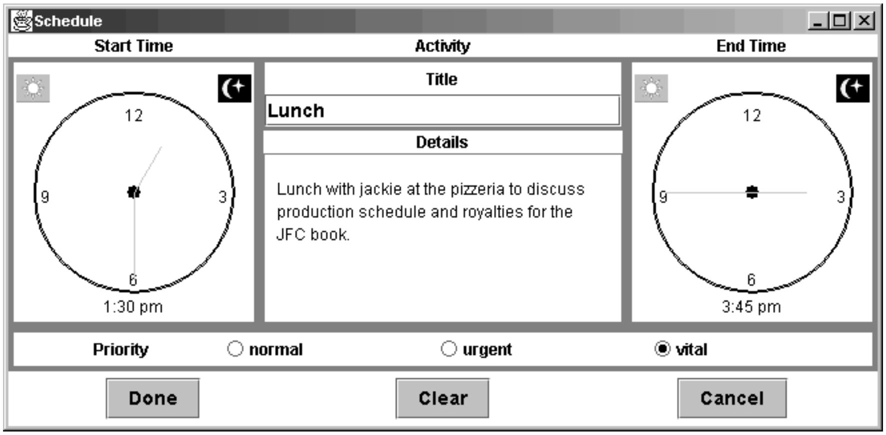
Hence techniques to evaluate an interface reactively should be a planned part of the production process, despite the expense involved. This can be most effectively achieved by producing a usability specification for the interface at the outset of the production process, in the same way that a functional specification for the system is prepared before intensive engineering commences.

A usability specification should state the tasks which are to be performed, the nature of the users who will be performing the tasks and the minimum acceptable levels of performance. These levels should be stated in terms of the time taken to perform the task, the number of errors that are allowed, the time taken to learn how to perform a task (its initial learnability), the time taken to relearn how to perform the same task after a specified period (its memorability) and the users' attitudes towards the interface. This last attribute is possibly the most important, as no system will ever perform effectively if its human parts have a negative attitude towards its technical parts.

If a usability specification is produced at the onset of the production process, reactive evaluation of the completed artifact then becomes a formal testing procedure. That is the quality of the finished product can only be assured by ensuring that it meets its specification. Without a usability specification there is no mechanism whereby the development team can validate the quality of its usability and so formally sign it off.

To illustrate some of these considerations, Figure 7.7 shows two instances of the *TimeInput* class being used as a part of a scheduling application. The left-hand *TimeInput* instance is used to indicate the time at which the activity starts and the right-hand one the time at which it ends. Between them is an area where a title for the activity can be entered and also details of the activity. Beneath this area is a set of radio buttons which can be used to indicate the activities' priority. At the very bottom of the interface are three buttons. The *Done* button will close the dialog panel which contains the interface, posting the information it contains into an electronic diary. The *Clear* button will reset both clocks to the current system time, clear the title and the details areas, and reset the priority to normal. The *Cancel* button will close the dialog without posting the information.

A part of the usability specification for this dialog might be along the following lines.



Schedule

Start Time | Activity | End Time

Title: Lunch

Details: Lunch with jackie at the pizzeria to discuss production schedule and royalties for the JFC book.

Start Time: 1:30 pm

End Time: 3:45 pm

Priority ○ normal ○ urgent ● vital

Figure 7.7 The *TimeInput* component as part of a system's interface.

> The subjects should be regular users of office productivity applications. They should feel sufficiently confident to enter activity details after having it demonstrated to them once. They should be able to set the Start and End times with perfect accuracy and be able to enter the title and details with only minor typos. Only an occasional missetting of the priority, caused by inattention, is acceptable. Likewise, only an occasional error in selecting the Done, Clear and Dismiss buttons is allowed. An activity should take no longer than 2½ minutes to enter.
>
> Users will be asked to enter details of two activities, with normal and vital priorities, via a simulated scripted phone call. During entry of the second activity they will be distracted from the task by the ringing of a second telephone, which they have been instructed to attend to. They are expected to be able to continue the task without more than a 5-second period to re-familiarize themselves with the state of the interface. The second activity will be cancelled, via the Cancel button, at the end of the phone call.
>
> After using the interface the users should agree with the statement:
>
> *I would be happy to use the two-clock dialog panel to enter details of my own appointments.*

The first sentence of the first paragraph of this specification indicates who the intended users of the application are. The rest of the paragraph states clear performance targets for operating this part of the application's interface. The second paragraph is attempting to cause the application to be used in a realistic simulation of an office environment. It acknowledges that users in this situation are often interrupted in mid-task, having to return to it after dealing with the interruption. The last paragraph is ensuring that the users will have a positive attitude towards using the application and its interface.

In order to conduct this usability evaluation typical end-users of the application would have to be recruited and an office-style environment prepared for them. For any interface, testing it with a single user would not be sufficient. However, for an interface with this degree of complexity, it would be unlikely that any benefit would be obtained from investigating more than half a dozen users, providing they were selected to provide a representative subset of all possible users including a mix of sexes, ages, computer experience and job roles.

An example of the importance of finding a sufficiently representative set of potential users comes from the usability investigation of a personal diary application by a major manufacturer. The diary had an auto-alarm function, that is it would post a dialog to attract the user's attention to an activity shortly before its start time.

When tested, users who had very few scheduled activities found this very useful. However senior managers, who spend the majority of their time in meetings, found it irritating. The investigation resulted in a prominent control to enable or disable this function but, more importantly, illustrated that the job role of a user has an impact upon functionality and hence usability.

The investigation itself should be conducted in a standardized manner. The investigator should attempt to ensure that the experience of each subject is comparable. This is achieved by the investigator attempting to follow a prepared script, which should be validated by a dry run before the actual investigation is performed. The script should commence by stressing that the investigation is attempting to evaluate the interface not the performance of the subject, and also that they are free to abandon the procedure at any stage. Despite this some users may feel that any failure of performance is attributable to them rather than to inadequacies in the interface that they are using. The investigator should be prepared for this possibility and if the subject seems to be experiencing undue stress the investigator can make the decision to finish the procedure early, if possible without the user being fully aware of this.

During the investigation a second investigator should be tasked with observing the user's behavior, possibly backed up by video recordings of the user and of the computer screen. These recordings should only be used if there is any uncertainty from the observer's notes and recollections of the user's behavior. Conducting the investigation will require the time of at least two investigators and having to devote additional time to reviewing the recordings will add to the cost. The observation may reveal some factors, which, although they do not impinge upon the performance as defined by the usability specification, are nonetheless of interest. For example when the *TimeInput* component was informally evaluated a majority of the users were observed to attempt to adjust the hour value by moving the minute hand across the 12 o'clock position and one used the multiple clicking method for setting the time

One further possibility for the conduct of an investigation is to ask the users to talk aloud about what they are thinking and doing. This can be valuable, but it also has two major disadvantages. Firstly, it is an artificial situation, as most users do not talk to the computers as they operate them, and if they do it is unlikely to be a detailed commentary on how they are interacting with them. Secondly, some subjects will be comfortable with this procedure, while others will not, and this difference can have an impact upon their performance.

In order to assess the users' attitude towards the interface there will have to be a process after the task has been completed. This can be accomplished by means of a printed questionnaire or by means of a structured interview conducted by an investigator. Whichever method is used, users should always be thanked for their participation, reminded that it was the quality of the interface which was being investigated and not their own performance, and arrangements made for them to have access to the results of the investigation when they are available.

This brief description of how to conduct a usability investigation gives only an initial overview of some of the issues involved; more details can be obtained from the references listed in the Appendix. The most fundamental message from this description is the importance of ensuring that the subjects of the investigation are treated with respect and that the investigators are ethically obliged to attempt to ensure that they suffer no undue stress from their participation.

Even if a formal usability investigation of the finished product is conducted (and in many cases it is abandoned, curtailed or performed far too late to have any impact due to slippage in the production schedule), it is, however carefully it is planned and executed, an artificial scenario.

The only true test of the quality of the interface is how real end-users actually interact with it when using it in the real world to perform real tasks. In many cases, interfaces which have passed a formal usability investigation have still had major faults discovered when they are deployed. One possible cause of this consequence is that the subjects might be more forgiving of faults in the interface during the investigation as they are attempting to please the investigators, or hold the belief that if they are critical of the interface it will reflect badly upon the production team.

A project budget should take this into account and allow for continued usability evaluation after deployment. One way of achieving this within an organization is to monitor the faults reported to the organization's computer help desk in the early stages of its deployment. The experiences of the help desk operators when assisting users experiencing difficulties in operating an application can be a very valuable input into the usability evaluation process at this stage.

One final consideration on reactive usability evaluation concerns the differences between usability investigations and usability experiments. These are two very different processes, and even usability professionals do not always seem to appreciate the differences between them. The procedures described above are relatively informal, the intention being to simulate the use of an artifact in a real-world scenario in an attempt to identify any faults before it is deployed.

A usability experiment is a much more formal process that attempts to obtain evidence to support or refute a general principle, or theory, of human–computer interaction. An experiment requires that a hypothesis, known as the experimental hypothesis, is formulated, and the process is designed so as to attempt to disprove the experimental hypothesis in an attempt to support an alternative, contradictory, hypothesis known as the null hypothesis.

For example, the principle of interface consistency would suggest that the location of elements on a menu should be immutable. However, the principle of allowing users to configure an interface to their own preferences would suggest that allowing them to rearrange the elements would be beneficial. Hence an experimental hypothesis could be that allowing a user to rearrange the location of elements on a menu would result in an improved overall performance. The null hypothesis would be that allowing such a configuration would have no effect or a negative effect. The procedures concerned with conducting an experiment are more stringent than those for an investigation and include the use of statistical manipulation of the measured outcomes to ensure that any effect observed is real and not just a random event. Precise details of the differences can be found in the references already cited. However, the results of a usability experiment will have much greater applicability than those of an investigation, justifying the additional investment.

Summary

- The usability design of an artifact should take into account the characteristics of its intended users, the task they will be performing with it and the environment they will be working within.
- A user interface should be effective, efficient and enjoyable.
- Proactive usability engineering activities are likely to be the most cost-effective and include checklists, style guides, usability engineers' opinions and heuristics.
- Neilsen's list of usability heuristics is widely known and can be applied by relatively unskilled engineers.
- Reactive usability engineering activities are essential and include investigative and experimental techniques.
- A usability specification should indicate precisely the task to be performed and the minimal acceptable level of performance. It can be used as part of reactive evaluation to prove that the artifact has met its design specifications.
- A software engineer working on a user interface is by definition a usability engineer and (arguably) vice versa.
- Whenever live investigation of an interface is taking place there is an ethical imperative upon the investigator to ensure that the users suffer no undue distress.

Exercises

7.1 Add a second hand to the *TimeInput* component and investigate if the user's performance is unduly impeded by having to interact with it.

7.2 Conduct a usability investigation of the *TimeInput* component as described in this chapter. While the users are completing the task, carefully observe them to see whether they commit any of the 'errors' predicted in the text.

7.3 Design and implement the spin box-based *TimeInput* component shown in Figure 7.1. When it is available, use it and the *TimeInput* component as developed in this chapter with two groups of users to attempt to demonstrate which is the most efficient, effective and enjoyable.

7.4 Conduct a usability investigation of any of the artifacts produced in the previous chapters or suggested as end of chapter exercises. To do this, define the intended users, establish a specific task and define minimum performance standards. Do any of the artifacts meet the required standards? How can they be improved so as to meet their specification more closely?

7.5 Implement the 12 o'clock wind-over behavior and any other improvements suggested in the chapter. Repeat the usability investigation from Exercise 7.2 and attempt to establish whether the component is not more effective, efficient or enjoyable.

7.6 Extend the *TimeInput* component to produce a *Clock* component which contains an internal Timer and behaves as a desktop clock. How can you now supply the user with the capability to change the time?

7.7 Extend the *Clock* component from Exercise 7.5 to implement an *AlarmClock* supplying the user with a mechanism to set the alarm. When it is available, conduct a usability investigation and consider any improvements that might be needed to its usability. Implement the improvements and conduct a second investigation to see whether its usability has been improved.

8

An HTML viewer and editor – introducing *JTextArea*s

8.1 Introduction

In Chapter 4 the single-line text input area component, JTextField, was introduced. The JTextField class is a part of the JTextComponent hierarchy, which includes a number of multiple line text areas. In this chapter these more complex and extensive multiple line text components will be briefly introduced by the development of a primitive Web browser and a simple HTML editor[1]. JTextComponents have a compound internal architecture which makes them very powerful to use, but dauntingly complex to extend. Accordingly, as this is an introductory book, only the use, and not the extension, of these components will be illustrated. However, even in an unextended state, because of their inherent complexity they provide very powerful facilities for the display of different forms of text-based documents.

The development of these artifacts will also require the introduction of a number of other components. In order to obtain an HTML page to be displayed a Web connection will have to be opened to its location and the contents obtained across the Internet. This capability is supplied by the URL (*U*niform *R*esource *L*ocator) class, which encapsulates the complex processes of opening, obtaining and closing an Internet stream. The HTML page will be shown to the user within a JScrollPane which contains vertical and horizontal scroll bars allowing the user to pan around the document. The artifacts also offer the capability to view an HTML document in one scroll pane and its raw HTML source in a second pane. These are presented within a JSplitPane which allows the user to adjust the amount of space afforded to each of the parts.

The final artifact presented in this chapter will describe a very primitive HTML editor in order to illustrate briefly the handling of Document events. In the following chapter the viewing component developed in this chapter will be used to illustrate the use of multiple internal windows within an artifact.

8.2 Text class hierarchy and architecture

The class hierarchy for the JFC text components is given in Figure 8.1. It shows that the JTextField, as used in Chapter 4, is a child of the abstract JTextComponent class which

[1] Although the JFC components are being used to illustrate HTML capability in this chapter, these components are general-purpose and very complex. For production purposes a third-party HTML component (such as the ICE browser, available from `http://www.icesoft.no/`) may be more appropriate.

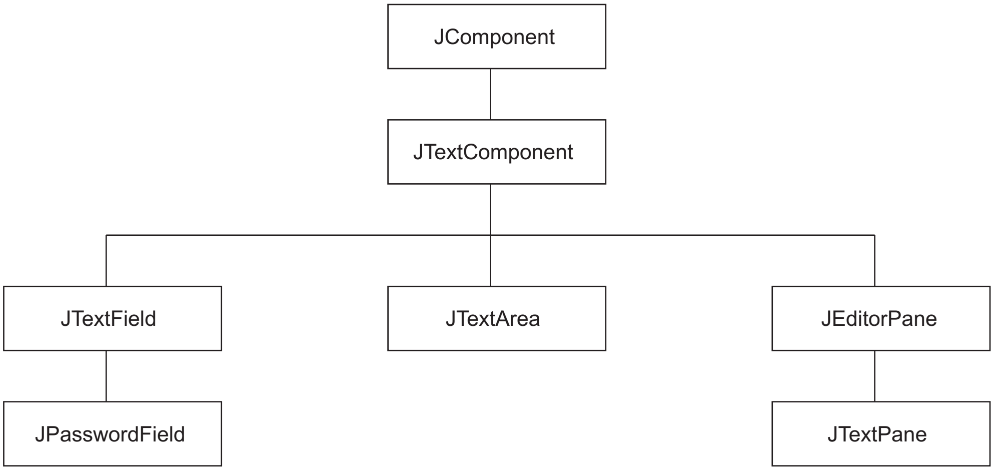


Figure 8.1 JFC text component hierarchy.

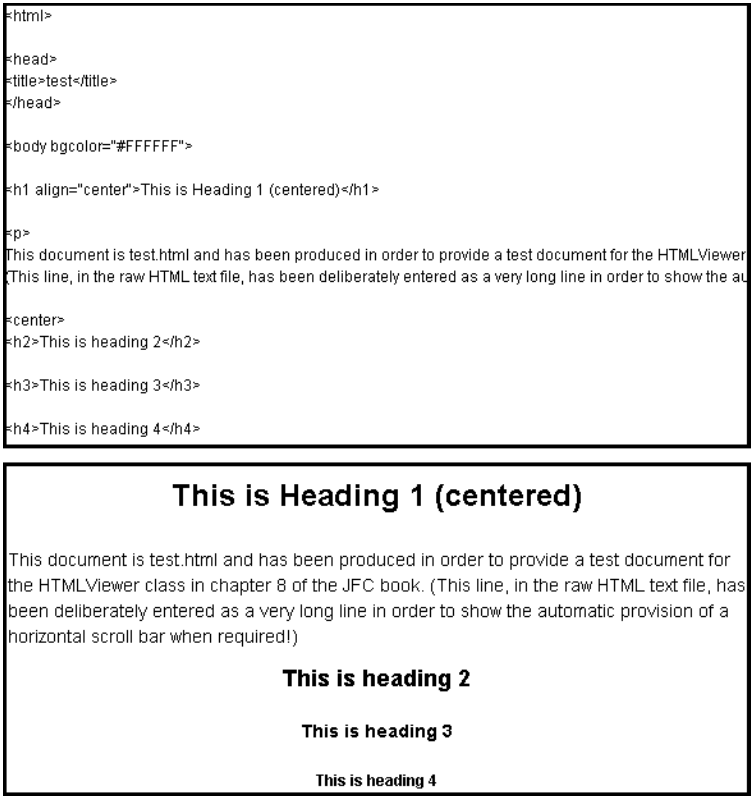
supplies it with its essential text-handling capability. The JTextField has a single child class, JPasswordField, which operates in an identical manner to a text field except that all characters typed in by the user are displayed on the screen using a single character controlled by its echoChar attribute. As its name suggests, this component can be used when confidential information, for example a password or personal identification number, needs to be obtained from a user.

The JTextArea class supplies a multi-line area for the display of *plain text*, that is the text supplied to it in its text resource will be rendered exactly as it is supplied, with no attempt at interpretation. This is in contrast to JEditorPane, which may attempt to interpret and render the text supplied. For example HyperText Markup Language (HTML) documents are obtained as a stream of plain text containing tags that are interpreted by a rendering engine to supply the graphical representation of the text that the user will see. Figure 8.2 shows an HTML fragment as rendered by a JTextArea and by a JEditorPane.

HTML is not the only system which uses plain text as a transmission and storage medium for edited text; others include Rich Text Format (RTF), PostScript and LaTeX. The JFC has built in, but limited, support for HTML and RTF only. The JTextPane component is supplied as a convenient point for the development of other specialized text rendering components. For example it could be extended to supply a specialized component for the display of Java source code; using bold for reserved words, italic for user-declared terms etc. The details of how to accomplish this are outside the scope of this book.

All of the JTextComponent classes use the same architecture for transforming the plain text contained within the component into a graphical representation on the screen. The architecture is illustrated in Figure 8.3, which shows that a JTextComponent contains a private Document instance containing the raw text. The text is passed to an appropriate ViewFactory that will render the text according to its style: plain, HTML, RTF or other. When the user interacts with the text in some way, for example typing or deleting some text, the Controller passes the changes to the Document and the image is re-rendered.

This is another example of a three-layer division of responsibility: the Document instance contains a model of the text, the ViewFactory provides a view of the text and the Controller controls the relationship between the other two parts. This *model*, *view*,



```
<html>

<head>
<title>test</title>
</head>

<body bgcolor="#FFFFFF">

<h1 align="center">This is Heading 1 (centered)</h1>

<p>
This document is test.html and has been produced in order to provide a test document for the HTMLViewer
(This line, in the raw HTML text file, has been deliberately entered as a very long line in order to show the au

<center>
<h2>This is heading 2</h2>

<h3>This is heading 3</h3>

<h4>This is heading 4</h4>
```

This is Heading 1 (centered)

This document is test.html and has been produced in order to provide a test document for the HTMLViewer class in chapter 8 of the JFC book. (This line, in the raw HTML text file, has been deliberately entered as a very long line in order to show the automatic provision of a horizontal scroll bar when required!)

This is heading 2

This is heading 3

This is heading 4

Figure 8.2 Raw HTML and interpreted HTML.

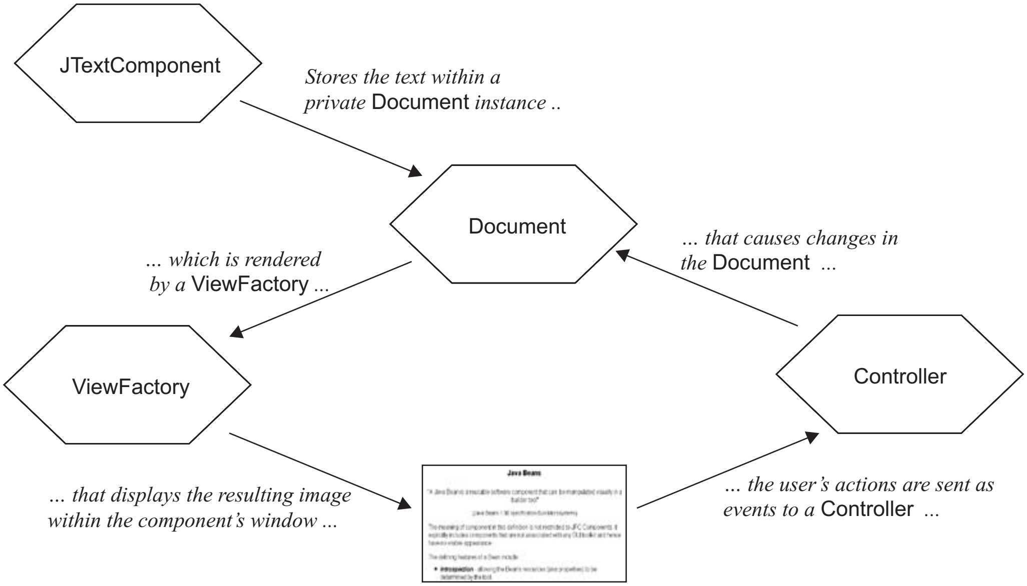

JTextComponent
Stores the text within a private Document *instance ..*
Document
... which is rendered by a ViewFactory *...*
... that causes changes in the Document *...*
ViewFactory
Controller
... that displays the resulting image within the component's window ...
Java Beans
... the user's actions are sent as events to a Controller *...*

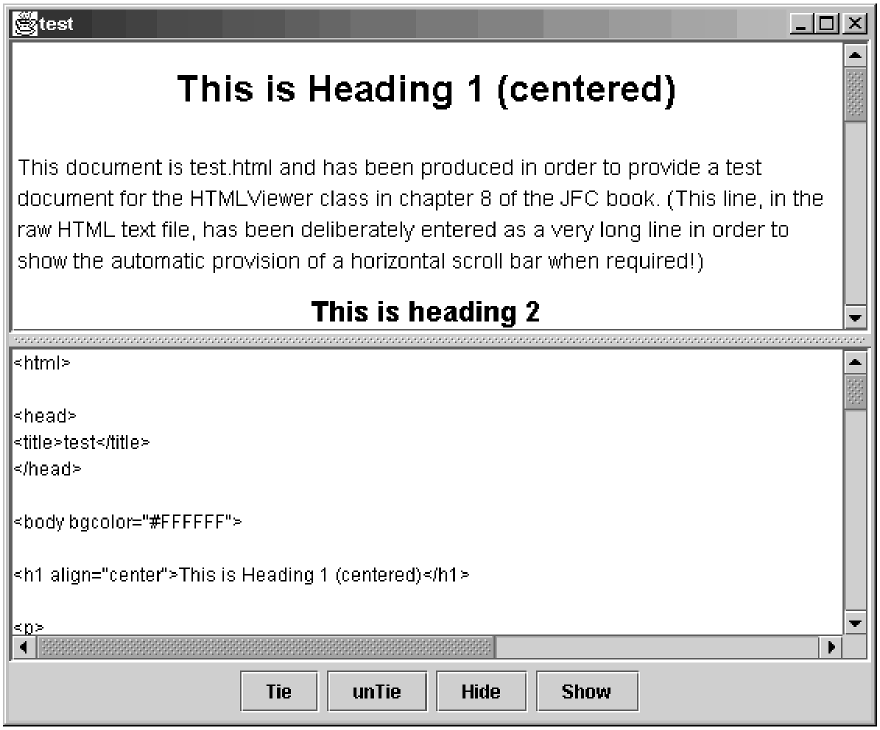
Figure 8.3 JTextComponent rendering architecture.

controller, architecture contains the same division of responsibility as is contained within the *application*, *presentation* and *translation* three-layer division of responsibility that is being used for the design and construction of complete artifacts. However, the use of this architecture is facilitated by the swing components that encapsulate the complexity within themselves. To illustrate this the next part of this chapter will show how different ViewFactories can be obtained, automatically, to render the same text in different ways.

8.3 The *HTMLViewer*: overview

In this part of the chapter an artifact, called *HTMLViewer*, which allows HTML pages to be displayed both as plain text and as rendered images, will be developed. The visual appearance of the artifact is shown in Figure 8.4. The top-level window is divided horizontally into two areas, an upper area showing the rendered HTML and a lower area showing the raw HTML. The proportion of space used for each of these areas can be varied by the user dragging the bar separating them up or down.

Each of the windows contains a vertical scroll bar that can be used to pan up and down the contents of the window. In the illustration the extent of the sliders is indicating that approximately one third of the rendered HTML is visible and approximately one fifth of the raw HTML. These two scroll bars can be linked together so that if the user adjusts the position of the slider in one of the windows the slider, and also the view, in the other window will be adjusted appropriately. The upper (rendered) window does not have a horizontal scroll bar, as an HTML document will adjust itself to the available width. The lower (raw) window is supplied with a horizontal scroll bar, as a line of raw text can



This is Heading 1 (centered)

This document is test.html and has been produced in order to provide a test document for the HTMLViewer class in chapter 8 of the JFC book. (This line, in the raw HTML text file, has been deliberately entered as a very long line in order to show the automatic provision of a horizontal scroll bar when required!)

This is heading 2

```
<html>

<head>
<title>test</title>
</head>

<body bgcolor="#FFFFFF">

<h1 align="center">This is Heading 1 (centered)</h1>

<p>
```

Figure 8.4 *HTMLViewer*: visual appearance.

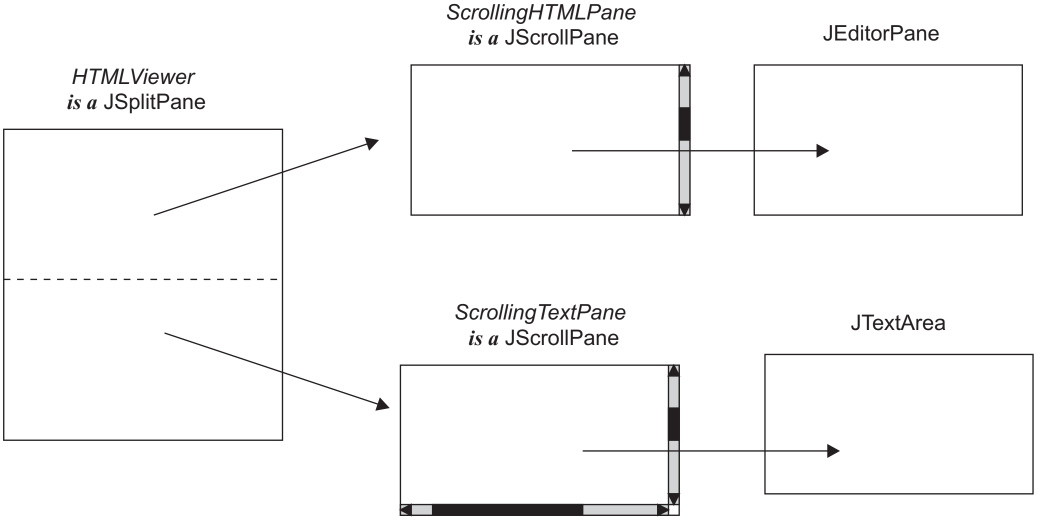


Figure 8.5 *HTMLViewer* layout diagram.

contain an indefinite number of characters and so may extend beyond the right-hand limit of the window.

Figure 8.5 contains a layout diagram indicating the components that are used to construct an *HTMLViewer* instance. It shows that the top-level window is an instance of the *HTMLViewer* class, which is an extended JSplitPane. The JSplitPane class is a container class that can manage exactly two children, separated by a bar which the user can adjust to vary the proportion of space accorded to each of the children. In this example the children are arranged vertically and so have a horizontal bar; the alternative is for them to be arranged horizontally alongside each other, requiring a vertical bar.

Each of the children in an *HTMLViewer* is an extended instance of the JScrollPane class. This class can manage exactly one child, whose size can be larger than that of the window accorded to the JScrollPane instance. The horizontal and vertical JScrollBars, which are provided automatically, can then be used to move its viewport across the underlying child component. For example, in the upper part of Figure 8.4 the rendered HTML is displayed in a window whose vertical extent is much larger than the height of the JScrollPane's window. The parts of the rendered HTML that cannot currently be seen can be made visible by dragging the vertical JScrollBar's slider downward, causing the JScrollPane's *viewport* to move downwards across the underlying window, making the rendered text apparently scroll upwards. This scrolling behavior is supplied automatically by a JScrollPane and does not require any code to be supplied by the developer to effect it. The JScrollPane class was introduced in detail when it was used to construct the *FontViewer* artifact in Chapter 2.

The upper area of the JSplitPane contains an instance of the *ScrollingHTMLPane* class that contains an instance of the JEditorPane in its viewport. This instance has been supplied with an HTML document and, as can be seen in Figure 8.4, has automatically installed an appropriate ViewFactory to render it. The lower area contains an instance of the *ScrollingTextPane* class whose viewport contains an instance of the JTextArea class. This instance has been supplied with the same HTML document and is displaying it as plain text. The effect is that the upper and lower parts of the artifact are showing different views of the same document. Internally the two JTextComponents are using different ViewFactories, as indicated by the architecture in Figure 8.3.

8.4 The *ScrollingTextPane* class

To start to explain the construction of the *HTMLViewer* the visually simpler of the scrolling windows, the *ScrollingTextPane*, will be described. As shown in Figure 8.5, this component is an extended JScrollPane containing a JTextArea instance in its viewport. The *ScrollingTextPane* class also has the capability, given a Internet address, to retrieve the contents of the HTML document from that address and display them within the text area. The class diagram for the *ScrollingTextPane* class is given in Figure 8.6.

The class diagram shows that the *ScrollingTextPane* class extends the JScrollPane class and is contained in the *brewser* package of classes. It has four attributes. The *textPage* an instance of JTextArea upon which the raw HTML will be shown as plain text. This attribute has protected visibility in order to facilitate extension of this class, as will be demonstrated later in the chapter. The three remaining attributes are all private. The *rawHTML* is stored as a String and the address from where the HTML was obtained stored both as a String in *address* and as a URL in *location*. Addresses of Web resources on the

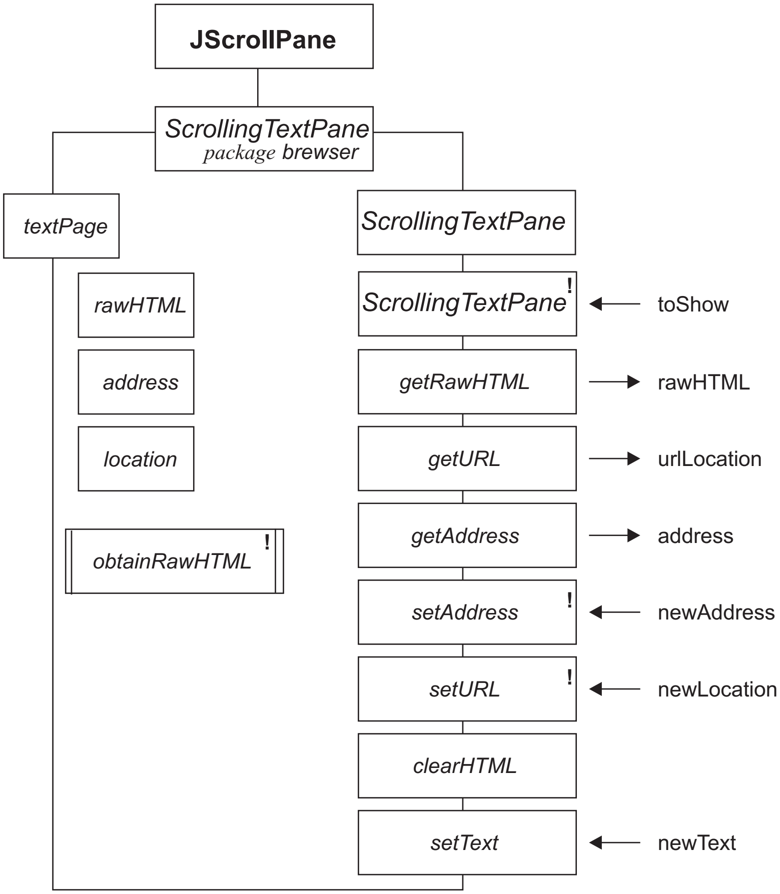


Figure 8.6 *ScrollingTextPane* class diagram.

Internet take the form protocol://server/filepath (e.g. http://www.scism.sbu.ac.uk/jfl) and are known as Uniform Resource Locators or URLs. The java.net.URL class provides objects that can be used to establish and obtain resources across the Internet with the minimum of developer effort, as will be shown below.

The *ScrollingTextPane* class also contains a private action called *obtainRawHTML()*, which will be used to open an Internet connection and retrieve the HTML contents from it. As this action is fraught with possible errors it is shown as possibly throwing an exception. The default *ScrollingTextPane* constructor creates an empty instance. The alternative constructor takes a String representation of an Internet address, which it will use to obtain its initial HTML contents; as this process is also fraught with potential errors it is noted as possibly throwing an exception.

The next three public methods are inquiry methods supplying the values of the three corresponding attributes. The two public methods that follow allow a new HTML document to be specified by supplying an Internet address either as a String or as a URL; both of these methods may also throw an exception during this process. The *clearHTML()* method clears all the attributes and also the text displayed in the window. Finally, *setText()*allows the HTML contents to be set directly. When the *setText()* method is used to provide the component with the HTML document that it is to display, the location of the document is unknown and accordingly the *address* is set to '*Unknown*' and *location* to **null**.

If an error is encountered in obtaining an HTML page during construction then an appropriate exception will be thrown and all attributes will be cleared. If an exception is thrown during *setAddress()* or *setURL()*, the attributes are left in the same states that they were in before the method was called. The start of the implementation of the *ScrollingTextPane* class, as far as the end of the default constructor, is as follows.

```
0011  package brewser;
0012
0013  import javax.swing.*;
0014  import javax.swing.text.*;
0015  import java.awt.*;
0016  import java.io.*;
0017  import java.net.*;
0018
0019
0020  public class ScrollingTextPane extends JScrollPane{
0021
0022  protected JTextArea textPage = null;
0023  private   String    rawHTML  = null;
0024  private   String    address  = null;
0025  private   URL       location = null;
0026
0027      public ScrollingTextPane( String toShow) {
0031         this( “”):
0032      } // End ScrollingTextPane default constructor.
```

Following the **package** declaration and the necessary **import**ations, the class is declared on line 0020. Lines 0022 to 0025 then declare the four data attributes, as shown on the class diagram. The *textPage* attribute is declared as **protected** as there is no **public** inquiry method to obtain its identity, so if the class is ever extended the child class could not obtain its identity if it were **private**. The default constructor indirects to the alternative constructor, passing an empty string as an argument. The declaration of the alternative constructor is as follows.

```
0034      public ScrollingTextPane( String toShow)
0035                          throws BrewserException {
```

This constructor requires a String containing the Internet address of the HTML page *toShow*. It is also declared as possibly throwing a *BrewserException*. This exception class is contained within the *brewser* package and extends the RuntimeException class. A RuntimeException, as opposed to a non-RuntimeException, need not be caught and handled if it is thrown. As such it is informative rather than prescriptive, and will be used by the client only so that it can become aware that an attempt to load an Internet resource has been unsuccessful. The implementation of this class, presented without comment, is as follows.

```
public class BrewserException
                         extends RuntimeException {

public final static int UNKNOWN    = 0;
public final static int BAD_URL    = 1;
public final static int LOAD_FAULT = 2;
private int theReason              = UNKNOWN;

   public BrewserException() {
      this( UNKNOWN);
   } // End BrewserException default constructor.

   public BrewserException( int reason) {
      super();
      theReason = reason;
   } // End BrewserException  constructor.

   public int getReason() {
      return theReason;
   } // End getReason.

   private String getReasonAsString() {
      switch ( theReason) {
         case BAD_URL    : return "URL incorrect";
         case LOAD_FAULT : return "Load unsuccessful";
      } // End switch.
      return "Unknown";
   } // End getReasonAsString.

   public String toString() {
      return "BrewserException " +
                this.getReasonAsString();
   } // End toString.

} // End BrewserException.
```

The implementation of the alternative *ScrollingTextPane* constructor, given below, commences on lines 0037 and 0038 by calling the **super**, JScrollPane, constructor. The two arguments are manifest values indicating that the vertical and horizontal scroll bars should always be displayed by the component, even when they are not required. Details of the JScrollPane class were given in Chapter 2.

```
0034      public ScrollingTextPane( String toShow)
0035                          throws BrewserException {
0036
0037         super( JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
0038                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
0039
0040         address  = new String( toShow);
0041         textPage = new JTextArea();
0042         textPage.setEditable( false);
0043         this.getViewport().add( textPage);
0044
0045         this.setMinimumSize(   new Dimension(100, 50));
0046         this.setPreferredSize( new Dimension(200, 200));
0047
0048         if ( toShow.length() > 0) {
0049            this.obtainRawHTML( address);
0050            textPage.setText( rawHTML);
0051         } // End if.
0052   } // End ScrollingTextPane constructor.
```

The constructor continues, on line 0040, by making a copy of the address *toShow* in the *address* attribute, before line 0041 constructs an empty JTextArea initializing the attribute *textPage* to reference it. The *textPage* will possibly be supplied with some text to display before the constructor ends, as will be described below. In this artifact the user is not to be allowed to edit the text, so on line 0042 the *textPage*'s editable attribute is set **false**.

Following this, on line 0043, the *textPage* is added to **this** *ScrollingTextPane*'s viewport. This line implements the layout relationship illustrated at the bottom of Figure 8.5. By installing the *textPage* into a JScrollPane's viewport it is automagically linked to its scroll bars. When the scroll pane becomes visible on the desktop it will, possibly, show only a proportion of the extent of the *textPage* and the user can adjust the proportion shown by interacting with the scroll bars. This behavior is implicit within a JScrollPane instance and does not need any developer effort to effect it.

JComponent → JTextComponent → javax.swing.JTextArea

```
public JTextArea ()
public JTextArea ( String toShow)
public JTextArea ( int rows, int columns)
public JTextArea (String toShow, int rows, int columns)
```

Constructs a JTextArea instance with various combinations of the String *toShow* and the (approximate) number of characters in a *row* and the number of *columns*.

```
public int  getColumns()
public void setColumns( int newColumns)
public int  getRows()
public void setRows(   int newRows)
```

Inquiry and setting methods for the number of *rows* and *columns*.

```
public void append( String toAppend)
public void insert( String toInsert, int offset)
public int  replaceRange( String replaceWith
                          int fromHere, int toHere)
public String getText()
public String getText( int fromHere, int toHere)
```

Methods to manipulate the *text* contained in the component. The getText() methods are inherited from JTextComponent.

```
public boolean getEditable()
public boolean setEditable(boolean yesOrNo)
```

The *editable* attribute determines if the user can change the text from the keyboard, default **true**.

```
public int     getLineCount()
public int     getLineStartOffset( int lineNumber)
public int     getLineEndOffset(   int lineNumber)
public int     getLineOfOffset(    int offset)
public boolean getLineWrap()
public void    setLineWrap( boolean yesOrNo)
public boolean getWrapStyleWord()
public void    setWrapStyleWord( boolean yesOrNo)
```

The text is considered as a sequence of lines. These methods allow the number of lines and their start and end locations within the text to be determined. The *lineWrap* attribute determines if lines that are wider than the width of the window are broken and continued on the next line (default **false**). The *wrapStyleWord* attribute works in concert to break lines at word boundaries.

The next part of the constructor, on lines 0045 and 0046, establishes *minimum* and *preferred* sizes for the component. This step is needed if the component is to be installed into a JSplitPane, which has complex requirements for layout negotiations.

The final part of the constructor, on lines 0048 to 0051, attempts to obtain and install the raw HTML into the JTextArea instance. It is this part of the construction that may throw a *BrewserException* which, if propagated from the constructor, will cause the ScrollingTextPane instance not to be constructed. This is a Java rule: if a constructor throws an exception then the construction attempt is abandoned, the instance is destroyed and the object reference is **null**. This may be appropriate if an unsuccessful attempt is made to obtain an Internet resource, but is not appropriate if the alternative constructor was called from the default constructor. Accordingly this final stage is contained within a guard, on line 0048, which ensures that if an empty address was supplied in the *toShow* argument there is no possibility of an exception being thrown.

Assuming that a non-empty address has been supplied, construction continues by calling the **private** *obtainRawHTML()* method. If this method does not throw an exception it will place the retrieved contents of the *address* into the *rawHTML* attribute. Accordingly, on line 0050, the contents of *rawHTML* are supplied as the *textPage*'s text attribute, which will cause it to display them when it becomes visible.

Should the *obtainRawHTML()* method throw a *BrewserException* it indicates that the contents of the *address* could not be retrieved. The exception is automatically propagated from the constructor so that the *ScrollingTextPane*'s client knows that the page requested in the argument could not be retrieved and the instance has not been constructed.

The implementation of the **private** *obtainRawHTML*() method is as follows.

```
  private void obtainRawHTML( String fromHere)
                              throws IOException {

  URL localLocation = null;

  final int BUFFER_SIZE   = 512;
  byte           buffer[] = new byte[ BUFFER_SIZE];
  StringBuffer obtained   = new StringBuffer( "");
  int     bytesObtained   = 0;
  InputStream  readFrom   = null;

  try {
       localLocation = new URL( fomHere);
       readFrom = localLocation.openStream();
       bytesObtained = readFrom.read( buffer);
       while ( bytesObtained >= 0) {
          obtained.append( new String( buffer,
                                       0, bytesObtained));
          bytesObtained = readFrom.read( buffer);
       } // End while.
       readFrom.close();

       rawHTML  = obtained.toString();
       address  = fromHere;
       location = localLocation;
  } catch ( MalformedURLException exception) {
     throw new BrewserException(
                        BrewserException.BAD_URL);
  } catch ( IOException exception) {
     throw new BrewserException(
                     BrewserException.LOAD_FAULT);
  } // End try/catch.
} // end obtainRawHTML.
```

This is a complex method which, as shown on the class diagram, may throw an exception. An overview of its operation is that it will first convert the String argument *fromHere* into a URL and then open an InputStream to the location. Having established a connection across the Internet to the resource, its contents are then obtained by reading them sequentially into a **byte** *buffer* and concatenating them into a StringBuffer. When all of the contents have been obtained the InputStream is closed, the contents of the StringBuffer are converted to a String and stored in the *rawHTML* attribute and the *address* and *location* attributes updated to reflect a successful operation. Should any exceptions be thrown during this operation this final stage will not be effected, so the value of the attributes will not change.

The method is declared on lines 0110 and 0111 and a local URL variable called *localLocation* is declared on line 0113. The next set of declarations, on lines 0115 to 0118, prepare for buffering the contents of the HTML page. These start, on line 0115, with a constant for the size of the buffer. The *buffer* is declared as an array of **byte**s on line 0116, following which an empty StringBuffer called *obtained* is constructed on line 0117.

The steps of the method commence on line 0121 where an attempt is made to construct a URL, in *localLocation*, from the *fromHere* String argument. This step might throw a MalformedURL exception. If the *fromHere* argument contains a valid URL an InputStream, called *readFrom*, is opened on line 0122. This step will open a stream connection across the Internet from the server where the HTML file is stored into this process, and may throw an IOException, as may any of the other steps taken with the *readFrom* instance.

Assuming no exception has been thrown, line 0123 will attempt to fill the *buffer* with bytes from the stream; if no IOException is thrown the read() method will return the number of *bytesObtained*. The **while** loop, between lines 0124 and 0128, will then iteratively append the *buffer* to the StringBuffer *obtained* and replenish the buffer from the stream. When all bytes have been obtained the value of *bytesObtained* will be –1 causing the loop to terminate. The effect of this fragment is to transfer the contents of the HTML document on the remote server into *obtained*, and so, on line 0129, the stream is closed.

Object → java.net.URL

```
public URL( String fullSpec)
public URL( String protocol, String host, int port, String file)
```

Constructs a URL either from a *fullSpec* (e.g. `http://www.sbu.ac.uk/jfl:80`)or from the four component parts.

```
public String getProtocol()
public String getHost()
public int    getPort()
public String getFile()
```

Inquiry methods for the component parts.

```
public InputStream openStream() throws IOException
```

Creates an InputStream to the resource identified in the URL.

If any exceptions were to be thrown they would have been thrown prior to line 0131, so it is now safe, on lines 0131 to 0133, to update the instance attributes from the argument and local variables. The effect of the action is to leave the state of the instance unchanged if any exception is thrown while an attempt is made to retrieve the contents of the *fromHere* Internet address argument, and to propagate the exceptions from the method. If no exception is thrown then the values of the attributes are updated to indicate the location and contents of the address.

The final part of the *obtainRawHTML()* method, on lines 0134 to 0139, catches any MalformedURLExceptions or IOExceptions that are thrown and propagates them as instances of the *BrewserException* class with appropriate manifest reasons. The three *ScrollingTextPane* class inquiry actions are implemented, as expected, as follows.

```
0051      public String getRawHTML() {
      return rawHTML;
   } // end getRawHTML;

   public URL getURL () {
      return location;
   } // end getURL.

   public String getAddress() {
      return address;
   } // end getAddress.
```

The *setURL()* method requires a URL as an argument and is implemented as a call of the other state-setting method, *setAddress()*, converting the URL *newLocation* into a String, as follows.

```
   public void setURL( URL newLocation)
                       throws BrewserException {
      this.setAddress( newLocation.toString());
   } // End setURL.
```

The *setAddress()* method calls *obtainRawHTML()*, passing on its argument *newAddress*. As this method does not have an exception handler, any exception thrown by the *obtainRawHTML()* method will be propagated onwards. If no exception is thrown then the contents of the *newAddress*, which will be in *rawHTML*, are installed into the *textPage* by calling its setText() method.

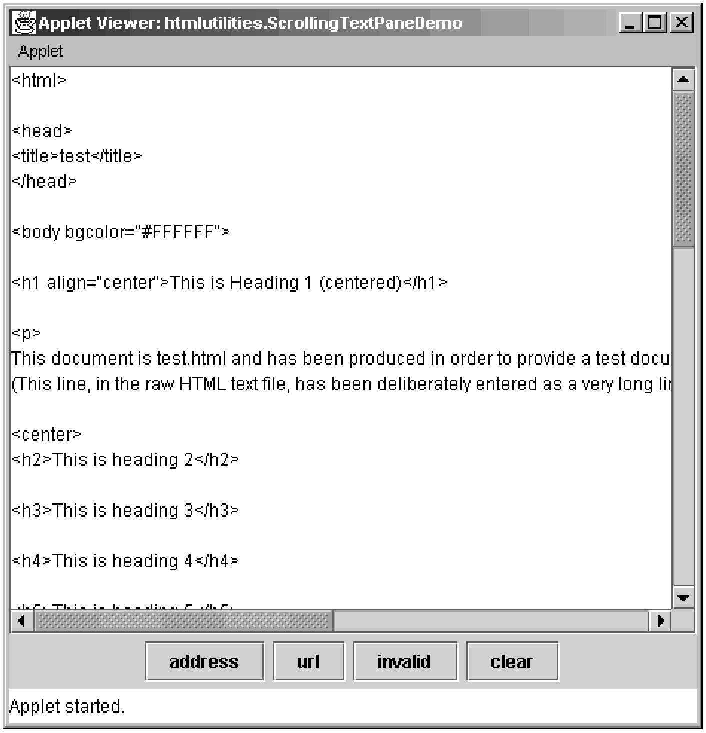
```
   public void setAddress( String newAddress)
                      throws BrewserException {
       this.obtainRawHTML( newAddress);
       textPage.setText( rawHTML);
   } // End setAddress.
```

The *clearHTML()* and *setText()*methods are implemented as follows.

```
   public void clearHTML() {
      address  = null;
      rawHTML  = new String( "");
      location = null;
      textPage.setText( rawHTML);
   } // End clearHTML.

   public void setText( String newText) {
      address  = new String( "Unknown");
      location = null;
      rawHTML  = new String( newText);
      textPage.setText( rawHTML);
   } // End setText.
```

Both of these methods set the state of the four attributes as appropriate. For the *clearHTML()* method this effectively empties all attributes. For the *setText()* method this installs the *newText* supplied and indicates that its Internet *address* is '*Unknown*'.

Figure 8.7 shows an instance of the *ScrollingTextPane* within a demonstration harness. The *address* button, at the bottom of the artifact, is connected to the *ScrollingTextPane*'s *setAddress()* method; likewise the *url* button is connected to its *setURL()* method. The



```
<html>

<head>
<title>test</title>
</head>

<body bgcolor="#FFFFFF">

<h1 align="center">This is Heading 1 (centered)</h1>

<p>
This document is test.html and has been produced in order to provide a test docu
(This line, in the raw HTML text file, has been deliberately entered as a very long li

<center>
<h2>This is heading 2</h2>

<h3>This is heading 3</h3>

<h4>This is heading 4</h4>

<h5>This is heading 5</h5>
```

Applet started.

Figure 8.7 *ScrollingTextPane* within a demonstration harness.

invalid button is also connected to the *setAddress()* method, but passes a String argument that cannot be resolved as a valid URL and the *clear* button is connected to the *clearHTML()* method. The implementation of this demonstration harness will be left as an end of chapter exercise.

> *To facilitate testing, a URL can be expressed, for example, as* `file:/brewser/test1.html`*. That is using* `file` *as the first term instead of* `http` *(hypertext transport protocol). This is followed by the full path on the local disk to the HTML document. This will cause the connection to be made directly to the file, obviating the need to have an available Internet connection.*
>
> *Even when a connection is being made to the local disk, Java's security mechanism may object and throw a SecurityException. To prevent this from happening the Java environment's security settings may have to be investigated and set as appropriate.*

8.5 The *ScrollingHTMLPane* class

The *ScrollingHTMLPane* class is very similar to the *ScrollingTextPane* class. The major difference is that an instance of JEditorPane is installed into the viewport, and this has consequential changes upon the implementation of the other **public** methods. The

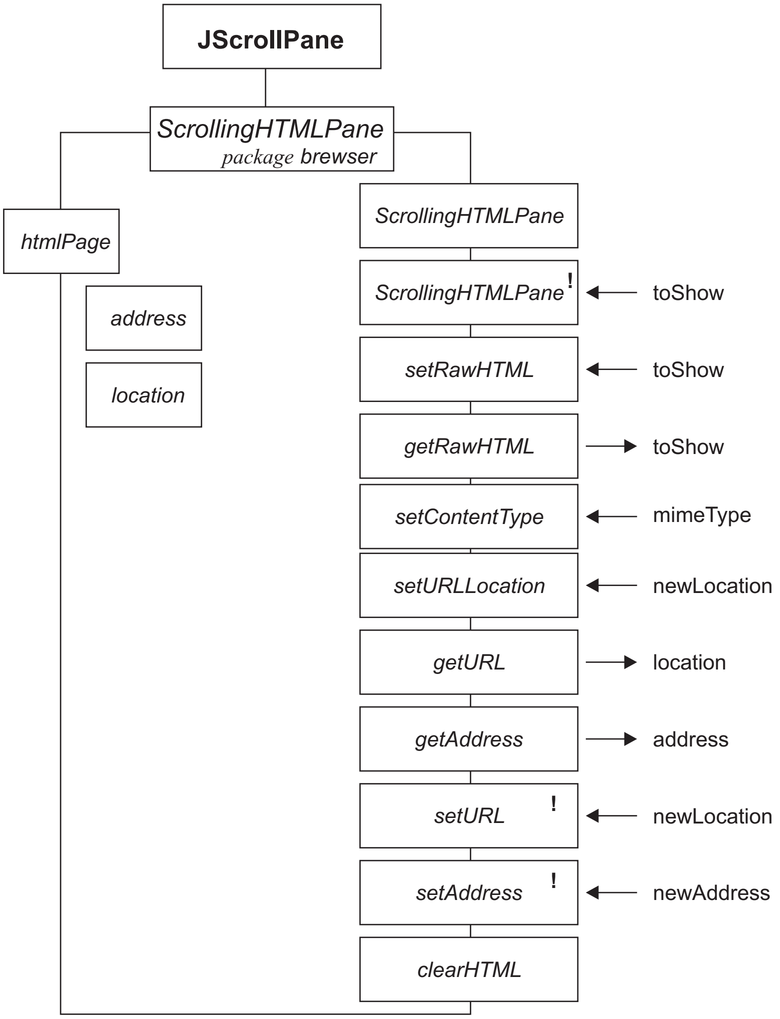


Figure 8.8 *ScrollingHTMLPane*: class diagram.

ScrollingHTMLPane class diagram is given in Figure 8.8. This component does not require an equivalent of the *ScrollingTextPane rawHTML* attribute, nor the *obtainRawHTML()* method, as the JEditorPane instance, *htmlPage*, is capable of obtaining this information from the Internet. The other two attributes are as before.

The default constructor will prepare an empty component without throwing an exception. The alternative constructor requires the Internet address of the information *toShow* and will throw an exception if it cannot be obtained. The *setRawHTML()* and *getRawHTML()* allow the HTML contents of the page to be set or obtained. The *setURLLocation()* method allows the *location* and *address* attributes to be changed without attempting to load the resource they identify; it is intended to be used when the contents of the component are set with the *setRawHTML()* method. As the type contents of the component could be changed by use of the *setRawHTML()* method, for example from HTML to RTF, a *setContentType()* method is required to inform the component of this.

The *getURL()* and *getAddress()* methods are simple inquiry methods. The *setURL()* and *setAddress()* methods attempt to load the resource identified by their argument and throw an exception if this is not possible. Finally, the *clearHTML()* method clears the *htmlPage* and sets the other two attributes to indicate this. The implementation of this class commences as follows.

```
package brewser;

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

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


public class ScrollingHTMLPane extends JScrollPane {

protected JEditorPane htmlPage = null;
private   String      address  = null;
private   URL         location = null;
```

These declarations are comparable with those of the *ScrollingTextPane* class as explained above, apart from the omission of the *rawHTML* attribute; the reasons for this will be explained shortly. The default constructor, as follows, is not noted as possibly throwing any exceptions as it will construct an instance which is known not to have any contents. It is implemented by indirecting to the alternative constructor, passing an empty string as the address to attempt to obtain the HTML contents from.

```
  public ScrollingHTMLPane() {
     this( "");
  } // End ScrollingHTMLPane default constructor.

  public ScrollingHTMLPane( String toShow)
                       throws BrewserException {

      super( JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
             JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

       htmlPage = new JEditorPane();
       htmlPage.setEditable( false);
       htmlPage.setContentType( “text/html”);
       this.setMinimumSize( new Dimension(100, 50));
       this.setMaximumSize( new Dimension(200, 200));
       this.getViewport().add( htmlPage);
       location = null;
       address  = new String("");

       if ( toShow.length() > 0) {
```

```
0047            try {
0048               htmlPage.setPage( toShow);
0049               location = htmlPage.getPage();
0050               address  = htmlPage.getPage().toString();
0051            } catch ( IOException exception) {
0052               throw new BrewserException();
0053            } // End try/catch
0054         } // End if.
0055      } // end ScrollingHTMLPane constructor.
```

The alternative constructor is noted as possibly throwing a *BrewserException*. The exception will be thrown if the HTML page at the address *toShow* cannot be opened and will indicate to the calling environment that the instance was not constructed, as explained for the *ScrollingTextPane* constructors above. However, the exception will not be thrown if *toShow* is an empty string, as this indicates that the alternative constructor was called from the default constructor.

The first stage of the constructor on lines 0034 and 0035, calling the **super** constructor and establishing the scroll bar policies, is essentially identical to the first stage of the *ScrollingTextPane* constructor. However, the rules for the rendering of an HTML document state that it must lay itself out within the available width of the window, so no horizontal scroll bar is ever required; this is indicated on line 0035.

The next part of the constructor, on lines 0037 to 0044, constructs, configures and installs into the viewport an empty JEditorPane instance, prepared to render HTML, referenced by *htmlPage*. As with the *ScrollingTextPage* constructor, the final stage, which attempts to obtain the contents to show in the component, is guarded on line 0046 by a test that ensures that a non-empty address has been supplied in the *toShow* argument. This will make sure that no exception is propagated via the default constructor.

If the address is non-empty, line 0048 calls the *htmlPage* setPage() method, passing on the *toShow* argument of the constructor. This method will attempt, partially asynchronously, to open a connection to the Internet address specified in its argument, obtain the contents, install an appropriate ViewFactory and render the document obtained. If this process is initiated successfully no IOException will be thrown and, on lines 0049 and 0050, the *address* and the *location* are initialized via calls of the *getPage()* method. This JEditorPane capability avoids the need to supply a method comparable to the *obtainRawHTML()* method of the *ScrollingTextPane* class as the component already knows how to interact with the Internet.

If the call of *setPage()* does throw an IOException it indicates that either the URL was not valid or there was a problem in obtaining the contents of the resource it identifies. No MalformedURLException is thrown to indicate a bad URL, any problems being reported as an IOException. The exception handler, on line 0052, propagates any exception from the constructor as an instance of the *BrewserException* class.

JComponent → JTextComponent → javax.swing.JEditorPane

```
public JEditorPane()
public JEditorPane ( String urlSpec)  throws IOException
public JEditorPane ( URL    location) throws IOException
```

Constructs a JEditorPane instance containing the resource identified in the argument, automatically installing the appropriate Document and ViewFactory components.

```
public URL  getPage()
public void setPage( URL newPage) throws IOException
```

Obtains the URL of the page being displayed or installs a new page, automatically installing a different Document and ViewFactory if required.

```
public void addHyperlinkListener(    HyperlinkListener listener)
public void removeHyperlinkListener( HyperlinkListener listener)
```

If the component is non-editable and rendering HTML a HyperlinkEvent will be dispatched to any registered listeners when the user activates it.

```
public String getContentType()
```

Identifies the MIME (*M*ulti-purpose *I*nternet *M*ail *E*xtensions) type of the contents e.g 'text/plain' or 'text/html'.

```
public Document getDocument()
```

Obtains the Document model of the content.

The *setRawHTML()* and *getRawHTML()* methods are implemented as follows.

```
0057    public void setRawHTML( String newText) {
0058       htmlPage.setText( newText);
0059       address  = new String( "Unknown");
0060       location = null;
0061    } // End setRawHTML.
0062
0063     public String getRawHTML() {
0064        return htmlPage.getText();
0065    } // End getRawHTML.
```

The *setRawHTML()* method commences, on line 0058, by calling the inherited *htmlPage* setText() method to install its argument, *newText*, into the component. As the origin of this HTML page is unknown the *address* and *location* attributes are set as appropriate, on lines 0059 and 0060, to complete the method. The *getRawHTML()* method is implemented as a wrapper on the inherited *htmlPage* getText() method.

Should the location of the contents that have been supplied by a call of *setRawHTML()* be known, the *setURLLocation()* method can be used to establish the values of the *location* and *address* attributes, as follows.

```
    public void setURLLocation( URL newLocation) {
       location = newLocation;
       address  = location.toString();
    } // End setURLLocation.

    public URL getURL() {
       return location;
    } // End getURL.

    public String getAddress() {
       return address;
    } // End getAddress.

```

```
0079    public void setContentType( String mimeType) {
0080       htmlPage.setContentType( mimeType);
0081    } // End setContentType.
```

This fragment includes the *getURL()* and *getAddress()* inquiry methods and also the *setContentType()* method. The last method takes an argument which is expected to indicate one of the supported MIME (*M*ultipurpose *I*nternet *M*ail *E*xtension) types, which are "*text/html*" and "*text/rtf*". The method is implemented as a wrapper on the *htmlPage* setContentType() method passing on the argument.

If the *mimeType* argument passed to the *htmlPage* setContentType() method identifies the same *contentType* which is already being rendered, the state of the component will not change. If the argument identifies a supported *contentType* which is not currently being rendered by the *htmlPage* a new Document model and ViewFactory, as shown in Figure 8.3, are installed into the component. This will cause the interpretation of the plain text contents to be reinterpreted and rendered as appropriate. Otherwise, if the argument identifies an unsupported *contentType* the state of the component will not change and the results of passing the plain text through the Document model and ViewFactory will be unpredictable.

The implementation of the *scrollingHTMLPane* class continues with the *setURL()* and *setAddress()* methods. Either of these can be used to install a new document obtained from the Internet into the component, so both are noted as throwing a *BrewserException*.

```
0084     public void setURL( URL newLocation)
0085                         throws  BrewserException {
0086        this.setAddress( newLocation.toString());
0087     } // End setURL.
0088
0089     public void setAddress( String newAddress)
0090                          throws BrewserException {
0091        try {
0092           htmlPage.setPage( newAddress);
0093           location = htmlPage.getPage();
0094           address  = location.toString();
0095        } catch ( IOException exception ) {
0096           throw new BrewserException();
0097        } // End try/catch.
0098     } // End setAddress.
```

The *setURL()* method indirects to the *setAddress()* method, passing on its URL argument as a String. The *setAddress()* method attempts to retrieve the document by calling the *htmlPage* setPage() method. If this method does not throw an IOException then the HTML document identified by its argument will be obtained and rendered in the *htmlPage* window and lines 0093 and 0094 will update the other two attributes. If the setPage() method does throw an exception the contents of the component will not be changed, so the values of the two attributes are left unchanged in the exception handler on line 0096, which propagates a *BrewserException*.

The implementation of the *ScrollingHTMLPane* concludes with the *clearHTML()* method, as follows.

```
0101    public void clearHTML() throws BrewserException {
0102       try {
```

```
0103          htmlPage.getDocument().remove( 0,
0104                          htmlPage.getDocument().getLength());
0105       } catch ( BadLocationException exception) {
0106          throw new BrewserException();
0107       } // End try/catch.
0108       address  = new String( "");
0109       location = null;
0110    } // End clearHTML.
0111
0112 } // End ScrollingHTMLPane class
```

In order to clear the contents of the *htmlPage*, the contents of its Document have to be removed. This is accomplished on lines 0092 and 0093 by using the *htmlPage* getDocument() method to obtain the identity of its Document. The Document remove() method is called to effect the clearance, passing as arguments the start of the document's range (0) and the extent of the document, established by calling its *getLength()* method. This change to the contents of the Document will result in the *htmlPage*'s window being cleared, as indicated in Figure 8.3. The *remove()* method may throw a BadLocationException, so this fragment is contained within a **try/catch** structure which propagates it as a *BrewserException*. Having cleared the contents of the *htmlPage*, the other two attributes are cleared on lines 0097 and 0098.

javax.swing.text.Document: interface

The Document interface defines the protocol for the Document model contained within every JTextComponent. It can view the text contents either as a sequence of **char**acters or as a sequence of Elements. An Element is a sequence of **char**acters (possibly empty) and associated Attribute information (e.g. font and style).

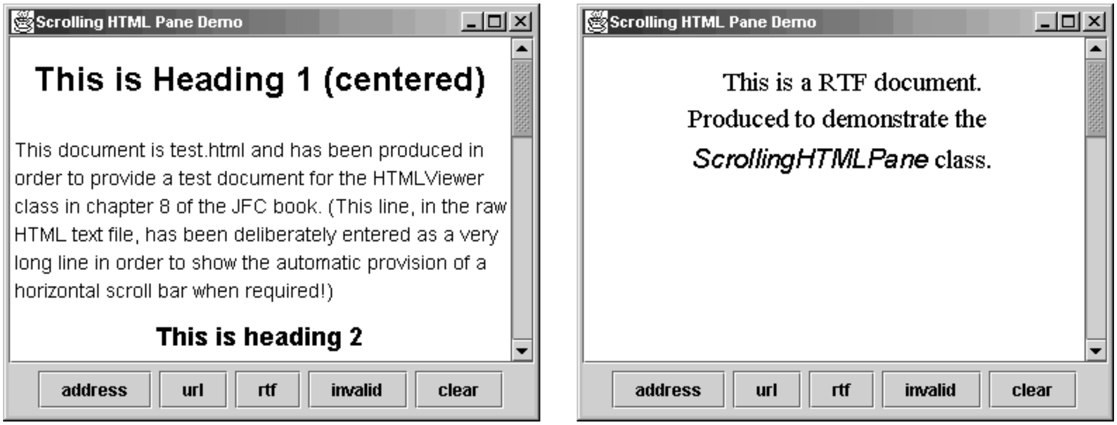
```
public void addDocumentListener(    DocumentListener listener)
public void removeDocumentListener( DocumentListener listener)
```

A DocumentEvent is created and dispatched to any registered listeners every time the contents of the Document are changed.

```
public void   getText( int offset, int length, Segment obtained)
                                        throws BadLocationException
public String getText( int offset, int length)
                                        throws BadLocationException
public void   remove( int offset, int length)
                                        throws BadLocationException
public void   insertString( int offset, String toInsert,
                             AttributeSet attributes)
                                        throws BadLocationException
```

Methods to obtain or manipulate the contents of the Document.

Figure 8.9 shows an instance of the *ScrollingHTMLPane* class within a demonstration test harness. The left-hand image shows the same document as was shown in Figure 8.4, illustrating again the automatic rendering of raw HTML by a ViewFactory. The right-hand illustration shows the instance rendering an RTF document, obtained by specifying a `*.rtf` file in the URL instead of a `*.html` file. The JEditorPane has automatically



This is Heading 1 (centered)

This document is test.html and has been produced in order to provide a test document for the HTMLViewer class in chapter 8 of the JFC book. (This line, in the raw HTML text file, has been deliberately entered as a very long line in order to show the automatic provision of a horizontal scroll bar when required!)

This is heading 2

This is a RTF document.
Produced to demonstrate the
ScrollingHTMLPane class.

Figure 8.9 *ScrollingHTMLPane* within a demonstration harness.

replaced the HTML ViewFactory with an RTF ViewFactory in order to render the file when it detected the type of the document during execution of its setPage() method.

8.6 The *HTMLViewer* class

The *HTMLViewer* class, as shown in Figure 8.4, is an extended JSplitPane instance containing a *ScrollingHTMLPane* and a *ScrollingTextPane* instance. In addition, the two vertical JScrollBars can be tied together so that moving one of them will cause the other, and the portion of the viewport it controls, to be adjusted. It also has the capability to hide or show the raw text pane. The *HTMLViewer* class diagram is given in Figure 8.10.

The single constructor requires an Internet location *toShow* as an argument and will attempt to retrieve and render the contents. If the resource *toShow* cannot be obtained the component will be initialized with an empty HTML document and the constructing client should test the value of the *address* or *URL* attribute to find out if this was the case. The first two attributes are the *ScrollingTextPane* and *ScrollingHTMLPane* instances contained within the component and the next two attributes will be used to store the identity of the two vertical scroll bars contained within them. The final attributes are two flags that determine whether the two panes are *tied* together and whether the raw HTML pane is to be shown or hidden.

The *setURL()* and *setAddress()* methods allow a different location to be specified and may throw a *BrewserException*. These two methods are complemented by *getURL()* and *getAddress()*, which obtain their values from one of the encapsulated scrolling components, as will be explained.

The next six methods are associated with the two flags, allowing their values to be set or inquired. The next method, *clearHTML()*, removes the contents of both scroll panes. The final method, *adjustmentValueChanged()* is mandated by the AdjustmentListener interface. The class must implement the AdjustmentListener interface, as instances of it, when the two panes are *tied* together, have to respond to the AdjustmentEvents generated when the user interacts with one of the vertical scroll bars.

The implementation of the *HTMLViewer* class, as far as the end of its attribute declarations, is as follows.

```
package brewser;
```

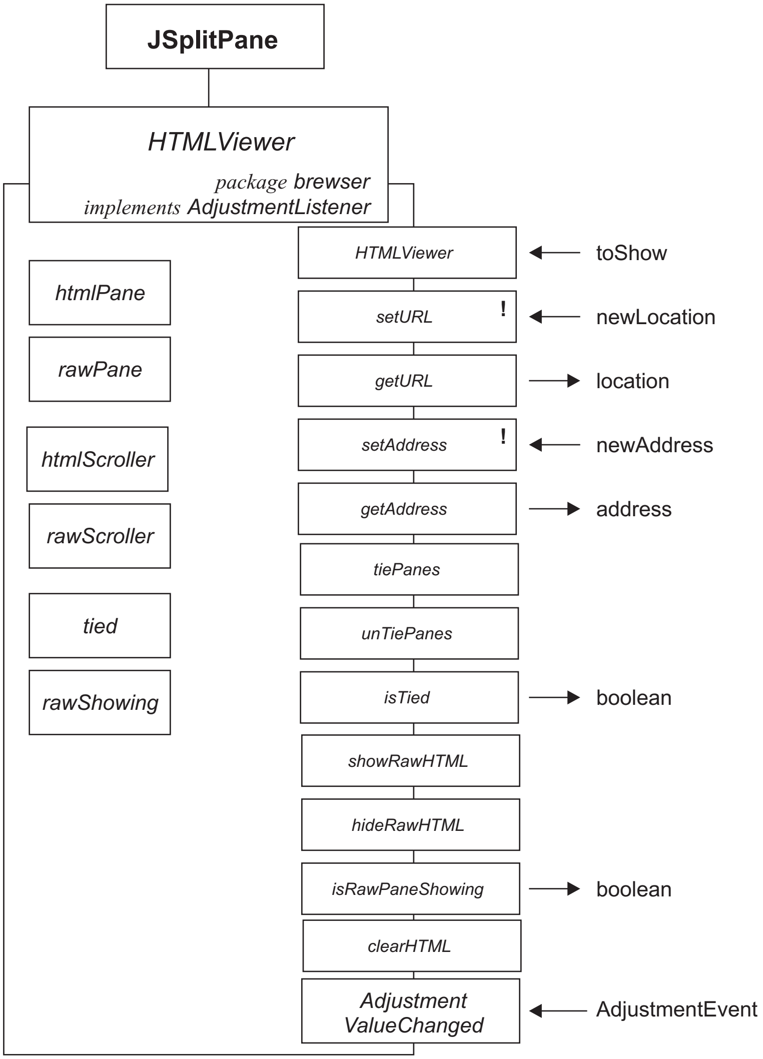


Figure 8.10 *HTMLViewer* class diagram.

```

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

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

public class HTMLViewer extends JSplitPane
                        implements AdjustmentListener {

private ScrollingHTMLPane htmlPane     = null;
```

```
0024  private JScrollBar         htmlScroller = null;
0025  private ScrollingTextPane  rawPane      = null;
0026  private JScrollBar         rawScroller  = null;
0027
0028  private boolean linked      = true;
0029  private boolean rawShowing  = true;
```

When the user interacts with the JScrollBar instances contained within the two extended JScrollPanes, AdjustmentEvents are generated. In order for the *HTMLViewer* class to be able to tie the two viewports together it will have to listen to these events and accordingly the *HTMLViewer* class is declared as implementing the AdjustmentListener interface. The first four private attributes, declared on lines 0023 to 0026, are the two panes themselves and the identities of their two vertical scroll bars. The two remaining **boolean** attributes determine whether the two panes are *tied* together and if the raw text pane is to be shown. The implementation of the constructor is as follows.

```
0033        public HTMLViewer( String toShow) throws
0034                                   BrewserException {
0035
0036        super( JSplitPane.VERTICAL_SPLIT);
0037
0038        rawPane  = new ScrollingTextPane();
0039        try {
0040           rawPane.setAddress( toShow);
0041        } catch ( BrewserException exception) {
0042           // Do Nothing
0043        } // End try/catch.
0044
0045        htmlPane = new ScrollingHTMLPane();
0046        htmlPane.setRawText( rawPane.getRawHtml());
0047        htmlPane.setURL( rawPane.getURLLocation());
0048
0049        htmlScroller = htmlPane.getVerticalScrollBar();
0050        htmlScroller.setName( "html");
0051        htmlScroller.addAdjustmentListener( this);
0052
0053        rawScroller = rawPane.getVerticalScrollBar();
0054        rawScroller.setName( "raw");
0055        rawScroller.addAdjustmentListener( this);
0056
0057        this.setTopComponent(    htmlPane);
0058        this.setBottomComponent(  rawPane);
0059        this.setDividerLocation( 0.5);
0060
0061     } // End HTMLViewer constructor.
```

The constructor takes a single String argument indicating the document *toShow* in each of the panes. Its first step, on line 0036, is to call its **super**, JSplitPane, constructor specifying a vertical split which will result in a horizontal bar. It then constructs the *ScrollingTextPane* instance, *rawPane*, using the default constructor, which is guaranteed not to throw an exception. Once the *rawPane* is constructed, on lines 0039 to 0043, an attempt is made to install the document identified by the *toShow* argument. If the URL indicated by the *toShow* argument does not exist, or if its contents cannot be retrieved, a *BrewserException* will be thrown and consumed by the empty handler on line 0042. Hence, on line 0044, it is guaranteed that the *rawPane* exists, even if it is empty.

JComponent → javax.swing.JSplitPane

```
public JSplitPane()
public JSplitPane( int orientation)
public JSplitPane( int orientation, Component left, Component right)
```

Constructors allowing the *orientation* of the split (HORIZONTAL_SPLIT (default) or VERTICAL_SPLIT) and the two Components to be specified. A HORIZONTAL_SPLIT has a vertical bar. The left Component is positioned at the top in a VERTICAL_SPLIT.

```
public void      setTopComponent( Component newTop)
public Component getTopComponent()
public void      setBottomComponent( Component newTop)
public Component getBottomComponent()
```

Setting and inquiry methods for the two components.

```
public int  getDividerLocation()
public void setDividerLocation ( int    location)
public void setDividerLocation ( double proportion)
public void setDividerSize(      int    newSize)
```

Methods to obtain and set the divider location bar. The **int** arguments are component-specific (e.g. lines for a text component or pixels for an image). The size is always in pixels.

```
public boolean isContinuouslayout()
public void    setContinuouslayout (boolean yesOrNo)
```

The continuousLayout attribute determines whether the view is continually updates as the user moves the divider (**true**) or not; default **true**.

Having constructed the *ScrollingTextPane*, lines 0045 to 0047 construct and initialize the *ScrollingHTMLPane* instance. The pane is constructed using its default constructor, which will prepare it for rendering HTML but will not supply it with any contents. After construction, on line 0046, it is supplied with its contents by retrieving the contents of the *rawPane*. The previous construction of the *ScrollingTextPane* will have blocked the thread of control while the document was obtained across the Internet. Rather than have to do this again, which would actually take place on a separate thread of control and lead to having to synchronize the constructor with this process, the contents of the *rawPane* can be easily obtained and supplied to the *htmlPane*. As the *htmlPane* now contains the contents of a Web resource this step concludes by setting its *URL* attribute from the *URL* attribute of the *rawPane*.

On line 0049 the *htmlScroller*'s getVerticalScrollBar() method, inherited from JScrollPane, is called to obtain the identity of its vertical scroll bar, and this identity is referenced by the *htmlScroller* attribute. Having obtained its identity the setName() method, inherited from AWT Component, is used to establish a name for it. This action is needed in order that the listener method can decide which of the two scroll bars dispatched an event. Finally, in this part of the constructor, **this** instance of *HTMLViewer* is registered as the JScrollBar's AdjustmentListener.

On lines 0053 to 0055 a corresponding sequence of actions is used to obtain, and configure, the identity of the *rawPane*'s vertical scroll bar. The final part of the HTMLViewer constructor, on lines 0057 to 0058, uses the JSplitPane setTopComponent() and setBottomComponent() methods to install the two panes into itself. The constructor then concludes by setting the dividing bar halfway down the window, using the setDividerLocation() method.

The effect of this constructor is to produce the physical appearance shown in Figure 8.4 and cause AdjustmentEvents to be dispatched to the *HTMLViewer*'s *adjustment ValueChanged()* method, which is mandated by the AdjustmentListener interface, when the user adjusts one of the vertical scroll bars. The first part of the *adjustment ValueChanged()* method is implemented as follows.

```
0150     public void adjustmentValueChanged( AdjustmentEvent event){
0151
0152     double movedTop     = 0.0;
0153     double movedBottom  = 0.0;
0154     double movedExtent  = 0.0;
0155     double movedMaximum = 0.0;
0156
0157     double respondingTop     = 0.0;
0158     double respondingBottom  = 0.0;
0159     double respondingExtent  = 0.0;
0160     double respondingMaximum = 0.0;
0161
0162     int    newValue = 0;
0163
0164     if( tied) {
0165        if (((Component) event.getSource()).
0166                              getName().equals( "html") ) {
0167
0168           movedMaximum  = (double) htmlScroller.getMaximum();
0169           movedTop      = htmlScroller.getValue() /
0170                                                  movedMaximum;
0171           movedExtent   = htmlScroller.getVisibleAmount() /
0172                                                  movedMaximum;
0173
0174           respondingMaximum = (double) rawScroller.getMaximum();
0175           respondingTop     = rawScroller.getValue() /
0176                                                respondingMaximum;
0177           respondingExtent  = rawScroller.getVisibleAmount() /
0178                                                respondingMaximum;
0179           respondingBottom  = respondingTop + respondingExtent;
0178
0179           if ( ( movedTop < respondingTop)    ||
0180                ( movedTop > respondingBottom) ){
0181              newValue = (int) ((movedTop + (movedExtent / 2.0))
0182                                             * respondingMaximum);
0183              rawScroller.setValue( newValue);
0184           } // End if.
```

```
0185       } else {
```

The method commences, on line 0164, with a guard which causes the rest of the method to do nothing if the value of the *tied* attribute indicates that the two panes are not to be linked together. The remaining part of the fragment is only concerned with responding to movements of the `html` scroll bar, as identified on lines 0165 and 0166. The source attribute of any Event contains the identity of the Object that dispatched the event. In this situation it is certain that the source is a Component and so the cast, on line 0165, will always be valid. Having obtained a Component identity, the AWT Component getName() method can be used to obtain the name of the component, as established in the *HTMLViewer* constructor.

Having established that the *event* originated from the *htmlScroller*, within the **if** structure commencing on line 0165, various attributes of the two JScrollBar instances are retrieved and used to initialize the values of various local **double** variables. Those variables whose names start with *moved* are initialized from the *htmlScroller* and those starting with *responding* are initialized from the *rawScroller*. Figure 8.11 shows on the left the meaning of various attributes of a *JScrollBar* and on the right the meaning of the variables calculated from the attributes. The attributes are provided as values related to the value of the Scrollbar's maximum attribute, and these are normalized into values between 0.0 and 1.0 in the local variables.

Once the variables have been initialized from the attributes the **if** decision on lines 0179 and 0180 establishes whether the *top* location of the *moved* scroll bar is above the corresponding *top* position, or below the corresponding *bottom* position, of the *responding* bar. If so, a *newValue* for the responding, *rawScroller*, scroll bar is computed, on lines 0181 and 0182, and installed into the bar by calling its *setValue()* method on line

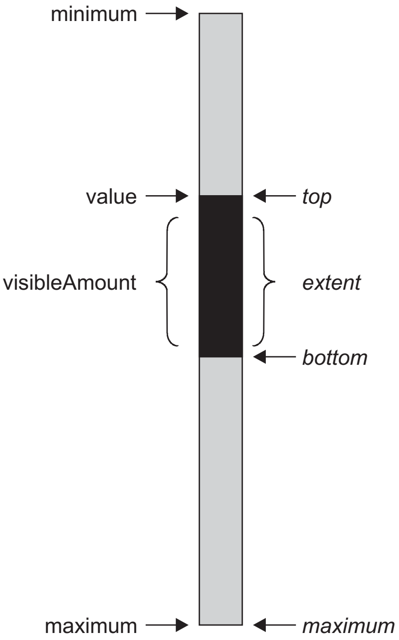


Figure 8.11 *JScrollBar* attributes (left) and *HTMLViewer adjustmentValueChanged()* local variables (right).

0183. Should the **if** decision indicate that the corresponding part of the *responding* bar is already visible then no message is sent to it. The effect is that when the `html` scroll bar is moved the `raw` scroll bar's slider will move, if required, to make sure that it is in a corresponding location. When the value of the JScrollBar is changed by having its setValue() method called, the *JScrollPane*'s viewport will also be adjusted in concert.

The second part of the *adjustmentValueChanged()* method is essentially identical, but obtains its *moved* values from the *rawScroller* and its *responding* values from the *htmlScroller*, and possibly calls the setValue() method of the *htmlScroller*. As it is so similar it will not be presented.

This implementation assumes that the raw HTML image in the *ScrollingTextPane* and the rendered HTML image in the *ScrollingHTMLPane* are essentially congruent. This might not be the case, so it is not guaranteed that the raw HTML visible in the bottom pane will always contain the code that produced the image currently visible in the upper pane.

> *When sending messages between linked components, such as these two scroll bars, some care should be exercised. When the user moves one of the bars an* AdjustmentEvent *is sent to the listener. If the receipt of this event always resulted in a* setValue() *message being sent to the other bar then not only would the position of its slider change, but an* AdjustmentEvent *would be generated and sent back to the listener. This would also cause an adjustment to the first bar, causing an event to be sent, which would cause the second bar to adjust, and so on. The consequence would be that both bars would creep towards one end, and messages would continue to be sent ad infinitum. To avoid this unwanted consequence there should be some mechanism, as in this example, that prevents events always being echoed upon receipt.*

The tying together of the two panes is controlled by the value of the *tied* attribute, which is supported by the following methods.

```
0112    public void tiePanes() {
0113       tied = true;
0114    } // End tiePanes.
0115
0116    public void unTiePanes() {
0117       tied = false;
0118    } // End unTiePanes.
0119
0120    public boolean isTied() {
0121       return tied;
0122    } // End isTied.
```

The *HTMLViewer* also supplies the capability to hide the *rawPane*, controlled by the value of the *rawShowing* attribute and supported by the following methods.

```
0125    public void showRawHTML() {
0126       if ( !rawShowing) {
0127          this.setBottomComponent(  rawPane);
0128          this.setDividerLocation( 0.5);
0129       } // End if.
```

```
      rawShowing = true;
   } // End showRawHTML.

   public void hideRawHTML() {
      if ( rawShowing) {
         this.remove( rawPane);
         this.doLayout();
      } // End if.
      rawShowing = false;
      tied       = false;
   } // End hideRawHTML.

   public boolean isRawShowing() {
      return rawShowing;
   } // End isRawShowing.
```

The *rawPane* is hidden and shown by using this *JSplitPane*'s *setBottomComponent()* and *remove()* methods, accompanied by other appropriate supporting steps. The remaining *HTMLViewer* methods are concerned with the address and URL of the document which is being displayed. The *HTMLViewer* class does not contain these as distinct attributes but wraps these methods onto the corresponding attributes contained within the *rawPane* and *htmlPane* components that it contains.

```
   public void setURL( URL newLocation)
                               throws BrewserException {
     this.setAddress( newLocation.toString());
  } // End setURL.

   public URL getURL() {
      return rawPane.getURL();
   } // End getURL.

   public void setAddress( String newAddress)
                               throws BrewserException {

   URL     newLocation = null;
   int     lastDot       = 0;
   char    mimeIndicator = ' ';
   boolean isRTF         = false;

     lastDot       = newAddress.lastIndexOf( ".");
     mimeIndicator = newAddress.charAt( lastDot +1);
     isRTF         = mimeIndicator == 'r';

      try {
         newLocation = new URL( newAddress);
         rawPane.setURL( newLocation);
         if ( isRTF) {
            htmlPane.setContentType( "text/rtf");
         } else {
            htmlPane.setContentType( "text/html");
         } // End if.
```

```
0097          htmlPane.setRawHTML( rawPane.getRawHTML());
0098          htmlPane.setURL( rawPane.getURL());
0099       } catch (MalformedURLException exception) {
0100          throw new BrewserException(
0101                   BrewserException.BAD_URL);
0102       } catch (IOException exception) {
0103          throw new BrewserException(
0104                   BrewserException.LOAD_FAULT);
0105       } // End try/catch.
0106    } // End setAddress
0107
0108    public String getAddress() {
0109       return rawPane.getAddress();
0110    } // End getAddress.
```

The *getURL()* and *getAddress()* methods are mapped onto the corresponding *rawPane* methods. The *setURL()* method indirects to the *setAddress()* method which will attempt to load the resources from the *newAddress* supplied, throwing a *BrewserException* if this cannot be accomplished.

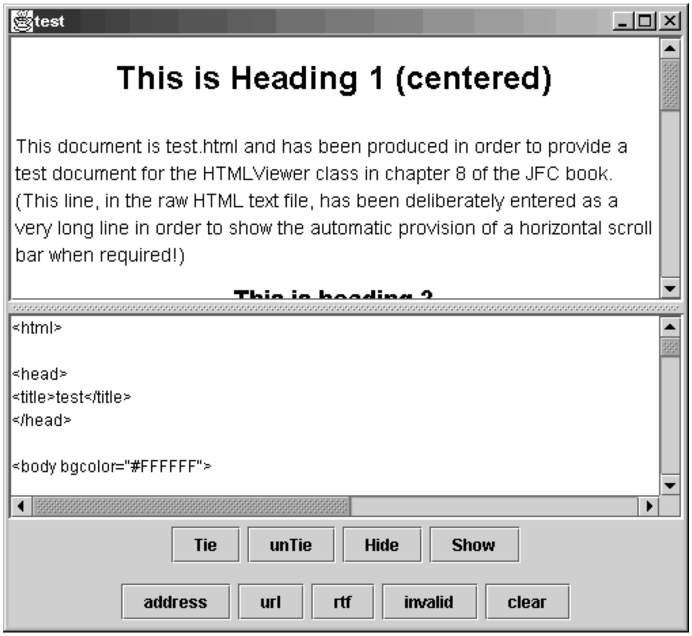
The substantive part of the *setAddress()* method commences, on line 0090, by attempting to construct a local URL instance, called *newLocation*, from the *newAddress* argument. This step may throw a MalformedURLException and so is contained within a **try/catch** structure. If the exception is not thrown an attempt is made to load the page using the *rawPane*'s *setURL()* method. If this method does not throw an exception, indicating that the page could not be loaded, on line 0097 the contents of the *rawPane* are obtained and supplied to the *htmlPane* to be rendered and the **try** steps conclude with setting the *htmlPane*'s URL attribute to that of the *rawPane*. Should any exceptions be thrown during this attempt the contents of the panes will not be changed and a *BrewserException*, with an appropriate reason, will be propagated from the method.

This is complicated by the possibility of the content type of the document obtained changing. To address this possibility the three local variables, declared on lines 0081 to 0083, and the initial steps on lines 0085 to 0087 parse the *newAddress* String. The method is assuming that the address will terminate with `~~.html` or `~~.htm` if it is an HTML document or `~~.rtf` if it is an RTF document. The *mimeIndicator* variable is initialized, on line 0086, to the character which follows the *lastDot* in the *newAddress* and the **boolean** flag *isRTF* set as appropriate.

Within the **try** structure, after the contents of the *newAddress* have been retrieved and installed into the *rawPane*, on lines 0092 to 0096, the *setContentType()* method of the *htmlPane* is called with an appropriate argument, before the contents of the *rawPane* are copied to the *htmlPane*, on line 0097. This will attempt to ensure that the JEditorPane contained within the *htmlPane* has the appropriate Document and ViewFactory installed before it is supplied with the plain text to interpret and render.

The only method remaining is *clearHTML()*, implemented without comment as follows.

```
0146    public void clearHTML() {
0147       rawpane.clearHTML();
0148       htmlPane.clearHTML();
0149    } // End clearHTML.
```



This is Heading 1 (centered)

This document is test.html and has been produced in order to provide a test document for the HTMLViewer class in chapter 8 of the JFC book. (This line, in the raw HTML text file, has been deliberately entered as a very long line in order to show the automatic provision of a horizontal scroll bar when required!)

This is heading 2

```
<html>

<head>
<title>test</title>
</head>

<body bgcolor="#FFFFFF">
```

Figure 8.12 *HTMLViewer* within a demonstration harness.

A demonstration test harness for the *HTMLViewer* class is illustrated in Figure 8.12. It contains two rows of buttons at the bottom of the window. The upper row allows the tying together and the hiding of the *rawPane* to be demonstrated. The lower row repeats the demonstrations supplied for the *scrollingHTMLPane*, as shown in Figure 8.9.

8.7 The *NavigatingHTMLPane* – a primitive browser

In this part of the chapter the *ScrollingHTMLPane* class will be extended to provide the *NavigatingHTMLPane* class that is capable of following hyperlinks within an HTML document, providing primitive HTML browser functionality. When a JEditorPane contains an HTML document that contains *hyperlinks*, the links will be rendered using what has become a conventional blue color to indicate their existence to the user. When the user clicks upon one of the links in a non-editable JEditorPane it will dispatch a HyperLink event, containing the URL of the hyperlink's target, to any registered listeners.

The *NavigatingHTMLPane* class overrides the *ScrollingHTMLPane* alternative constructor and adds two methods called *addHyperlinkListener()* and *removeHyperlink Listener()*, both of which take an argument of the HyperlinkListener interface class. The implementation of these methods, as follows, passes the argument onto the corresponding methods of the *htmlPane* JEditorPane instance, declared with **protected** visibility in the *ScrollingHTMLPane* class.

```
public class NavigatingHTMLPane extends ScrollingHTMLPane {

  public NavigatingHTMLPane( String toShow)
                                 throws  BrewserException {
     super( toShow);
  } // End NavigatingHTMLPane constructor.

  public void addHyperlinkListener(
                  HyperlinkListener listener) {
     htmlPage.addHyperlinkListener( listener);
  } // End addHyperlinkListener.

  public void removeHyperlinkListener(
                  HyperlinkListener listener) {
     htmlPage.removeHyperlinkListener( listener);
  } // End removeHyperlinkListener.

} // End class NavigatingHTMLPane.
```

javax.swing.event.HyperlinkListener: interface

The interface that mandates the resources for a HyperlinkListener, capable of receiving HyperlinkEvents, from a JEditorPane.

```
public void hyperlinkUpdate( HyperlinkEvent event)
```

Single method required, called when the user activates a hyperlink.

The *HyperlinkListener* interface mandates a method called hyperlinkUpdate() taking a single argument of the HyperlinkEvent class. The implementation of this method within the *NavigatingHTMLPaneDemo* class is as follows.

```
  public void hyperlinkUpdate( HyperlinkEvent event) {
     try {
        navigatingPane.setURLLocation( event.getURL());
     } catch ( BrewserException exception) {
        System.out.println("Link not reached " +
                                  event.getURL());
     } // End try/catch.
  } // End hyperlinkUpdate.
```

The listener method responds to the receipt of a HyperlinkEvent by using its getURL() method to extract the URL of the hyperlink's target. This information is then passed to the *navigatingPane* so that it can update its contents. The effect is that when the user clicks upon a hyperlink in the HTML pane, assuming that the target of the hyperlink is valid and after a noticeable delay, the contents will change to that of the target. In this demonstration should the URL be invalid the exception handler is implemented to output a message onto the terminal. The next chapter will address this problem by developing a more sophisticated HTML browser.

Object → EventObject → java.swing.event.HyperlinkEvent

```
public HyperlinkEvent( Object                   source,
                       HyperlinkEvent.EventType type,
                       URL                      target)
```

Constructor specifying the source of the event, the type of the event (ACTIVATED, ENTERED or EXITED) and the URL of the link's target.

```
public Object getSource()
public HyperlinkEvent.EventType getType()
public URL getURL()
```

Inquiry methods for the attributes.

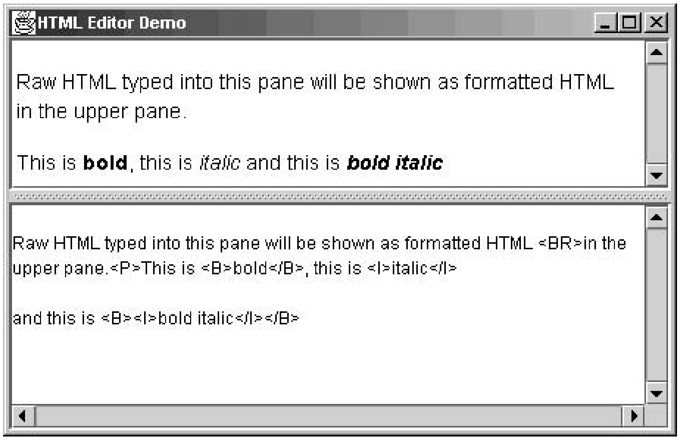
8.8 The *EditingHTMLPane* class: a primitive HTML editor

The final artifact to be presented in this chapter is illustrated in Figure 8.13. Visually it is similar to an *HTMLViewer* instance, but the lower, raw text, pane is editable and its contents are linked to the contents of the upper, HTML, pane.

To implement this artifact the *ScrollingTextPane* class is extended to provide the *EditingTextPane* class. This new class overrides the two constructors and adds two new methods called *addDocumentListener()* and *removeDocumentListener()*.

Referring back to Figure 8.3, when the user interacts with the visible JTextArea component by adding, deleting or replacing text it causes the Document instance that it contains to generate and dispatch a DocumentEvent. In order to implement the HTML editing capability these DocumentEvents will have to be listened to. In order to accomplish this the listener supplied to the *EditingTextPane* class's *addDocumentListener()* method will have to be registered with the Document instance associated with the JTextArea that is contained within it. The implementation of the *EditingTextPane* class, as far as the end of the constructors, is as follows.

```
package brewser;

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;


public class EditableTextPane extends ScrollingTextPane {


   public EditableTextPane() {
```



Raw HTML typed into this pane will be shown as formatted HTML
in the upper pane.

This is **bold**, this is *italic* and this is ***bold italic***

Raw HTML typed into this pane will be shown as formatted HTML <BR>in the
upper pane.<P>This is <B>bold</B>, this is <I>italic</I>

and this is <B><I>bold italic</I></B>

Figure 8.13 *HTMLEditor* within a demonstration harness.

```
      this( "");
   } // End EditableTextPane default constructor.

   public EditableTextPane( String toShow)
                         throws BrewserException {
      super( toShow);
      textPage.setEditable( true);
   } // End EditableTextPane  constructor.
```

The default constructor indirects to the alternative constructor, passing an empty String as an argument. The alternative constructor, which is noted as possibly throwing a *BrewserException*, commences by calling its **super**, *ScrollingTextPane* constructor. Assuming that no exception has been thrown the *textPage* JTextArea, declared with **protected** visibility in the *ScrollingTextPane* class, is set editable. Hence the only difference after construction between a *ScrollingTextPane* and an *EditableTextPane* is that the latter is editable allowing the user to enter, replace or remove the text that it contains.

The *addDocumentListener()* and *removeDocumentListener()* methods, which follow, complete the declaration of the *EditableTextPane* class.

```
   public void addDocumentListener(
                          DocumentListener listener) {
      textPage.getDocument().
                      addDocumentListener( listener);
   } // End addDocumentListener.

   public void removeDocumentListener(
                          DocumentListener listener) {
      textPage.getDocument().
                    removeDocumentListener( listener);
   } // End removeDocumentListener.

} // End EditableTextPane.
```

Both of these methods are wrappers on the Document addDocumentListener() and removeDocumentListener() methods. The identity of the Document instance contained within the *textPage* is established by using its inherited getDocument() method. The effect of registering a listener is that when the user changes the text in the visible component a DocumentEvent will be dispatched to it.

The *HTMLEditor* class is substantively identical to the *HTMLViewer* class described in detail above. The major omission is the capability to hide the raw pane, which would not be appropriate for an editor. Its single constructor does not take an argument and creates a JSplitPane containing an empty *ScrollingHTMLPane* instance in the upper location and an empty *EditableTextPane* instance in the lower location. It also registers itself as the DocumentListener of the *EditableTextPane* instance and consequently has to implement the DocumentListener **interface**. This **interface** mandates three methods, which are implemented as follows.

```
   public void insertUpdate( DocumentEvent event) {
      this.processDocumentEvent( event);
   } // End insertUpdate.

   public void removeUpdate( DocumentEvent event) {
```

```
0140        this.processDocumentEvent( event);
0141     } // End removeUpdate.
0142
0143     public void changedUpdate( DocumentEvent event) {
0144        this.processDocumentEvent( event);
0145     } // End changedUpdate.
```

All three of these methods indirect to the private *processDocumentEvent()* method, implemented as follows.

```
0148     private void processDocumentEvent( DocumentEvent event) {
0149
0150     Document document = event.getDocument();
0151     String   contents = null;
0152
0153        try {
0154          contents = document.getText( 0, document.getLength());
0155          htmlPane.setRawHTML( contents);
0156        } catch ( BadLocationException exception) {
0157           // do nothing.
0158        } // End try/catch.
0159     } // End processDocumentEvent.
```

The identity of the Document which generated the event is established on line 0150 and the entire *contents* of the document are obtained and stored on line 0154, using the Document getText() method. It is possible, but unlikely, that this method will throw a BadLocationException, so this step must be contained within a **try/catch** structure. Should the exception not be thrown the *contents* of the Document that generated the event, i.e. those of the editable *rawPane* at the bottom of the editor, are installed as the contents of the upper *htmlPane*.

EventListener → javax.swing.event.DocumentListener

```
public void changedUpdate( DocumentEvent event)
public void insertUpdate(  DocumentEvent event)
public void removeUpdate(  DocumentEvent event)
```

The insertUpdate() method is called when text is added or inserted, removeUpdate() when it is deleted or cut. The changedUpdate() is called when it is programmatically changed in some way.

The effect is that whenever the user changes even a single character in the lower part of the editor, the entire contents of the upper area are re-rendered. This is only just acceptable for a primitive demonstration of the capability, such as this component, but would be unacceptable for a production component. The considerations and techniques involved in delving within the DocumentEvent and Document classes to change only the appropriate part of the HTML document are beyond the scope of this book. It also lacks the capability to store the edited contents, the techniques for which will be introduced in the next chapter.

Summary

- The JTextComponent classes have an internal model, view, controller architecture.
- The JTextArea will render plain text; the JEditorPane can render plain text as RTF or HTML.
- The URL class supplies the capability to retrieve resources across the Internet.
- A JEditorPane can obtain and render an HTML document by being supplied with a URL.
- The JSplitPane class can contain at most two children and supplies a horizontal or vertical bar which the user can drag to vary the space afforded to each child.
- A JScrollBar will dispatch an AdjustmentEvent to an AdjustmentListener's adjustment ValueChanged() method when the user drags on it.
- A non-editable JEditorPane showing an HTML document will dispatch a Hyperlink Event to a HyperLinkListener's hyperLinkUpdate() method when the user activates it.
- An editable JTextComponent will dispatch a DocumentEvent to various DocumentListener methods when the user changes the text in some way.

Exercises

8.1 Extend the *TipPanel* class from Chapter 3 to include a JPasswordField component into which the user must type a password in order to be able to change the tool tip.

8.2 Implement the *ScrollingTextPane* demonstration harness as illustrated in Figure 8.7.

8.3 Investigate the limitations of the JEditorPane's HTML compatibility by loading sample HTML documents which use increasingly complex HTML constructs. (The early releases of the JFC have only limited HTML capability, but Sun Microsystems has given a commitment to improve the capability in subsequent releases and third-party JFC suppliers may have different levels of compatibility.)

8.4 Extend the *HTMLViewer* class to contain three panes, with the third pane containing the raw HTML stripped of all its tags. All three of the pane's scroll bars should be linked together.

8.5 Add a main menu bar to the *HTMLEditor* class containing an Insert menu whose items, when activated, will place an HTML tag into the text area. For example a Paragraph item would cause `<P>` to be inserted.

8.6 Extend the artifact from Exercise 8.5 to include specialized dialogs which allow more complex tags to be inserted. For example the `<FONT>` tag can allow the `SIZE`, `FONT` and `COLOR` attributes of the text to be specified and the *HTMLFontDialog* should allow the user to specify these options.

9

Internal frames, loading, saving and printing

9.1 Introduction

This chapter will continue the introduction to the pre-supplied JFC classes by developing artifacts which illustrate the use of one of the most complex Swing capabilities, internal frames. Many standard desktop applications including word processors, IDEs and graphics editors, make use of multiple internal windows. For example, the design of a graphics editor which can have more than one image open for viewing or editing at any instant could use single or multiple windows. A single-window design would use the application's main work area to display the current image and would change to show a different image when required. A multiple internal window design would have a different window for each image and the user could indicate which should be the current focus of concern by clicking upon it or indicating via a *Windows* menu.

For most modern artifacts the multiple internal window design is favored, as it recognizes that users rarely concentrate upon a single task at a time and may have to split their attention between multiple (related or unrelated) tasks. For example, the image editor may be being used to prepare an image which will be a constituent part of a second image. When the time comes for the composite image to be assembled it is easier for the user to have both windows visible and then copy from one and paste into the other. The alternative would be to be forced to navigate an *Images* menu between the copy and the paste operation, which would make the task less concrete, more complex and less natural.

In this chapter the HTML components from the previous chapter will be further developed to provide a simple Web browser known as a *Brewser*. This artifact will not only illustrate how to use multiple internal windows but will also allow the means of loading from and saving to the local file system and also the simple use of a printer to be illustrated.

9.2 The *Brewser* – overview

Figure 9.1 illustrates the *Brewser* artifact in use with the *Window* main menu, *Windows...* cascading menu posted. The cascaded menu indicates that there are four HTML pages available, each of which is identified by its URL and has the page's title associated with it as a tool tip. The reason for this is that a URL is necessarily unique, whereas a page title may not be; however, the page titles are shown on decorations of the internal windows. Of the four pages which are open, three of the internal windows that they are displayed within are clearly visible and the fourth is iconified in the lower left-hand corner. When a

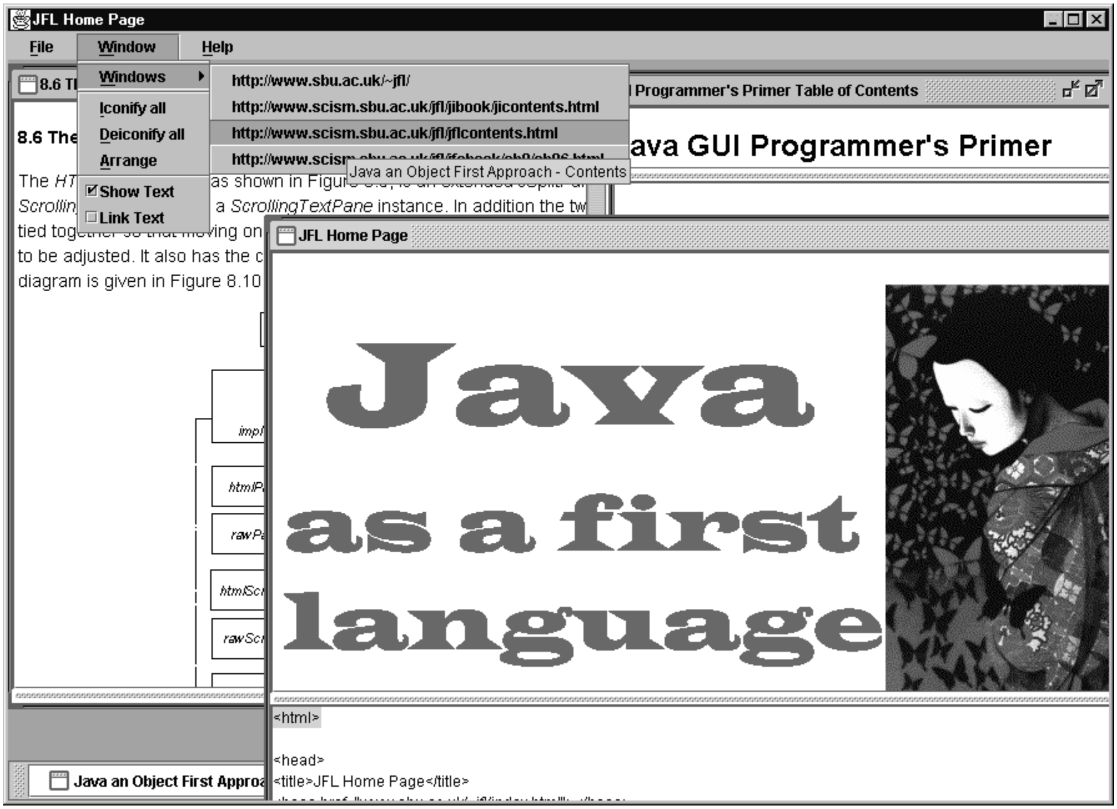

JFL Home Page
File
Window
Help
Windows
Iconify all
Deiconify all
Arrange
☑ Show Text
□ Link Text
http://www.sbu.ac.uk/~jfl/
http://www.scism.sbu.ac.uk/jfl/jibook/jicontents.html
http://www.scism.sbu.ac.uk/jfl/jflcontents.html
http://www.scism.sbu.ac.uk/jfl/jfobook/ch9/ch06.html
Java an Object First Approach - Contents
8.6 Th
8.6 The
Programmer's Primer Table of Contents
ava GUI Programmer's Primer
JFL Home Page
Java
as a first
language
impl
htmlP
rawPa
htmlScr
rawScr
Java an Object First Approa
<html>
<head>
<title>JFL Home Page</title>

Figure 9.1 *Brewser*: visual appearance.

number of internal windows are open, one (and only one) can be active, and is visually distinguished in some way from all the others. In the *Brewser* the title of the currently active window is also displayed in the artifact's frame.

The menu bar *Windows* menu also contains controls to *Iconify* all windows, to *Deiconify* all iconified windows and to *Arrange* all the windows within the application's main work area. The last two controls are instances of the JCheckBoxMenuItem class and reflect the state of, and control, the currently active window. The *Show Text* check box determines whether the raw HTML is to be shown in a split pane below the rendered HTML and, if the text is showing, the second check box determines whether the two panes are to be linked together.

The *Help* menu contains only an *About...* item, which leads to an information dialog which will not be presented. The two possible appearances of the *File* menu are shown in Figure 9.2. The left-hand illustration shows its appearance when no documents are loaded and the only items available are those to open one by specifying its *URL...* or from a *File...* on the local file store. The *Exit* item is also available when the *File* menu is in this state and leads to a confirmation dialog as presented in Chapter 6. The right-hand illustration shows the state of the *File* menu when there is at least one document open, and the various *Save...*, *Print...* and *Close* items refer to the currently active window.

The instance diagram for the *Brewser* artifact is given in Figure 9.3. A *brewser* contains an instance of *BrewserMenuBar* and also a JDesktopPane called *mainArea*. A

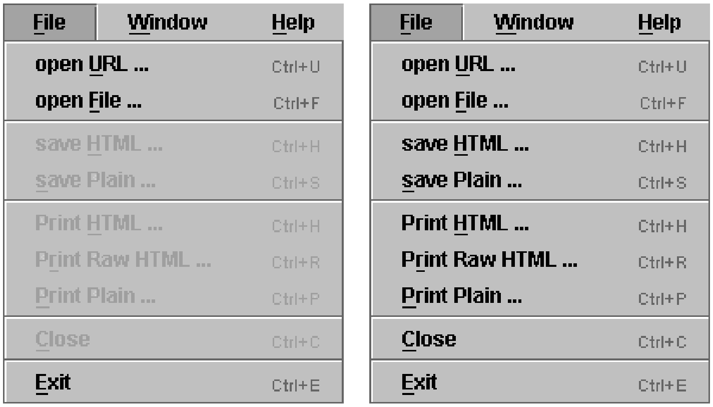


Figure 9.2 *Brewser*: *File* menu.

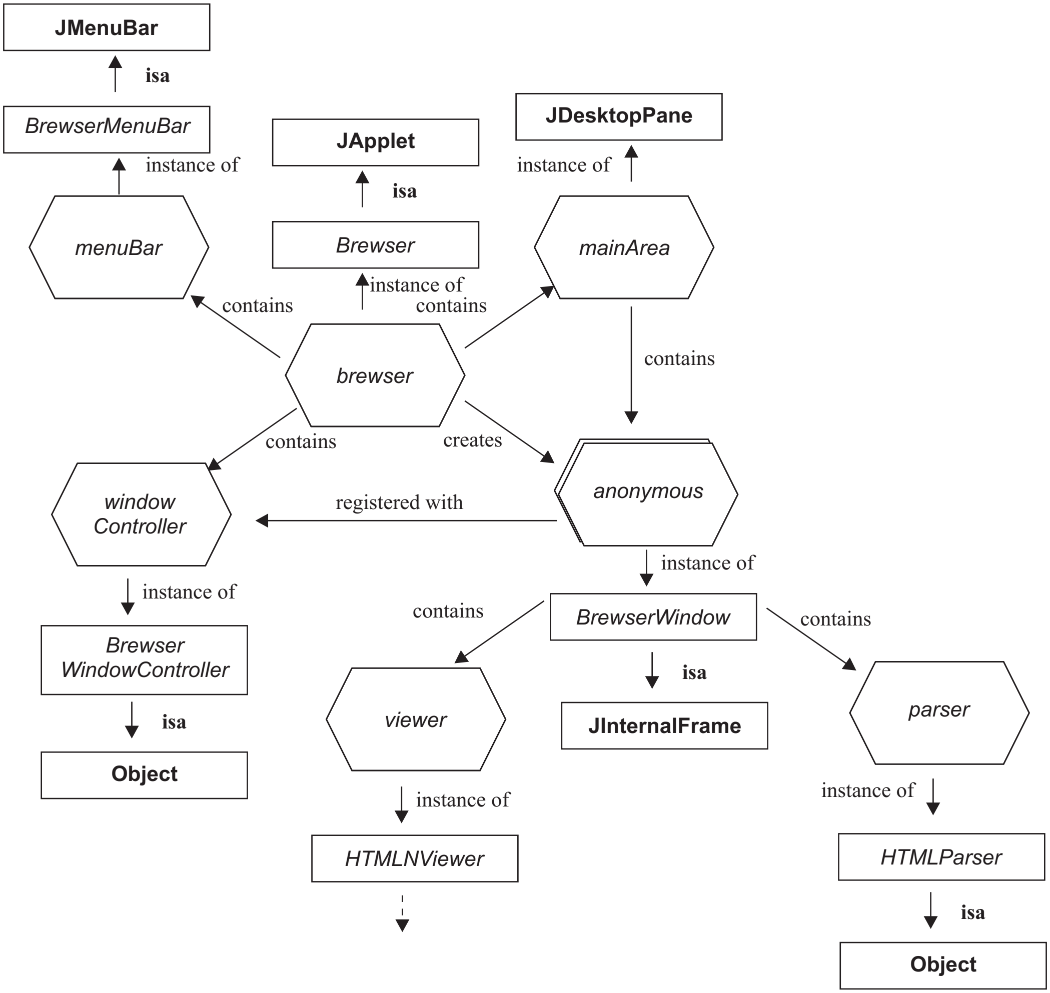


Figure 9.3 *Brewser*: instance diagram.

JDesktopPane supplies the capability to contain and manage a number of JInternalFrame instances which are used to provide the multiple internal windows. The diagram also shows that the *mainArea* contains a number of anonymous *BrewserWindow*s which are extended JInternalFrame instances, and which are created by the *brewser*. Each *BrewserWindow* contains an instance of *HTMLNViewer*, a navigating version of the *HTMLViewer* developed in the last chapter, and also an instance of the *HTMLParser* class. This second attribute is needed to parse the raw HTML in order to extract the *title* tag from it. Each *BrewserWindow* is registered with the *windowController*, an instance of the *BrewserWindowController* class, which maintains knowledge of the URL, title and identity of all windows.

9.3 *Brewser*: subsidiary classes

The focus of this part of this chapter is on the use of multiple internal windows, loading and saving and printing. Accordingly, many of the other classes shown in the instance diagram will only be described in overview. The Appendix contains details of where a complete copy of the source code used in this chapter can be obtained.

The *BrewserMenuBar* class diagram is given in Figure 9.4. Its implementation does not differ significantly from the *MemoryGameMenuBar* implementation given in Chapter 6. The constructor will create the main menu bar system shown in Figures 9.1 and 9.2 and requires as arguments an *actionListener* and a *menuListener*, which are installed as resources in all items and menus respectively. The *setFileMenuOptions()* method takes a single **boolean** argument and configures the file menu, depending upon the argument, to one of the two states shown in Figure 9.2. The *setWindowOptionsOff()* method will clear and disable the two check boxes on the *Windows* menu. The *setShowCheckState()* and

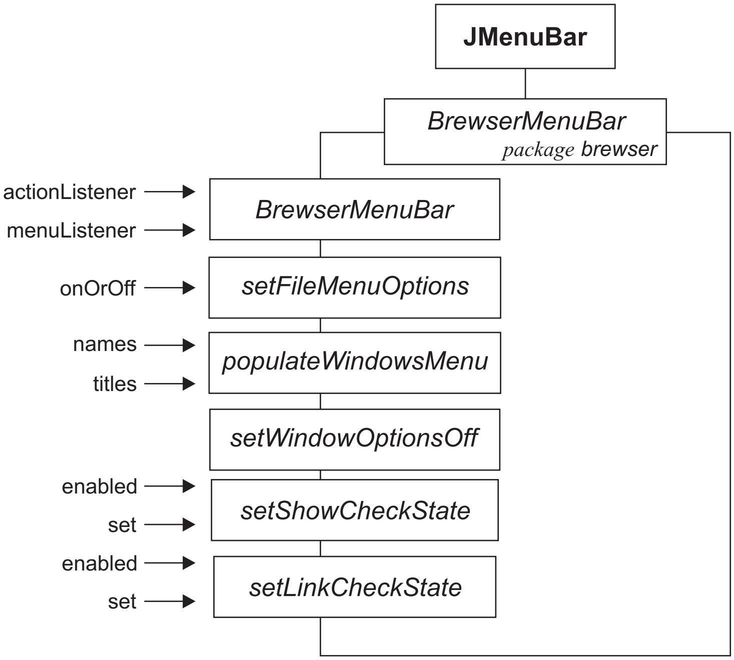


Figure 9.4 *BrewserMenuBar*: class diagram.

setLinkCheckState() each control one of the check boxes and their two arguments if the item is to be shown enabled and, if so, if its checkmark is to be set.

The *populateWindowsMenu()* method takes as argument an Enumeration of *names* and an Enumeration of *titles* which it uses to populate the *Windows...* cascading menu. It is the only *BrewserMenuBar* method whose implementation, as follows, will be given.

```
0124    protected void populateWindowsMenu( Enumeration names,
0125                                        Enumeration titles) {
0126
0127    String    thisName  = null;
0128    String    thisTitle = null;
0129    JMenuItem thisItem  = null;
0130
0131       if ( ! names.hasMoreElements()) {
0132          windowsMenu.setEnabled( false);
0133          iconifyAllButton.setEnabled( false);
0134          deIconifyAllButton.setEnabled( false);
0135          arrangeButton.setEnabled( false);
0136       } else {
0137          windowsMenu.setEnabled( true);
0138          iconifyAllButton.setEnabled( true);
0139          deIconifyAllButton.setEnabled( true);
0140          arrangeButton.setEnabled( true);
0141          windowsMenu.removeAll();
0142          while( names.hasMoreElements()) {
0143             thisName  = (String) names.nextElement();
0144             thisTitle = (String) titles.nextElement();
0145             thisItem = new JMenuItem( thisName);
0146             thisItem.setToolTipText( thisTitle);
0147             thisItem.setActionCommand( "switchWindow");
0148             thisItem.addActionListener( sendActionsTo);
0149             windowsMenu.add( thisItem);
0150          } // End while.
0151       } // End if.
0152    } // End populateWindowsMenu.
```

The substantive part of the method is implemented as an **if/else** structure which is controlled by the *names* Enumeration passed as an argument and which may be empty. If the Enumeration is empty, then, on lines 0132 to 0135, the first four items on the *Windows* main menu, including the *Windows...* cascading menu button, are disabled, indicating that no documents are loaded. Otherwise, if the Enumeration is not empty, lines 0137 to 0140 enable these four components. Line 0141 then removes all the items currently on the cascading menu before the **while** loop, between lines 0143 and 0149, populates it with JMenuItems whose *label* and *toolTip* resources are obtained from the Enumerations. The actionCommand of each item is set to *switchWindow* and all will dispatch ActionEvents to the listener supplied to the menu bar constructor, when the user activates them. The effect of this method, when supplied with a pair of Enumerations each containing four elements, can be seen on Figure 9.2.

The class diagram for *BrewserWindowController* is given in Figure 9.5. The three encapsulated attributes are all Vectors containing the names (URLs), titles and identities of the windows that are currently open. The methods are supplied to support the registration,

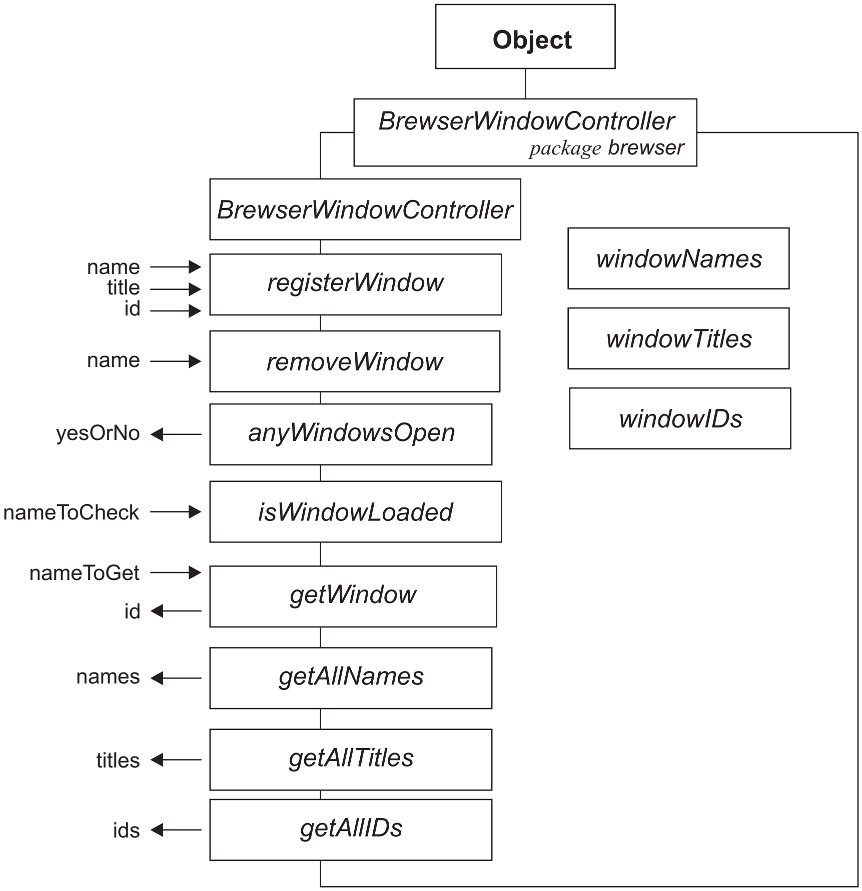


Figure 9.5 *BrewserWindowController*: class diagram.

removal and various inquiries of the state of the controller. In particular, the last three methods supply Enumerations which are used by the *brewser* to maintain the *Windows...* menu as just described.

The class diagram for the *HTMLNViewer* class is given in Figure 9.6 and it is very similar to the *HTMLViewer* class diagram given in Chapter 8, with the addition of a capability to support a HyperlinkListener and to dispatch HyperlinkEvents to it when the user activates a link within the encapsulated *htmlPane* component. This is an instance of the *NavigatingHTMLPane* class, also described in the last chapter.

The class diagram for *HTMLParser* will not be given. It takes an HTML string as an argument to its constructor and has two inquiry methods which will be used in the *brewser* artifact. These are *getPlainText()*, which returns a copy of the <BODY> part of the HTML document with all the tags removed, and *getTitle()*, which returns the contents of the <TITLE> tag in the <HEAD> of the document. The class diagram for the *BrewserWindow* class is given in Figure 9.7. The constructor takes as an argument a String URL whose contents it is *toShow* and will throw a *BrewserException* if the URL cannot be opened or its contents retrieved. The *addNotify()* method is supplied so that the component can establish a size and position as it becomes visible. The remaining methods are all

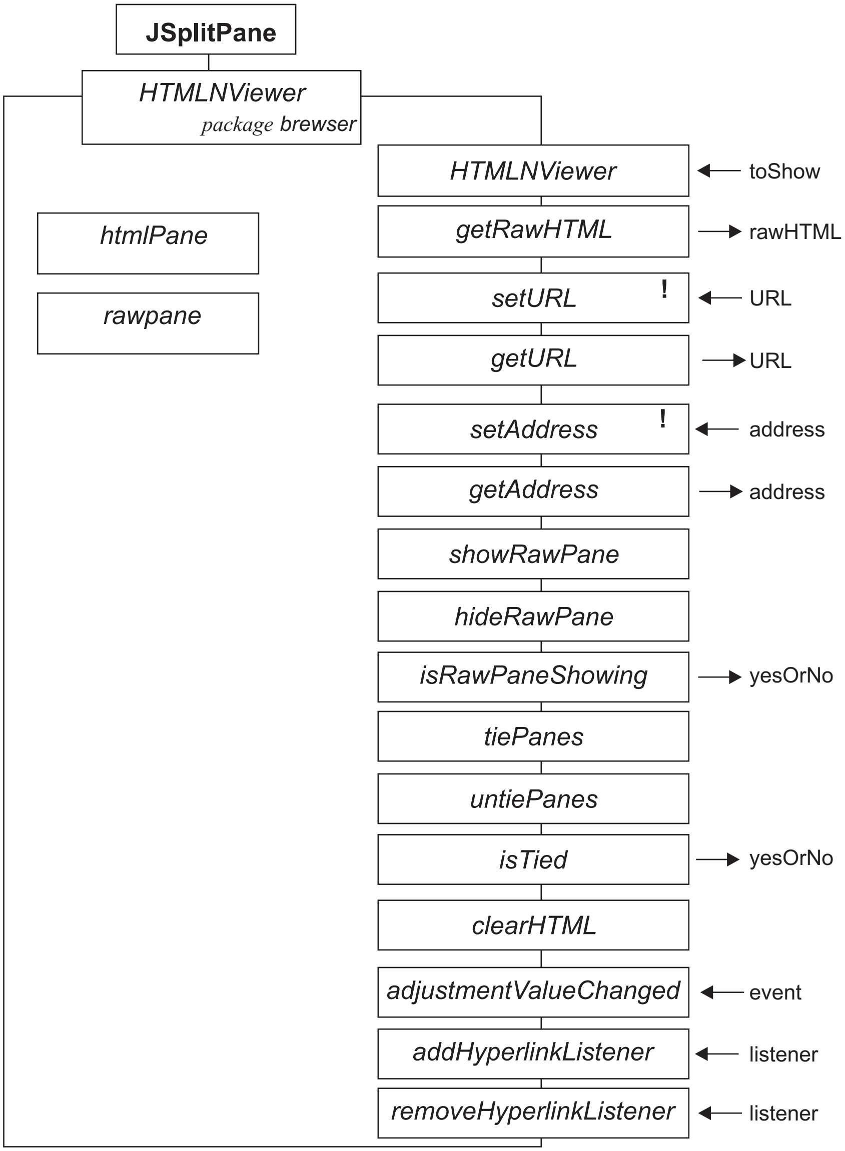


Figure 9.6 *HTMLNViewer*: class diagram.

wrappers on to the corresponding *viewer* or *parser* methods. The implementation of the class, as far as the start of its constructor, is as follows.

```
package brewser;

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

```

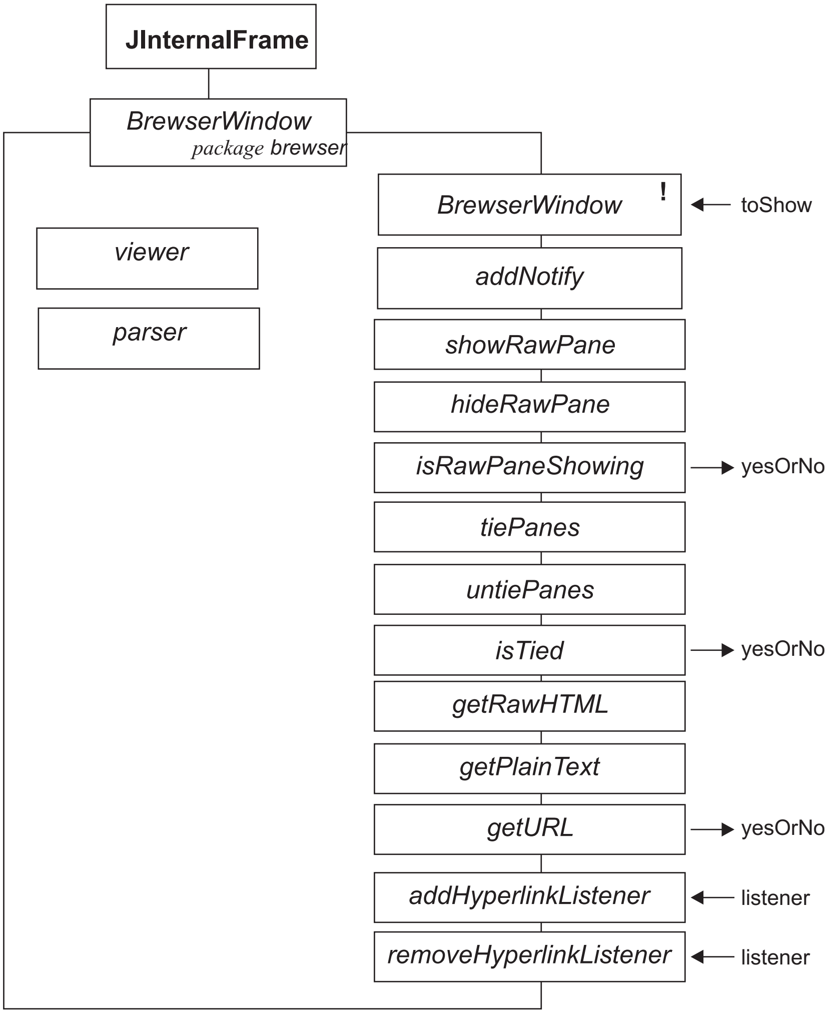


Figure 9.7 *BrewserWindow*: class diagram.

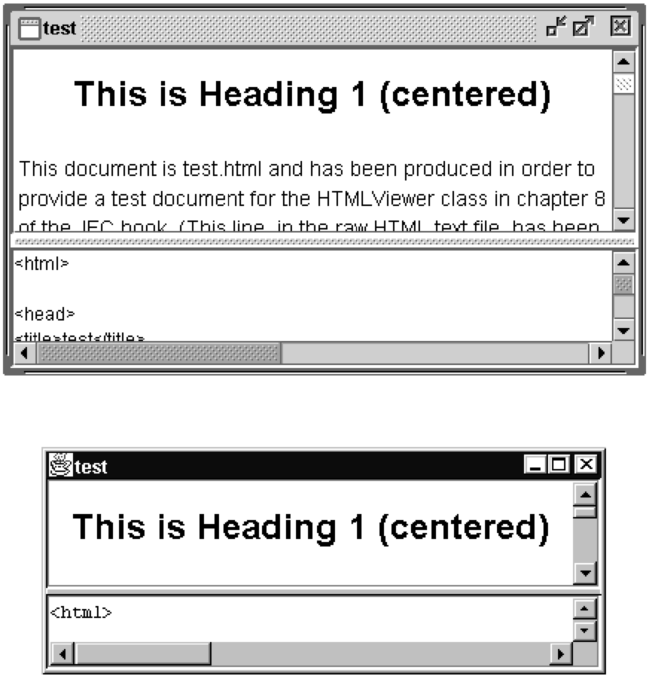
```
public class BrewserWindow extends JInternalFrame {

private HTMLNViewer viewer = null;
private HTMLParser  parser = null;

   public BrewserWindow( String toShow) {
      super( "", true, true, true, true);

      viewer = new HTMLNViewer( toShow);
      if ( viewer == null) {
         throw new BrewserException(
                          BrewserException.LOAD_FAULT);
      } else {
         parser = new HTMLParser( viewer.getRawHTML());
0031          this.setTitle( parser.getTitle());
0032          viewer.doLayout();
0033          this.getContentPane().add( viewer);
0034       } // End if.
0035    } // End BrewserWindow constructor.
```

The constructor commences on line 0023 with a call of the **super**, JInternalFrame, constructor. The arguments to the constructor are the title to display on its decoration, empty at the moment, and permissions to make it resizable, closeable, maximizable and iconifiable. The latter four arguments will cause the internal frame to show the appropriate controls, as illustrated for the Metal and Windows look and feels in Figure 9.8. In both cases three of the controls are shown at the top right of the frame with the resizable capability effected by dragging upon one of the corners.

The constructor continues, on line 0025, by passing the argument, *toShow*, on to the *HTMLNViewer* constructor. If the resource cannot be opened the instance will not be constructed, the value of *viewer* will be **null** which will result, on lines 0027 and 0028, in the throwing of a *BrewserException* from the constructor, as indicated in the class diagram. Otherwise, on lines 0030 to 0033, the *parser* can be constructed and the title of the internal frame established from its *getTitle()* method. The final steps in the constructor are to establish the layout of the *viewer* before it is added to the contentPane of **this** JInternalFrame.

The *addNotify()* method will be called immediately before the component becomes visible to the user and is implemented as follows.

```
0038    public void addNotify() {
0039
```



test

This is Heading 1 (centered)

This document is test.html and has been produced in order to provide a test document for the HTMLViewer class in chapter 8 of the JFC book. (This line, in the raw HTML text file, has been

```
<html>

<head>
<title>test</title>
```

test

This is Heading 1 (centered)

```
<html>
```

Figure 9.8 *JinternalFrame* controls with Metal and Windows look and feels.

```
0040    Dimension itsParentsSize = null;
0041
0042       super.addNotify();
0043       itsParentsSize = this.getParent().getSize();
0044       this.setSize( itsParentsSize.width  - 75,
0045                     itsParentsSize.height - 75);
0046       this.toFront();
0047    } // End addNotify.
```

The effect of this method is to establish a size for the internal frame which is a little smaller than the instance parent, JDesktopPane, that it will appear upon. The last step, on line 0046, ensures that it appears at the front, above any other internal frames and ensures that it has the user's focus. The remaining *BrewserWindow* methods are all wrappers on to the corresponding *viewer* or *parser* methods and will not be presented.

JComponent → java.swing.JInternalFrame

```
public JInternalFrame()
public JInternalFrame( String title)
public JInternalFrame( String title, boolean resize)
public JInternalFrame( String title, boolean resize,   boolean close)
public JInternalFrame( String title, boolean resize,   boolean close,
                                     boolean maximize)
public JInternalFrame( String title, boolean resize,   boolean close,
                                     boolean maximize, boolean iconify)
```

Constructors: if a control is not explicitly established by an argument then it is not supplied.

```
public String  getTitle()
public void    setTitle( String newTitle)
public boolean isResizable()
public void    setResizable( boolean yesOrNo)
public boolean isCloseable()
public boolean isClosed()
public void    setCloseable( boolean yesOrNo)
public boolean isMaximizable()
public boolean isMaximum()
public void    setMaximizable ( boolean yesOrNo)
public boolean isIconifiable()
public boolean isIcon ()
public void    setIconifiable( boolean yesOrNo)
```

State setting and inquiry methods on the attributes and the actual state of the frame.

```
public void addInternalFrameListener(
                          InternalFrameListener listener)
public void removeInternalFrameListener(
                          InternalFrameListener listener)
```

Internal frames dispatch InternalFrameEvents as the user interacts with them to any registered listeners.

```
public JRootPane    getRootPane()
public void         setRootPane(    JRootPane newPane)
public JContentPane getContentPane()
public void         setContentPane( JContentPane newPane)
public JGlassPane   getGlassPane()
public void         setGlassPane(   JGlassPane newPane)
public JMenuBar     getJMenuBar()
public void         setJMenuBar( JMenuBar newMenu)
public JDesktopPane getDesktopPane()
```

An internal frame has a complete complement of panes and can support a menu bar.

```
public void dispose()
public int  getLayer()
public void setLayer( Integer layer)
public void moveToBack()
public void moveToFront()
```

Various methods to control the internal frame's stacking order, that is its layered location relative to the other windows determining which will overlay which. Also a method to dispose of the frame, and its contents, when it is no longer required.

9.4 *Brewser*: window control

Having described the subsidiary classes shown on the instance diagram in Figure 9.3 the implementation of the *Brewser* can be described. This section of the chapter will describe a minimal set of internal window controls and actions. This will include all the controls on the *Windows...* menu and the *Open URL...* item on the *File* menu. The next section of the chapter will describe the remaining options, which rely upon access to the local machine's resources. The implementation of the *Brewser* class as far as its *init()* method is as follows.

```
package brewser;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.plaf.*;
import javax.swing.filechooser.*;
import java.beans.PropertyVetoException;
import java.io.*;
import java.net.*;
import java.util.*;

public class Brewser extends JApplet
                  implements ActionListener,
                               MenuListener,
                          HyperlinkListener,
                             WindowListener,
                     InternalFrameListener {

private final static int HTML_FORMAT  = 0;
```

```
0031 private final static int RAW_FORMAT   = 1;
0032 private final static int PLAIN_FORMAT = 2;
0033
0034 private BrewserMenuBar menuBar   = null;
0035 private JDesktopPane   mainArea  = null;
0036
0037 private BrewserWindowController windowController = null;
0038 private BrewserWindow           currentlyActive  = null;
0039
0040 private boolean frameFound = false;
0041 private JFrame  topLevel   = null;
```

The *Brewser* class extends the JApplet class and implements a number of interfaces. It has to implement the ActionListener and MenuListener interface in order to listen to the events dispatched as a result of the user's interaction with the main menu. It has to implement the HyperlinkListener interface to listen to the events generated when the user activates a link in an HTML document; and it has to implement the WindowListener and InternalFrameListener interfaces in order to listen to the events when the user interacts with the top-level window or one of the internal frames.

The three manifest values declared on lines 0030 to 0032 are used to indicate the save and print formats as shown on the *File* menu. The first three attribute declarations, starting on line 0034, are as shown on the instance diagram. The *currentlyActive* attribute, declared on line 0038, will always indicate the identity of the *BrewserWindow* which has mouse and keyboard focus. The final two attributes are used to record the *topLevel* frame within which the currently active window's title can be displayed. The *init()* method is implemented as follows.

```
0043    public void init() {
0044       this.setResources();
0045
0046       windowController = new BrewserWindowController();
0047       menuBar          = new BrewserMenuBar( this, this);
0048       mainArea         = new JDesktopPane();
0049
0050       this.setJMenuBar( menuBar);
0051       this.getContentPane().add( mainArea);
0053    } // End init.
```

Once the resources have been set, lines 0046 to 0048 construct the three major attributes as shown on the instance diagram. The *menuBar* is then installed and the *mainArea* added as the only instance child of the applet's *contentPane*. The consequence of this method is that the *Brewser* will become visible to the user with an empty desktop pane area.

JComponent → JLayeredPane → java.swing.JDesktopPane

```
public JDesktopPane()
```

Constructs an empty desktop pane.

```
public JInternalFrame[] getAllFrames()
public JInternalFrame[] getAllFramesInLayer( int layer)
```

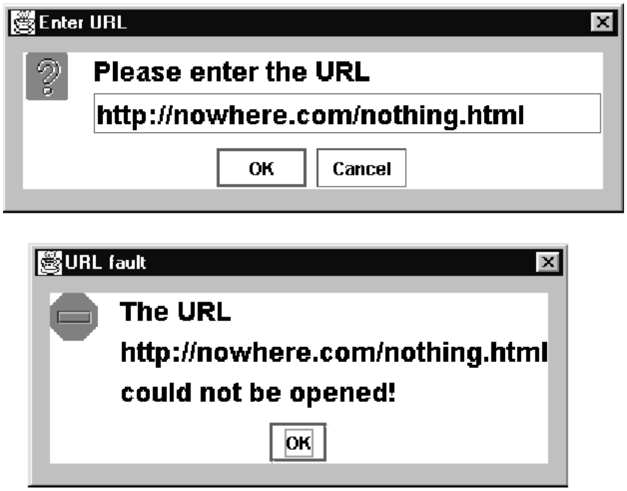
Methods to obtain the identities of the InternalFrames displayed on the pane; methods to add and remove frames are inherited from JContainer.

When the *Brewser* first becomes visible the most likely action of the user will be to open a document. If the user activates the *File* menu *Open URL...* item it will dispatch an ActionEvent containing the actionCommand *openURL* to the *Brewser*'s *actionPerformed()* method which will be detected in an **if/else if** structure and indirect to a method called *openURL()* whose implementation is as follows.

```
private void openURL() {

String toOpen = null;

   toOpen = JOptionPane.showInputDialog( this,
                "Please enter the URL", "Enter URL",
                      JOptionPane.QUESTION_MESSAGE);

   if ( toOpen != null       &&
        toOpen.length() > 0  ){
      this.openByURL( toOpen);
   } // End if.
} // End openURL.
```

The method posts a JOptionPane input dialog, illustrated in Figure 9.9, asking the user to enter a URL and stores the reply in the local String variable *toOpen*. If the user has typed something in, the method indirects to the *openByURL()* method passing the user's input as the argument. The implementation of this method is as follows.

```
private void openByURL( String toOpen) {

   try {
      this.openNewWindow( toOpen);
   } catch ( BrewserException exception) {
```


Enter URL

Please enter the URL

http://nowhere.com/nothing.html

URL fault

The URL
http://nowhere.com/nothing.html
could not be opened!

Figure 9.9 *Brewser Enter URL* and *URL fault* dialogs.

```
0162           JOptionPane.showMessageDialog( this,
0163                           "The URL \n" + toOpen +
0164                           "\ncould not be opened!",
0165                           "URL fault",
0166                           JOptionPane.ERROR_MESSAGE);
0167        } // End try/catch.
0168     } // End openByURL.
```

This method, which, as will be shown, may also be called as a consequence of the user activating a hyperlink, indirects to a third method, called *openNewWindow()*. The implementation of this method, which will be given shortly, will throw a *BrewserException* if the HTML resource could not be obtained. Should this happen the exception will be caught by this method, resulting in a JOptionPane error dialog being shown to the user, as also illustrated in Figure 9.9.

The called *openNewWindow()* method is the only place in the *Brewser* class which can place a new internal window on the desktop and may also be called as a consequence of the user activating a hyperlink or posting the *File* menu *Open File...* dialog. It is implemented as follows.

```
0171     private void openNewWindow( String urlToOpen)
0172                                     throws BrewserException {
0173
0174     BrewserWindow newWindow = null;
0175
0176        if ( windowController.isWindowLoaded( urlToOpen)) {
0177           this.setActiveWindow( urlToOpen);
0178        } else {
0179           newWindow = new BrewserWindow( urlToOpen);
0180           newWindow.addInternalFrameListener( this);
0181           newWindow.addHyperlinkListener( this);
0182           mainArea.add( newWindow);
0183           windowController.registerWindow( urlToOpen,
0184                               newWindow.getTitle(), newWindow);
0185           this.setActiveWindow( newWindow);
0186           menuBar.populateWindowsMenu(
0187                              windowController.getAllNames(),
0188                              windowController.getAllTitles());
0189        } // End if.
0190     } // End openNewWindow.
```

The method commences by using the *windowController* inquiry method *isWindow Loaded()* to determine whether the *urlToOpen* passed as an argument is already available in an existing window. If so, on line 0177, that window is nominated as the active window by calling the *setActiveWindow()* method. Otherwise an attempt has to be made to create a new internal window displaying the resources identified by the *urlToOpen*. This commences, on line 0179, by constructing a new *BrewserWindow* passing the *urlToOpen* as an argument. It is this step that may throw a *BrewserException*, which will be propagated from this method and, as shown above, may cause the *URL Fault* dialog to be posted.

If the exception is not thrown **this** *brewser* instance is registered as its InternalFrame and Hyperlink listener before, on lines 0182 to 0185, it is added to the *mainArea*, registered with the *windowController* and nominated as the active window. The last step in the method, on lines 0186 to 0188, updates the *Windows* menu *Windows...* cascading menu by calling the *menuBar populateWindowsMenu()* method, passing as arguments two Enumerations obtained from the *windowController* which contain the names and titles of all registered windows.

The effect of all these methods is that when users activate the *File* menu *Open URL...* item a dialog will be posted inviting them to type in a URL. If they *Cancel* the dialog or press <ENTER> in the text field without typing anything the request is cancelled. Otherwise, if the HTML resource could not be obtained an error dialog will explain this to the user, or an existing window will be promoted to the top of all the windows in the main area or a new window will appear, possibly after a delay, containing the requested resource.

As **this** *brewser* instance has been registered as the Hyperlink listener of all the BrewserWindows, whenever the user activates a link in an HTML document a HyperlinkEvent will be dispatched to the hyperlinkUpdate method, implemented as follows.

```
public void hyperlinkUpdate( HyperlinkEvent event) {

   if ( event.getEventType() ==
               HyperlinkEvent.EventType.ACTIVATED) {

      this.openByURL( event.getURL().toString());
   } // End if.
} // End hyperlinkUpdate.
```

This method extracts the URL from the *event* and passes it to the *openByURL()* method which will result in either a *URL Fault* dialog being posted, the window containing the resource becoming the active window or a new window containing the resource appearing. The *brewser* is also registered as the InternalFrame listener of all the *BrewserWindow*s, so it receives InternalFrameEvents as the windows are manipulated. Of the seven methods mandated by the InternalFrame listener interface, only two, as follows, need to be supplied with bodies; the rest can be provided as empty methods.

javax.swing.event.Interface InternalFrameListener interface

```
public void internalFrameOpened(      InternalFrameEvent event)
public void internalFrameActivated(   InternalFrameEvent event)
public void internalFrameDeactivated( InternalFrameEvent event)
public void internalFrameIconified(   InternalFrameEvent event)
public void internalFrameDeiconified( InternalFrameEvent event)
public void internalFrameClosing(     InternalFrameEvent event)
public void internalFrameClosed(      InternalFrameEvent event)
```

Methods called as an internal window is manipulated.

Object → EventObject → AWTEvent → javax.swing.event.InternalFrameEvent

```
public InternalFrameEvent( JInternalFrame source, int id)
```

The two significant attributes of the event are the source and the id, which can be one of INTERNAL_FRAME_ACTIVATED, INTERNAL_FRAME_CLOSED, INTERNAL_FRAME_CLOSING, INTERNAL_FRAME_DEACTIVATED, INTERNAL_FRAME_

DEICONIFIED, INTERNAL_FRAME_ICONIFIED or INTERNAL_FRAME_OPENED. The attributes can be obtained using inherited inquiry methods.

```
public void internalFrameActivated(
                                 InternalFrameEvent event) {
   this.setActiveWindow( (BrewserWindow) event.getSource());
} // End internalFrameActivated

public void internalFrameClosed( InternalFrameEvent event) {
   this.closeWindow( (BrewserWindow) event.getSource());
} // End internalFrameClosed.
```

The *intenalFrameActivated()* method indirects to the *setActiveWindow()* method which is also called from the *openNewWindow()* method, and ensures that the *brewser* attribute *currentlyActive* is always maintained to indicate the window which has focus. There are two versions of the *setActiveWindow()* method, implemented as follows.

```
private void setActiveWindow( String windowName) {
   this.setActiveWindow(
          windowController.getWindow( windowName));
} // End setActiveWindow.

private void setActiveWindow( BrewserWindow toSet) {

Component thisOne   = null;
Component itsParent = null;
   if ( ( toSet == null)            ||
        ( toSet == currentlyActive) ){
      return;
   } // End if

   if ( ! frameFound) {
      itsParent = this.getParent();
      while ( ! frameFound) {
         thisOne   = itsParent;
         itsParent = itsParent.getParent();
         if ( itsParent == null) {
            frameFound = true;
            if ( thisOne instanceof JFrame) {
               topLevel = (JFrame) thisOne;
            } // End if.
         } // End if.
      } // End while.
   } // End if.

   currentlyActive = toSet;
   if ( topLevel != null) {
      topLevel.setTitle( “Brewser “ +
                        currentlyActive.getTitle());
   } // End if.

   try {
```

```
0470             currentlyActive.setIcon( false);
0471             currentlyActive.toFront();
0472             currentlyActive.setSelected( true);
0473         } catch ( PropertyVetoException exception) {
0474             // Do nothing.
0475         } // End try/catch.
0476      } // End setActiveWindow
```

The first version takes a String as an argument and indirects to the second method after asking the *windowController* for the window identity associated with the string. The second version commences with an early return if the window identity *toSet* is **null** or if the window being nominated is already the active window. Otherwise, if the value of the *brewser fameFound* attribute is **false**, lines 0449 to 0460 ascend the instance window hierarchy looking for a *topLevel* frame.

If a *topLevel* frame can be found, which indicates that the *brewser* is executing within a top-level window and not as part of a conventional browser, on lines 0465 and 0456, the title of the *topLevel* frame is set to include the *title* of the new active window. The final part of the method, on lines 0470 to 0472, ensures that the active window is not iconified, that it is at the front of all other windows and that it is selected to receive mouse and keyboard events. The setIcon() and setSelected() methods may throw a PropertyVeto exception which cannot be ignored, but which does not require any actions giving the empty exception handler at the end of the method. PropertyVeto exceptions will be introduced and explained in detail in the next chapter.

The currently active window can be closed by activating the *Close* item on the *File* menu, which will be intercepted within the *brewser actionPerformed()* method and result in the *closeWindow()* method being called, as follows.

```
0089          } else if ( command.equals( "close")) {
0090             this.closeWindow( currentlyActive);
```

An internal window can also be closed by clicking upon the close icon in its frame decoration. This will result in an *InternalFrameEvent* being dispatched to the *brewser InternalFrameClosed* method given above, which also calls the *closeWindow()* method, implemented as follows.

```
0479     private void closeWindow( BrewserWindow toClose) {
0480
0481     String       itsURL     = toClose.getURL();
0482     Enumeration  allWindows = null;
0483
0484        windowController.removeWindow( itsURL);
0485        if ( toClose == currentlyActive        &&
0486             windowController.anyWindowsOpen() ){
0487           allWindows = windowController.getAllWindows();
0488           this.setActiveWindow(
0489                      (BrewserWindow) allWindows.nextElement());
0490        } // End if.
0491
0492        toClose.dispose();
0493        menuBar.populateWindowsMenu(
0494                           windowController.getAllNames(),
0495                           windowController.getAllTitles());
0496     } // End closeWindow
```

The method commences, on line 0481, by obtaining the *URL* of the window *toClose* and then asking the *windowController* to remove it from its list of active windows. It then checks to see whether the window being closed is the *currentlyActive* window and if so, on lines 0486 to 0490, nominates the first window in the *allWindows* Enumeration obtained from the *windowController* as the *currentlyActive* window. This ensures that, so long as there is at least one window still open, there will always be a *currentlyActive* window.

The final part of the method, on lines 0492 to 0495, disposes of the window which removes it from the desktop pane and updates the *Windows* main menu *Windows...* cascading menu via a call of the *menuBar populateWindowsMenu()* method. If the closure of this window means that there are no longer any open windows, this will result, as explained above, in the first four options of the *Windows* main menu becoming disabled.

The other two controls on the *Windows* menu, the two check boxes at the bottom, are configured every time the menu is posted by a call of the *brewser setWindowPostState()* called from the *menuSelected()* method mandated by the MenuListener interface. The implementation of this listener method is as follows.

```
0116      public void menuSelected( MenuEvent event) {
0117
0118      String  name   = ((Component)
0119                            event.getSource()).getName();
0120
0121         if ( name.equals( "window")) {
0122            this.setWindowPostState();
0123         } else if ( name.equals( "file")) {
0124            menuBar.setFileMenuOptions(
0125                       windowController.anyWindowsOpen());
0126         } // End if.
0127      } // End menuSelected.
```

The *setWindowPostState()* method sets the state of the two check box controls according to the state of the *currentlyActive* window, as follows.

```
0348      private void setWindowPostState() {
0349         if ( !windowController.anyWindowsOpen()) {
0350            menuBar.setWindowOptionsOff();
0351         } else {
0352            if ( currentlyActive.isRawPaneShowing()) {
0353               menuBar.setShowCheckState( true, true);
0354            } else {
0355               menuBar.setShowCheckState( false, true);
0356            } // End if.
0357
0358            if ( currentlyActive.isRawPaneShowing()) {
0359               if ( currentlyActive.isTied()) {
0360                  menuBar.setLinkCheckState( true, true);
0361               } else {
0362                  menuBar.setLinkCheckState( false, true);
0363               } // End if.
0364            } else {
0365               menuBar.setLinkCheckState( false, false);
0366            } // End if.
0367         } // End if.
0368      } // End if.
```

If the *windowController* reports that there are no windows open then, on line 0350, both check boxes are cleared and disabled. Otherwise, if the *rawPane* is showing in the *currentlyActive* window the *Show Text* control is enabled and checked; or enabled and unchecked if the pane is not showing. In the second part of the method, starting on line 0358, the *Link Text* control is cleared and disabled if the *rawPane* is not showing. Otherwise, on lines 0359 to 0363, it is enabled and checked or unchecked depending upon the current state of the *currentlyActive* window's *tied* attribute.

These two check boxes will dispatch ActionEvents to the *brewser*'s *actionPerformed()* method when the user checks or unchecks them. The detection of these events and the methods called to effect the user's request are as follows.

```
0104        } else if ( command.equals( "showText")) {
0105           this.showText();
0106
0107        } else if ( command.equals( "linkText")) {
0108           this.linkText();

0327      private void showText() {
0328         if ( windowController.anyWindowsOpen()) {
0329            if ( currentlyActive.isRawPaneShowing()) {
0330               currentlyActive.hideRawPane();
0331            } else {
0332               currentlyActive.showRawPane();
0333            } // End if.
0334         } // End if.
0335      } // End showText
0336
0337      private void linkText() {
0338         if ( windowController.anyWindowsOpen())) {
0339            if ( currentlyActive.isTied()) {
0340               currentlyActive.unTiePanes();
0341            } else {
0342               currentlyActive.tiePanes();
0343            } // End if.
0344         } // End if.
0345      } // End linkText
```

The *showText()* and *linkText()* methods toggle the state of the appropriate attribute of the *currentlyActive* window, so long as the *windowController* indicates that there is at least one window open. As these controls are enabled and configured every time the *Windows* menu is posted there is no need for the two methods to be any more complex.

The *menuSelected()* method, given above, will also set the state of the *File* menu as it is posted, by calling the *menuBar setFileMenuOptions()* on lines 0124 and 0125. If there are no windows open the *windowController anyWindowsOpen()* method will return **false**, which will result in the *File* menu being configured as shown in the left-hand image in Figure 9.2. Otherwise a **true** value, indicating at least one open window, will produce the appearance shown in the right-hand image.

The *Iconify all* and *DeIconify all* items on the *Windows* menu dispatch ActionEvents which are intercepted by the *brewser actionPerformed()* method as follows.

```
   } else if ( command.equals( "iconifyAll")) {
      this.iconifyAll( true);
   } else if ( command.equals( "deiconifyAll")) {
      this.iconifyAll( false);
```

The *iconifyAll()* method is implemented, with minimal comment, as follows. It obtains an Enumeration of all the window identities from the *windowController* and then iterates through the enumeration calling the JInternalFrame setIcon() method to iconify or deiconify the window as appropriate.

```
private void iconifyAll( boolean yesOrNo) {

Enumeration   allWindows = windowController.getAllWindows();
BrewserWindow thisWindow = null;

   try {
      while ( allWindows.hasMoreElements()) {
         thisWindow =
                   (BrewserWindow) allWindows.nextElement();
         thisWindow.setIcon( yesOrNo);
      } // End while.
    } catch ( PropertyVetoException exception) {
      // Do nothing.
   } // End try/catch.
} // End iconifyAll.
```

The implementation of the *Arrange* action is similar and presented without comment as follows.

```
   } else if ( command.equals( "arrange")) {
      this.arrangeAll();

private void arrangeAll() {

Enumeration   allWindows = windowController.getAllWindows();
BrewserWindow thisWindow = null;
int           thisWindowX = 0;
int           thisWindowY = 0;
int           xStep       = 50;
int           yStep       = 50;

      while ( allWindows.hasMoreElements()) {
         thisWindow =
                  (BrewserWindow) allWindows.nextElement();
         thisWindow.setLocation( thisWindowX, thisWindowY);
         thisWindowX += xStep;
         thisWindowY += yStep;
      } // End while.
} // End arrangeAll.
```

The effect of calling this method is to arrange all the windows in a left– right diagonal starting at the upper left of the desktop pane. The effectiveness of this method and the provision of alternative layouts will be left as an end of chapter exercise. Finally, in this part of the chapter, each of the items on the *Window* menu *Windows...* cascading menu will dispatch an ActionEvent containing the actionCommand *switchWindow*, which is handled within the *brewser actionPerformed()* method as follows.

```
  } else if ( command.equals( "switchWindow")) {
     this.setActiveWindow(
       ((JMenuItem) event.getSource()).getText());
```

This section of the chapter has illustrated a minimal set of controls and techniques for the opening, closing and manipulation of a set of internal windows. The end of chapter exercises will suggest some more complex controls which might also be provided. The next part of the chapter will describe the implementation of the remaining controls.

9.5 *Brewser*: loading, saving and printing

The facilities described in this part of the chapter rely upon the artifact having access to some of the local machine's resources; specifically reading from, writing to and creating files on the local drives and access to the printer. This may not always be allowed, particularly for applets loaded across the Internet, and may result in SecurityExceptions being thrown. The techniques for informing the SecurityManager object which artifacts can be allowed access to which resources are dependent upon the Java environment installed and are outside the scope of this book. For the remainder of this section it will be assumed that the *Brewser* has full access to all required resources and so no exceptions will be thrown.

When the user activates the *File* menu *Open File...* item an ActionEvent is dispatched to the *brewser*'s actionPerformed() method with the actionCommand *openFile*. This results in the *openFile()* method, as follows, being called.

```
private void openFile() {

JFileChooser        chooser     = null;
ExtensionFileFilter filter      = null;
int                 usersAction = 0;
File                fileChosen  = null;
String              fileName    = null;
boolean             loadedOK    = false;

  chooser = new JFileChooser();
  filter  = new ExtensionFileFilter();
  filter.addExtension(   "htm");
  filter.addExtension(   "html");
  filter.setDescription( "HTML files");
  chooser.addChoosableFileFilter( filter);

  usersAction = chooser.showOpenDialog( this);

  if ( usersAction == JFileChooser.APPROVE_OPTION) {
     fileChosen = chooser.getSelectedFile();

0219            try {
0220               fileName = fileChosen.toURL().toString();
0221               this.openNewWindow( fileName);
0222               loadedOK = true;
0223            } catch ( HTMLUtilitiesException exception) {
0224               loadedOK = false;
0225            } catch ( MalformedURLException  exception) {
0226               loadedOK = false;
0227            } // End try/catch.
0228
0229            if ( !loadedOK ) {
0230               JOptionPane.showMessageDialog( this,
0231                                     "The File \n" + fileName +
0232                                     "\ncould not be opened!",
0233                                     "File Open Failure",
0234                                    JOptionPane.ERROR_MESSAGE);
0235            } // End if.
0236         } // End if.
0237       } // End openFile.
```

The method commences, on line 0206, by constructing an instance of the JFileChooser class called *chooser*. After configuring the *chooser*, on lines 0207 to 0211, it is presented to the user by a call of its showOpenDialog() method, on line 0214. As with the JOptionPane dialogs this is a modal dialog and the showOpenDialog() method blocks until the user has responded. The appearance of the *chooser* dialog when presented to open a file, with the Metal and Windows look and feels, is presented in Figure 9.10.

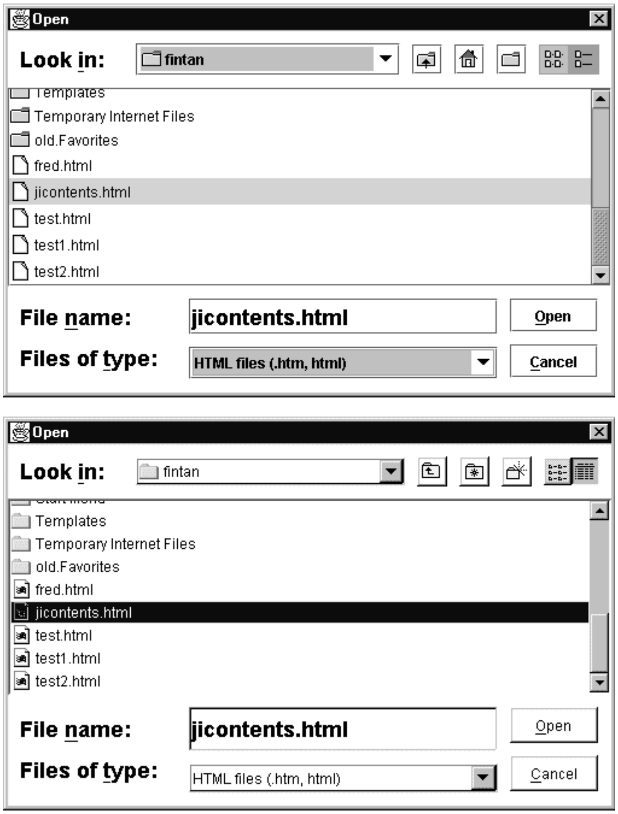
Users can use the dialog to explore the local file system and select a file to be opened, confirming the action by pressing <ENTER> in the text input field or by pressing on the *Open* button. Alternatively they can dismiss the dialog by pressing on the *Cancel* button. In the *openFile()* method the pressing of the *Open* button is detected on line 0216 and the *fileChosen* is obtained by using the *chooser*'s getSelectedFile() method on line 0217.

Having obtained the user's choice as a File a URL format String, called *fileName*, is prepared on line 0220 and passed to the *brewser*'s *openNewWindow()* method, as described in the previous section. If neither of these steps throws an exception the result will be that the window containing the file identified will become the active window (if it is already loaded) or a new window containing the HTML contents of the file indicated will appear (if it is not already loaded).

Should either of the steps on lines 0220 and 0221 throw an exception it is handled, on lines 0223 to 0227, by setting the value of the local flag *loadedOK* **false**. If no exception is thrown the value of the flag is set **true** on line 0222. The method concludes on lines 0229 to 0236 by posting a JOptionPane error dialog informing the user that the file could not be opened if the *loadedOK* flag indicates that this is the case.

JComponent → javax.swing.event.JFileChooser

```
public JFileChooser()
public JFileChooser( String defaultDirectory)

public FileFilter getAcceptAllFileFilter()
public FileFilter getFileFilter()
```



Look in: fintan

Templates
Temporary Internet Files
old.Favorites
fred.html
jicontents.html
test.html
test1.html
test2.html

File name: jicontents.html

Files of type: HTML files (.htm, html)

Look in: fintan

Templates
Temporary Internet Files
old.Favorites
fred.html
jicontents.html
test.html
test1.html
test2.html

File name: jicontents.html

Files of type: HTML files (.htm, html)

Figure 9.10 *JFileChooser* open dialog with Metal and Windows look and feels.

```
public void      setFileFilter( FileFilter filter)
public void      addChoosableFileFilter(    FileFilter filter)
public void      removeChoosableFileFilter( FileFilter filter)
```

Methods to manipulate the list of FileFilters.

```
public void    setDialogTitle( String dialogTitle)
public String  getDialogTitle()
public boolean isMultiSelectionEnabled()
public void    setMultiSelectionEnabled( boolean yesOrNo)
public int     getFileSelectionMode()
public void    setFileSelectionMode( int mode)
```

Methods to configure a file dialog prior to showing it. The multipleSelection attribute determines if more than one file can be selected and selectionMode can take one of the values FILES_ONLY, DIRECTORIES_ONLY, FILES_AND_DIRECTORIES (default).

```
public int  getDialogType()
public void setDialogType( int dialogType)
public int  showDialog( Component parent, String approveText)
public int  showOpenDialog( Component parent)
public int  showSaveDialog( Component parent)
```

Methods to show the dialog and configure what type of dialog is shown. The dialogType attribute can take one of the values OPEN_DIALOG, SAVE_DIALOG or CUSTOM_DIALOG.

```
public File   getSelectedFile()
public void   setSelectedFile( File toSelect)
public File[] getSelectedFile()
public void   setSelectedFiles( File[] toSelect)
public File   getCurrentDirectory()
public void   setCurrentDirectory( File directory)
```

Methods to obtain or set details about the directory and file(s) selected.

The configuration of the JFileDialog, on lines 0207 to 0211, involves the construction and addition of a ExtensionFileFilter to it. A FileFilter will determine which files in the directories being explored will be presented to the user and the list of available filters is presented to the user in the *Files of type* drop-down list at the bottom of the dialog. By default a file dialog will show all files in a directory unless a filter is applied. In this example a filter is supplied which will cause only those files with extensions `htm` or `html` to be displayed.

Object → FileFilter → javax.swing.filechooser.JFileFilter

```
public JFileFilter()
public JFileFilter( String extension)
public JFileFilter( String extension,   String description)
public JFileFilter( String[] extensions)
public JFileFilter( String[] extensions, String description)
```

Constructors allowing various combinations of extension(s) and description to be specified.

```
public String getDescription()
public void   setDescription( String description)
public void   addExtension(   String newExtension)
```

Methods to support the attributes.

The ActionEvents dispatched by the *File* menu *Save HTML...* and *Save Plain...* options are dispatched by the *brewser*'s *actionPerformed()* method, using the **private** manifest constants, as follows.

```
      } else if ( command.equals( "saveHTML")) {
         this.saveFile( HTML_FORMAT);
      } else if ( command.equals( "savePlain")) {
         this.saveFile( PLAIN_FORMAT);
```

The saving of a document is somewhat more complex than the loading of a document. Figure 9.11 presents the STD for the *save interface* which is implemented by the *saveFile()* method. The simplest path through the STD is for the user to cancel the operation when the *save* dialog is presented, and the transition shown to the right of the *save* dialog leads directly to the terminal state. The next simplest path would be for the user to confirm the operation, for a file with the file name indicated by the user not to exist already and for the

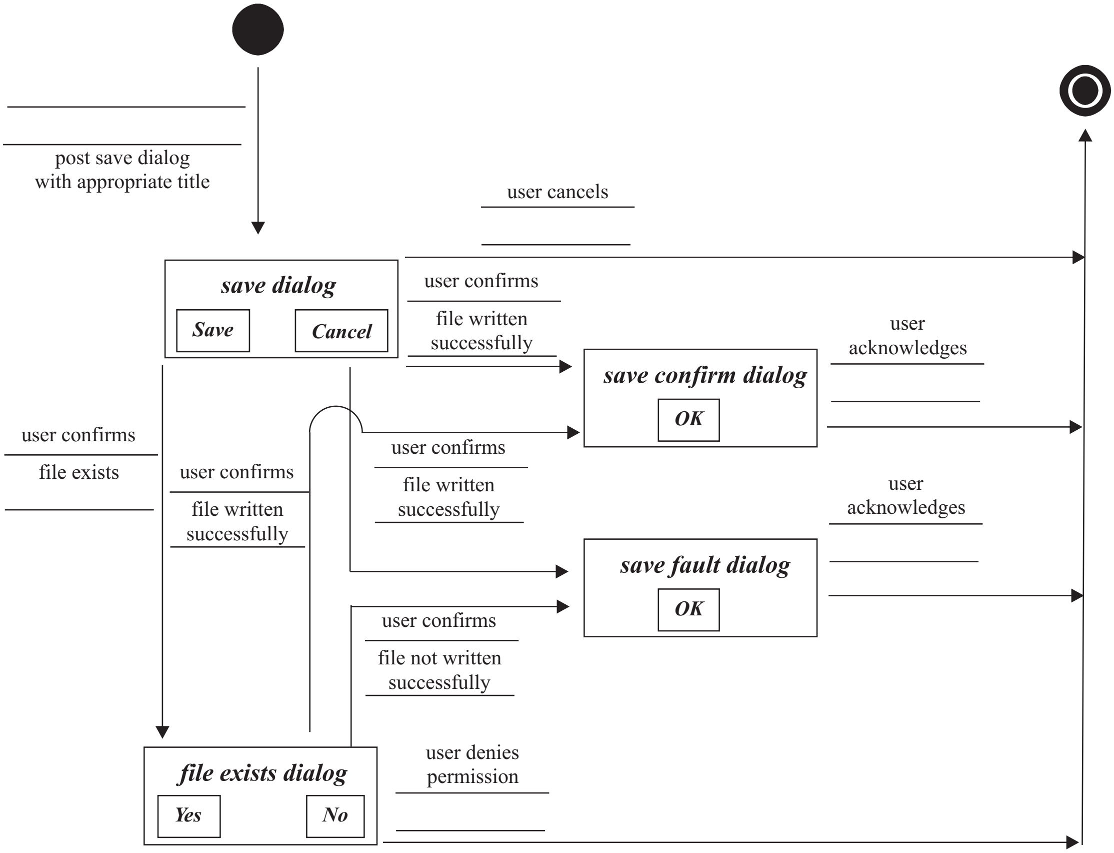


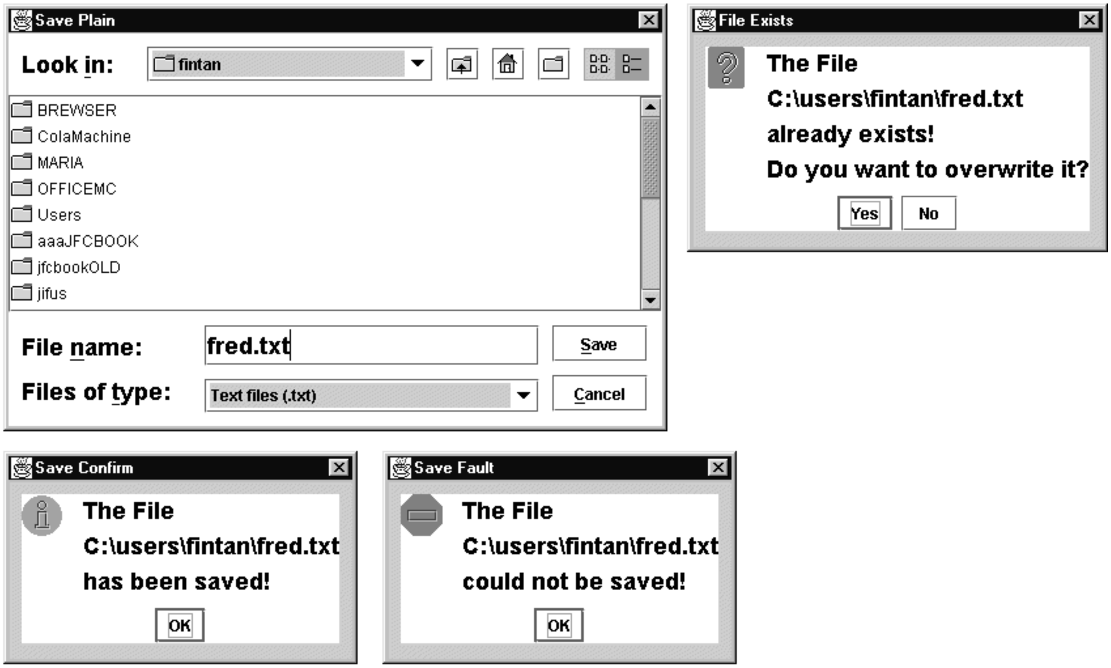
Figure 9.11 *Brewser* save STD.

file to be written successfully. This results in a *save confirm* dialog, which confirms the success of the operation and, once the user acknowledges it, leads to the terminal state.

If the user confirms the *save* operation and a file with the filename entered or selected exists, the *file exists* dialog is presented for the user to confirm overwriting the existing file. If the user does not give permission for the overwriting the transition will lead to the terminal state. Otherwise, if the overwriting is confirmed and the save is successful the transition also leads to the *save confirm* dialog. From either the *save* or the *file exists* dialog states, if the attempt to save the file is not successful the *save fault* dialog is presented to inform the user of this. Once the user has acknowledged the failure the transition leads to the terminal state. The appearance of the four dialogs required by this interface is illustrated in Figure 9.12.

The first part of the *saveFile()* method is as follows.

```
0242     private void saveFile( int saveFormat) {
0243
0244     String       toSave        = null;
0245     JFileChooser chooser       = null;
0246     ExtensionFileFilter filter = null;
0247     int          usersAction   = 0;
0248     File         fileChosen    = null;
```



File name: fred.txt

Files of type: Text files (.txt)

The File
C:\users\fintan\fred.txt
already exists!
Do you want to overwrite it?

The File
C:\users\fintan\fred.txt
has been saved!

The File
C:\users\fintan\fred.txt
could not be saved!

Figure 9.12 *Brewser* save dialogs.

```
0249      String       fileName      = null;
0250      PrintWriter  writeStream   = null;
0251      boolean      clearToWrite  = false;
0252      int          writeReply    = -1;
0253
0254        chooser = new JFileChooser();
0255        if ( saveFormat == HTML_FORMAT) {
0256           chooser.setDialogTitle( "Save HTML");
0257        } else {
0258           chooser.setDialogTitle( "Save Plain");
0259        } // End if.
0260
0261        filter = new ExtensionFileFilter();
0262        filter.addExtension(   "htm");
0263        filter.addExtension(   "html");
0264        filter.setDescription( "HTML files");
0265        chooser.addChoosableFileFilter( filter);
0266        filter  = new ExtensionFileFilter();
0267        filter.addExtension(   "txt");
0268        filter.setDescription( "Text files");
0269        chooser.addChoosableFileFilter( filter);
0270
0271        usersAction = chooser.showSaveDialog( this);
```

The use made of the local variables will be described, as they are introduced in the body of the method. The first step, after constructing the JFileChooser *chooser*, on lines 0255 to 0259, is to install a suitable dialogTitle into the file dialog, determined by the *saveFormat* argument to the method. The method continues, on lines 0261 to 0269, by installing an HTML and text file filter into the dialog before it is presented as a save dialog on line 0271. The implementation of the method continues as follows.

```
0273        if ( usersAction == JFileChooser.APPROVE_OPTION) {
0274           fileChosen = chooser.getSelectedFile();
0275           fileName   = fileChosen.toString();
0276           if ( fileChosen.canRead()) {
0277              writeReply = JOptionPane.showConfirmDialog( this,
0278                             "The File \n" + fileName +
0279                             "\nalready exists!\n"    +
0280                             "Do you want to overwrite it?",
0281                             "File Exists",
0282                  JOptionPane.YES_NO_OPTION );
0283              clearToWrite =
0284                        (writeReply == JOptionPane.YES_OPTION);
0285           } else {
0286              clearToWrite = true;
0287           } // End if.
0288
0289           if ( clearToWrite) {
0290              if ( saveFormat == HTML_FORMAT) {
0291                 toSave = currentlyActive.getRawHTML();
0292              } else {
0293                 toSave = currentlyActive.getPlainText();
0294              } // End if.
```

The **if** structure commencing on line 0273 extends to the end of the method and ensures that no further action is taken if the user presses the *save* dialog *Cancel* button, as indicated on the STD. Otherwise, on lines 0274 and 0275, the *fileChosen* by the user is obtained and its *fileName* extracted from it. Line 0276 uses the *fileChosen* canRead() method to decide whether or not the *fileChosen* exists, on the assumption that if it can be read it exists and can also be written.

If the file does not exist the **else** branch of the structure, on line 0286, indicates that it is *clearToWrite* the file. If the file does exist, lines 0277 to 0284 present the user with the *file exists* dialog and set the value of *clearToWrite* according to the user's reply. If the user does not confirm the overwriting, by not pressing the *Yes* button on the *file exists* dialog, the **if** structure commencing on line 0289 will ensure that no further steps are taken, again in accord with the STD. If it is *clearToWrite* the last part of this fragment obtains the *rawHTML* or the *plainText* contents of the *currentlyActive* window storing them in the *toSave* string. The difference between the two formats is that the *rawHTML* contains the entire contents of the HTML document whereas the *plainText* contains only the contents of the body of the HTML document with all tags removed. The *saveFile()* method concludes as follows.

```
0296              try {
0297                 writeStream = new PrintWriter(
0298                           new FileOutputStream( fileChosen));
0299                 writeStream.print( toSave);
0300                 writeStream.close();
0301                 JOptionPane.showMessageDialog( this,
0302                                    "The File \n" + fileName +
0303                                    "\nhas been saved!",
0304                                    "Save Confirm",
0305                             JOptionPane.INFORMATION_MESSAGE );
0306              } catch ( IOException exception) {
0307                 JOptionPane.showMessageDialog( this,
0308                                    "The File \n" + fileName +
0309                                    "\ncould not be saved!",
0310                                    "Save Fault",
0311                                   JOptionPane.ERROR_MESSAGE);
0312              } // End try/catch
0313           } // End if clear to write.
0314        } // End if - save dialog confirmed.
0315      } // End saveFile.
```

The first part of this fragment, on lines 0297 to 0300, constructs a PrintWriter stream connected to the *fileChosen* and then writes the contents of *toSave* to it, after which it is closed. If this does not throw any exceptions, lines 0301 to 0305 present the *save confirm* dialog to the user and, when attended to, this path through the method terminates. Otherwise, if an exception is thrown, lines 0307 to 0311 present the *save fail* dialog to the user before the method terminates.

The implementation of this method can be validated against the STD by following all possible paths in the STD through the implementation and ensuring that all steps in the implementation have been followed at least once.

The only *brewser* actions remaining to be described are the *Print* options on the *File* menu. The ActionEvents dispatched from these items are handled by the *brewswer*'s *actionPerformed()* method as follows.

```
0082        } else if ( command.equals( "printHTML")) {
0083           this.printFile( HTML_FORMAT);
0084        } else if ( command.equals( "printRaw")) {
0085           this. printFile( RAW_FORMAT);
0086        } else if ( command.equals( "printPlain")) {
0087           this. printFile( PLAIN_FORMAT);
```

The first part of the implementation of the *printFile()* method is as follows.

```
0318      private void printFile( int printFormat) {
0319
0320      String      toPrint      = null;
0321      JFrame      printFrame   = null;
0322      JEditorPane aPane        = null;
0323      JTextArea   anArea       = null;
0324      PrintJob    printJob     = null;
0325      Graphics    printContext = null;
0326
0327         if ( printFormat == PLAIN_FORMAT) {
0328            toPrint = currentlyActive.getPlainText();
0329         } else {
0330            toPrint = currentlyActive.getRawHTML();
0331         } // End if.
0332
0333         printFrame = new JFrame();
```

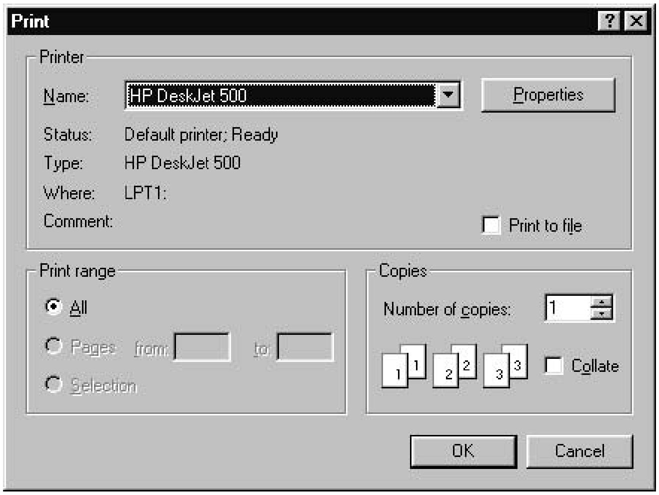
The use made of the local variables will be explained as their role is described in the implementation. This first step in the method is to obtain either the *plainText* or the *rawHTML* contents of the *currentlyActive* window, and this fragment concludes by creating a JFrame called *printFrame*. The image to be printed will be rendered into this frame, which will remain invisible, and will be printed from it, not from the on-screen visible image. The reason for this is that the invisible image can be resized to match the dimensions of the paper without disturbing the user, which would be the case if it were printed from the on-screen image. The method continues as follows.

```
0335          if ( printFormat == HTML_FORMAT) {
0336             aPane = new JEditorPane();
0337             aPane.setEditable( false);
0338             aPane.setContentType( "text/html");
0339             aPane.setText( toPrint);
0340             printFrame.getContentPane().add( aPane);
0341          } else {
0342             anArea = new JTextArea( toPrint);
0343             printFrame.getContentPane().add( anArea);
0344          } // End if.
```

If the user has asked for the rendered HTML to be printed, lines 0336 to 0340 install the *rawHTML* into a suitable configured JEditorPane instance and add it to the *printFrame*. Otherwise, either the *rawHTML* or the *plainText* will be installed into a JTextArea which is added to the *printFrame*. At the end of this fragment the frame will contain either the fully rendered HTML, or the HTML presented as raw text or the plain text, depending upon the value of the *printFormat* argument. The concluding part of the method, as follows, actually prints the frame's contents.

```
0348          printJob = this.getToolkit().getPrintJob(
0349               printFrame, currentlyActive.getTitle(), null);
0350          if ( printJob!= null) {
0351             (RepaintManager.currentManager( this)).
0352                        setDoubleBufferingEnabled( false);
0353             printFrame.setSize( printJob.getPageDimension());
0354 
0355             printContext = printJob.getGraphics();
0356             while( printContext != null) {
0357                printFrame.print( printContext);
0358                printContext.dispose();
0359                printContext = printJob.getGraphics();
0360             } // End while.
0361 
0362             printJob.end();
0363            (RepaintManager.currentManager( this)).
0364                         setDoubleBufferingEnabled( true);
0365          } // End if.
0366          printFrame.dispose();
0367       } // End printFile.
```

The call of the Toolkit getPrintJob() method, on lines 0348 and 0349, will post a modal blocking minimal system print dialog to the user. The appearance of the dialog under Windows 98 is illustrated in Figure 9.13. The arguments to getPrintJob() are the frame to



Print

Printer

Name: HP DeskJet 500

Status: Default printer; Ready

Type: HP DeskJet 500

Where: LPT1:

Comment:

□ Print to file

Print range

● All

○ Pages from: to:

○ Selection

Copies

Number of copies: 1

□ Collate

Figure 9.13 *Brewser*: standard Windows 98 print dialog.

display the dialog within, the name of the print job which will be used to identify the job in the system print queue and a printer-specific set of flags, which can be specified as **null**. If the user confirms the print operation the method will return a PrintJob instance which is referenced by *printJob*, or **null** if the user does not confirm the operation.

Object → java.awt.PrintJob

```
public PrintJob()
```

PrintJobs are not normally constructed, but obtained via a call of the Toolkit getPrintJob() method.

```
public PrintJob getPrintJob( Frame showWithin, String jobName,
                             Properties properties)
```

The Toolkit getPrintJob() method arguments are the frame to show a system print dialog, the name of the job to identify it in the print queue and printer specific properties.

```
public Dimension getPageDimension()
public Graphics  getGraphics()
public void end()
```

The getPageDimension() method will indicate the extent of a printer page in pixels. The getGraphics() method can be called repeatedly and will supply a Graphics object for each page in sequence, returning **null** when all pages have been printed. The end() method should be called when all pages have been printed so that any printer, or spooler, resources, can be released.

If the user confirms the print operation, the first step, on lines 0351 and 0352, is to turn double buffering off for the duration of the print operation. If double buffering is switched on then the image will be sent to the printer as a very large image file; if it is switched off then a more efficient format may be used. As the *printJob* is now established, on line 0353, the size of the *printFrame* can be set to the dimension of the printer's page, obtained via a call of the *printJob* getPageDimension() method.

Printing is accomplished by calling the *printFrame*'s paint() method passing a Graphics instance obtained from the *printJob*. This implies that no special techniques need to be supplied for printing a specialized component, as the *paint()* method provided to render the component on the screen will be used. However, it is possible that the extent of the printed image will extend over more than one page. To allow for this a series of Graphics instances can be requested from the *printJob*, each of which will print a separate page; and a **null** Graphic object will be returned when all pages have been printed.

These considerations inform the implementation of the **while** loop on lines 0355 to 0360. The call of the *printContext* dispose() method on line 0358 will flush the page to the printer. Once all the pages have been printed the method concludes, on lines 0363 to 0366, by ending the *printJob*, which releases the printer or the printer queue, re-enabling double buffering and disposing of the *printFrame* and hence all of its contents.

This is a minimal print routine which can be adapted for the printing of other components. However, a printing operation can take a significant amount of time, and so should be performed on a separate thread of control. A more serious limitation is that the page format is fixed, and control over such aspects as headers and footers and page numbers cannot be obtained. The java.awt.print package contains a collection of classes that address these limitations, but whose complexity makes then beyond the scope of this book.

This concludes the implementation of the *Brewser* artifact. It has again illustrated the complexity of designing and implementing a more realistic artifact and should give some indication of the amount of effort that would be required for a production-quality implementation. However, the fundamentals of the techniques for managing multiple internal windows, loading and saving information to and from an artifact, and printing have been introduced. These techniques can be adapted for use with other artifacts that require these capabilities.

Summary

- ♦ Multiple Internal Windows are supplied by the JInternalFrame class and have to be installed into a JDesktopPane instance.
- ♦ JInternalFrame windows dispatch InternalFrameEvents to various InternalFrame EventListener methods as they are interacted with.
- ♦ The JFileChooser class supplies a modal file dialog which allows, subject to security considerations, the host machine's file system to be explored.
- ♦ The ExtensionFileFilter class can be used in conjunction with the JFileChooser class to restrict the types of file which are shown.
- ♦ The PrintJob class supplies a modal system print dialog which allows the user to select a particular printer and set options on it.
- ♦ The Graphics instances obtained from a PrintJob instance can be used to send information to a printer by supplying them as arguments to existing *paint()* methods.
- ♦ More sophisticated printing capability is contained within the java.awt.print package.

Exercises

9.1 Extend the *Window* menu *Arrange* item so that it posts a cascading menu with different options, e.g. cascaded, tiled, horizontal or vertical.

9.2 Revisit the *HTMLEditor* artifact from Chapter 8 and provide it with the capability of loading and saving the HTML documents being edited.

9.3 The load facility as implemented in this chapter has no memory of where the last file was loaded. Every time it is presented it will show the user's home directory, and this may entail a complex navigation sequence to retrace the path to the last directory from which a file was loaded. Design, implement, demonstrate and test a memory facility for the artifact.

9.4 The opening of an HTML document in a separate window is not the usual behavior of a browser. Implement an *Options* menu which supplies a control to allow the user to turn this behavior on and off.

9.5 The loading of an HTML page can be a time-consuming process. Implement a specialized dialog which will confirm to the user that a page is being loaded, and give some indication of progress.

9.6 Implement a 'pages visited' panel that contains a JComboBox instance allowing the user to review which pages have been visited and return to one by choosing it from the list. Supply an additional control on the *Options* menu to show and hide this capability.

9.7 Implement a button control panel which contains a home page button, next visited and last visited button and a specialized dialog panel to allow the user to enter a home page URL. Add a control to the *Options* menu to allow this panel to be shown and hidden.

9.8 Critically review the controls provided and capabilities of established browsers and implement versions of the *brewser* that contain, or do not contain, a particular capability. Conduct a usability investigation to attempt to determine whether these features are actually used and appreciated by end users.

10

Extending JFC components as JavaBeans

10.1 Introduction

The specialized components that have been introduced in previous chapters have been engineered with sufficient regard to their intended use but only to the point where they can illustrate the considerations of developing specialized components. In this chapter another specialized component, called *DynaLabel*, will be presented. It will again only be engineered to the point where it can be used to illustrate some additional design considerations. As has been shown, the development of specialized components is not a trivial task and the investment involved in producing an extension to the JFC classes should be made so as to maximize the potential for its successful reuse. This can be assured by designing and implementing the extended class so as to satisfy *JavaBean* requirements.

The JavaBeans specification, and the java.beans package, is intended to improve the chances of successful reuse by defining a *component architecture*. That is: a set of rules and protocols which, if followed and implemented, will allow the bean-compliant class to interoperate effectively with other bean-compliant classes. All of the AWT and JFC component classes are bean-compliant, so implementing extended components in a bean-compliant manner will increase their potential to be used effectively with the pre-supplied user interface components.

In addition to the improved capabilities for reuse and interoperability with other components a bean can easily be used within a bean-aware software development tool. To assist with the development of JavaBeans, Sun Microsystems provides a rudimentary Bean Development Kit (BDK) which includes an environment called *BeanBox* where beans being developed can be configured and demonstrated. In the latter part of this chapter the BeanBox will be used to demonstrate, and further develop, the extended *DynaLabel* class, which will be designed and implemented in the first part of the chapter. Details of how to obtain the BDK and a document defining the protocols, called the JavaBeans White Paper, are contained in the Appendix.

10.2 The *DynaLabel* class

In this section the initial implementation of an extended JFC Component, called *DynaLabel*, will be presented. A *DynaLabel*, like a JLabel, has no interaction with the user. It is used to display a short text message, but no icon, within its window. The message can be animated in one of three different ways. In the first animation style, known as *STATIONARY*, the message is just shown, centered within the available area. In the second style, known as *CONTINUAL*, the message repeatedly slides in from the right-hand side of the window and the slides out to the left. In the third style, known as *SLIDE_IN*, the message slides in from the left of the window until it is centered within the available space and then remains stationary for the same period of time it took to slide in; following this, the window clears and it slides in again. An incomplete class diagram for the *DynaLabel* class is given in Figure 10.1.

The inheritance relationship in Figure 10.1 indicates that the *DynaLabel* class extends the JComponent class, is a member of the *dynalabel* package of classes and that it implements the Runnable interface. In order to be able to interoperate effectively with other JFC classes the *DynaLabel* class has to extend one of the JFC classes, and the JComponent class is the most convenient one to extend. As instances of this class must animate the

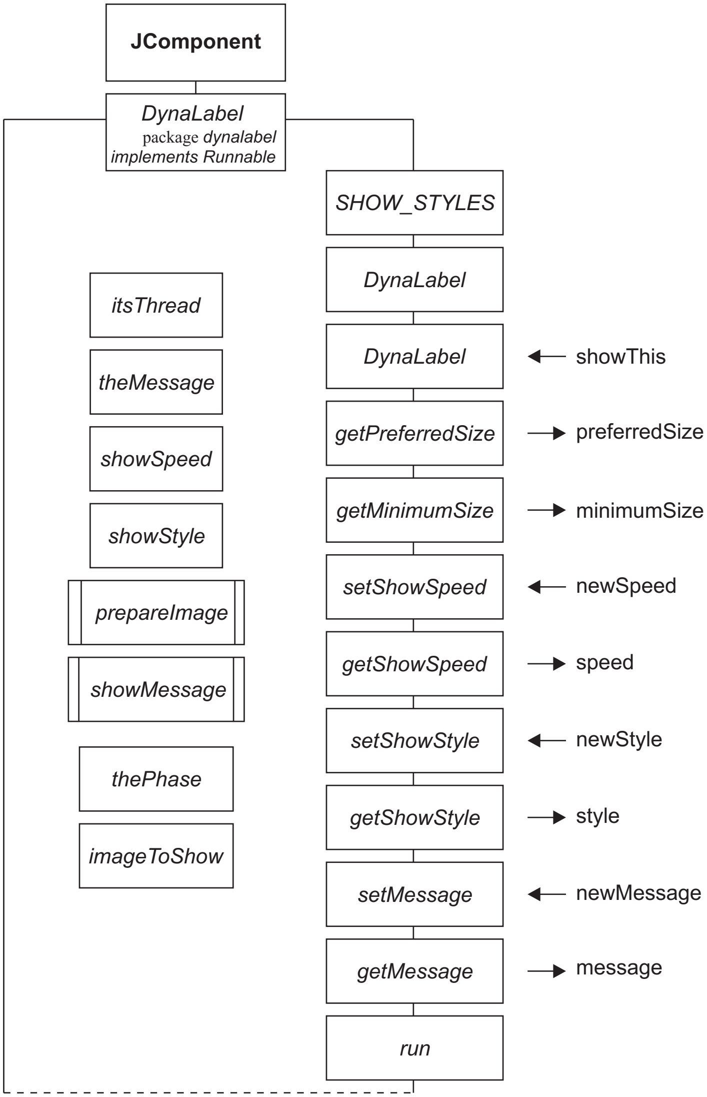


Figure 10.1 *DynaLabel* class diagram.

presentation of their messages they need to operate concurrently. In Chapter 1 the *Timer* class extended the Thread class in order to inherit concurrent capability, but this is not possible in this design, as the DynaLabel class has to extend a JFC class. Instead, the class implements the Runnable interface and encapsulates a Thread instance, shown as *itsThread*. The Runnable interface mandates a single method called run() and, using techniques which will be explained below, will arrange for this method to be executed concurrently.

The first public resource of the *DynaLabel* class is a set of manifest values (*STATIONARY*, *CONTINUAL*, and *SLIDE_IN*) which identify the animated behaviors of the class. The encapsulated attribute, *showStyle*, is provided to indicate the behavior and the pair of methods, *setShowStyle()* and *getShowStyle()*, support it. The non-default constructor takes a single argument, *showThis*, which identifies the *message* to be shown. The default constructor will call the non-default constructor, passing a default message to be shown. The *setMessage()* and *getMessage()* methods allow the message to be changed, or queried, after construction. The encapsulated *showSpeed* attribute determines the speed of the animation and the *setShowSpeed()* and *getShowSpeed()* methods support it.

The pair of methods *getPreferredSize()* and *getMinimumSize()* are required in order for instances of this class to take part in layout negotiations. When layout negotiations are taking place the layout manager will use these methods to determine how much space on the screen the component would prefer and what the minimum size it would be happy with is. However, there is no guarantee that an instance will be granted even its minimum requirements.

The class diagram also indicates that there are two private methods called *prepareImage()* and *showMessage()* and two further private attributes called *thePhase* and *imageToShow*. Figure 10.2 illustrates the techniques which are used by this class to animate the *message* onto the component's window.

The diagram shows that the *message* is first rendered onto the *imageToShow* attribute, which is an instance of the AWT Image class. Instances of this class can be thought of as off-screen windows where drawings can be prepared and subsequently transferred to the on-screen window. In this example the off-screen image is shown partially within the on-screen window at a location which is determined by *thePhase*. This, in turn, is incremented periodically by the *run()* method at a rate determined by the value of *showSpeed*. Every time that *run()* increments *thePhase* the window is redrawn with the visible location

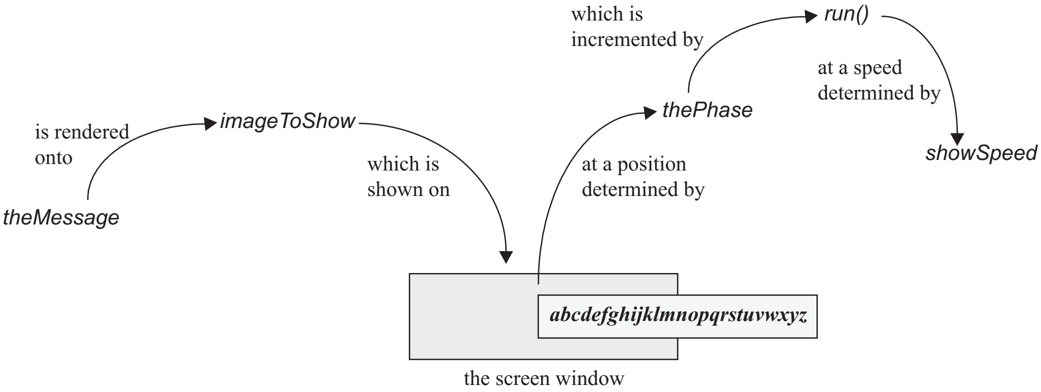


Figure 10.2 *DynaLabel*: operational overview.

of the off-screen image changed, causing the animation effect. The implementation of this design commences as follows.

```
0010  package dynalabel;
0011
0012  import javax.swing.*;
0013  import javax.swing.border.*;
0014  import java.awt.*;
0015
0016  public class DynaLabel extends JComponent
0017                    implements Runnable  {
0018
0019  public static final int  STATIONARY =0;
0020  public static final int  CONTINUAL  =1;
0021  public static final int  SLIDE_IN   =2;
0022
0023  private static final int START_PHASE = 0;
0024  private static final int END_PHASE   = 100;
0025  private static final int MID_PHASE   = 50;
0026
0027  private Thread   itsThread   = null;
0028  private String   message     = null;
0029  private Image    imageToShow = null;
0030
0031  private Dimension   optimalSize = null;
0032
0033  private int     showSpeed   = 10;
0034  private int     showStyle   = STATIONARY;
0035  private int     thePhase    = START_PHASE;
0036  private boolean clearWindow = true;
```

Following the package declaration and the necessary importations the class is declared on lines 0016 and 0017, exactly as indicated on the class diagram. Lines 0019 to 0021 then publicly declare the manifest values identifying the three possible animation styles. The three private manifest declarations on lines 0023 to 0025 are supplied in order to support the *thePhase* attribute. The value of *thePhase* will increment from *START_PHASE* to *END_PHASE* before being reset to *START_PHASE*, and then cycling again. The *MID_PHASE* value is provided to assist with knowing when the *message* is centered within the window.

Lines 0027 to 0029 then declare the encapsulated Thread instance, *itsThread*, the *message* String and the Image *imageToShow*. The *optimalSize* attribute, of the Dimension class, declared on line 0031 is supplied so that the *preferredSize()* and *minimalSize()* methods can reply to inquiries during layout negotiations. Dimension is an AWT class which has two public **int**eger attributes called width and height.

Object → java.awt.Dimension

```
public Dimension( int itsWidth, int itsHeight)
public int width
public int height
```

A utility class to indicate the extent of an area. Its two integer attributes, width and height, are fully public!

The *showSpeed*, *showStyle* and *thePhase* attributes are then declared, on lines 0033 to 0035, and initialized with suitable default values. The final attribute declaration, *clearWindow*, on line 0036 is a **boolean** flag used to indicate when the entire window should be cleared. The class declaration continues with its constructors, as follows.

```
0039     public DynaLabel()
0040        this( "This is a DynaLabel!");
0041     } // End default constructor.
0042
0043     public DynaLabel( String showThis) {
0044        super();
0045        message    = new String( showThis);
0046        itsThread  = new Thread( this);
0047        itsThread.start();
0048     } // End DynaLabel constructor.
```

The alternative constructor commences by calling its **super**, JComponent, constructor and then stores a copy of the *showThis* argument in the *message* attribute. It continues by constructing the Thread attribute *itsThread*, whose constructor takes as an argument the identity of a Runnable instance. Owing to this method of construction the thread becomes aware of the *run()* method which is to be executed, on a separate flow of control, when its start() method is called. In this example the identity of the Runnable instance specified is **this** *DynaLabel* instance, which is currently being constructed. Hence, when the *itsThread* start() method is called, on line 0043, it is the *run()* method that the *DynaLabel* class must provide in order to implement the Runnable interface, which will be executed. The default constructor indirects immediately to the alternative constructor, with a default message to be displayed.

java.lang.Runnable

```
public void run()
```

An interface which mandates a single method. When a Runnable instance is passed to a Thread constructor, it is the mandated run() method which will be executed concurrently after the Thread's start() method is called.

The next two methods to be declared by the *DynaLabel* class are *getPreferredSize()* and *getMinimumSize()*, which are implemented as follows.

```
0050     public Dimension getPreferredSize(){
0051        if ( optimalSize == null) {
0052        FontMetrics metrics = this.getFontMetrics(
0053                                         this.getFont());
0054        int preferredHeight = 2  * (metrics.getHeight());
0055        int preferredWidth  = 20 * (metrics.stringWidth( "m"));
0056
0057           optimalSize = new Dimension( preferredWidth,
0058                                        preferredHeight);
0059        } // End if.
0060         return optimalSize;
0061      } // End getPreferredSize.
0062
0063      public Dimension getMinimumSize(){
0064         return this.getPreferredSize();
0065      } // End getMinimumSize.
```

The *getMinimumSize()* method indirects to the *getPreferredSize()* method, so both of them will return the width and height, which are stored in the *optimalSize* Dimension attribute. This attribute is declared with the default value **null**, so, on line 0048, if the value is still **null** a width and height have yet to be decided upon. The method could just return a default value (for example a width of 300 and a height of 50 (pixels)), but it is preferable for it to make a decision based upon the amount of space that it might actually require. In this class the size requested should be related to the size of the characters which it is to display.

Information about the size of characters in a particular Font can be obtained from an instance of the *FontMetrics* class. Consequently the **if** structure commences, on lines 0049 and 0050, by retrieving the identity of the Font which will be used to render the message, using the getFont() method inherited from AWT Component, and using it as an argument to the getFontMetrics() method also inherited from AWT Component. Having obtained the *FontMetrics* associated with this component in *metrics*, its getHeight() method will return the height, in pixels, of each character glyph in the Font. There is no equivalent value for the width of any glyph, as some, for example 'm', are wider than others, for example 'i'. Accordingly a width can only be supplied if a String is provided to its getWidth() method.

In the *getPreferredSize()* method, on lines 0054 and 0055 the *preferredHeight* of the *DynaLabel* instance is determined to be twice the height of each character in the font and the *preferredWidth* to be 20 times the width of a single 'm'. There is no significance in the basis of these decisions; they were chosen so as to be related to the size of the glyphs in the Font which will be used to render the message. Having decided on the *preferredHeight* and *preferredWidth* of the instance they are used as arguments to the Dimension constructor on lines 0057 and 0058.

The effect of these methods is that when, during layout negotiations, a *DynaLabel* instance is asked for its preferred and minimum sizes, it will reply requesting a size based upon the font which it will use to display its message. However, there is no guarantee that it will be accorded even the minimum size requested: an individual component may be accorded any size, and should be constructed so as to take account of this.

The implementation of the *DynaLabel* class continues with the declaration of various state setting and inquiry methods, as follows.

```
0068      public synchronized void setShowSpeed( int newSpeed) {
0069         showSpeed = newSpeed;
0070      } // End setShowSpeed.
0071
0072      public int getShowSpeed() {
0073         return showSpeed;
0074      } // End getShowSpeed.
0075
0076
0077      public synchronized void setShowStyle( int newStyle) {
0078         showStyle   = newStyle;
0079         thePhase    = 0;
0080           imageToShow = null;
0081           clearWindow = true;
0082        } // End setShowStyle.
0083
0084        public int getShowStyle() {
0085           return showStyle;
0086        } // End getShowStyle.
0087
0088
0089        public synchronized void setMessage( String newMessage) {
0090           message  = new String( newMessage);
0091           this.setShowStyle( this getShowStyle());
0092        } // End setMessage.
0093
0094        public String getMessage(){
0095           return message;
0096        } // End getMessage.
```

The *setShowSpeed()*, *getShowSpeed()*, *getShowStyle()* and *getMessage()* methods contain no surprises, setting or returning the value of the appropriate attribute. The *setShowStyle()* method is more complex. When the *showStyle* attribute is changed the animation sequence must restart from the beginning, so, on line 0079, *thePhase* is reset to zero. In order to ensure that this is fully effected, any existing image in *imageToShow* is disposed of and the *clearWindow* flag set to cause the window to be completely cleared. As this method is changing the values of a number of important attributes of the instance it is declared with the **synchronized** modifier to make it thread-safe. The other state setting methods are similarly declared **synchronized**, as they are also changing the state of the instance.

The same set of modifications to the attributes must be effected if the message is changed with a call of *setMessage*, so, on line 0091, the *setShowStyle()* method is called with the existing *showStyle*, as indicated by *getShowStyle()*, after the *newMessage* has been stored in *message*.

The consequence of the changes to the attributes caused by *setShowStyle()* will become clear when the *prepareImage()* and *showMessage()* methods are described. However, the remaining public method, *run()*, will be described first; its implementation is as follows.

```
0181        public void run() {
0182
0183           while ( !this.isValid()) {
0184              try {
0185                 itsThread.sleep( 10);
0186              } catch (InterruptedException exception) {
0187                 // Do nothing.
0188              } // End try/catch.
0189           } // End while.
0190
0191           while( true) {
0192              try {
0193                 itsThread.sleep( this.getShowSpeed() * 10);
0194              } catch (InterruptedException exception) {
```

```
0195               // Do nothing.
0196            } // End try/catch.
0197
0198            if (++thePhase > END_PHASE) {
0199               thePhase = START_PHASE;
0200            } // end if.
0201
0202            this.showMessage();
0203         } // end while.
0204      } // End run.
```

The *run()* method's prototype, on line 0181, complies with that required for the *DynaLabel* class to be able to implement the Runnable interface. It will be called automatically, on a separate thread of control, as a consequence of the *itsThread* start() method being called during the *DynaLabel*'s construction, as explained above. This may cause it to be called before the component is fully visible on the screen, so the first part of its implementation, on lines 0183 to 0189, ensures that the window is fully visible before proceeding. The **boolean** isValid() method is inherited from the AWT Component class and will only return **true** when the component is both fully initialized and visible on the screen.

Object → java.awt.Component life cycle methods

```
public boolean isValid()
public void validate()
public void invalidate()
public boolean isVisible()
public void setVisible( boolean yesOrNo);
public boolean isShowing()
```

A component is initially shown on the screen, after becoming visible, in an invalid state and is validated by the layout manager.

The second part of the *run()* method is implemented as a non-terminating loop between lines 0191 and 0203. The first part of the loop, on lines 0192 to 0196, puts the thread to sleep for a period of time determined by the value of *showSpeed*, as indicated on Figure 10.2. The next step, on lines 0198 to 0200, increments the value of *thePhase*, constraining it between the values of *START_PHASE* and *END_PHASE*. The final step of the loop, on line 0202, is to call the *showMessage()* method, which will render the *message* onto the screen at a location determined by the value of *thePhase* and *showStyle*, as will be explained shortly.

Figure 10.2 shows that before the *message* can be shown on the window it must be rendered onto the off-screen Image *imageToShow*. This is accomplished by the *prepareImage()* method, as follows.

```
0099      private void prepareImage() {
0100
0101      FontMetrics metrics = this.getFontMetrics(
0102                                          this.getFont());
0103      int      imageWidth   = metrics.stringWidth( message);
0104      int      imageHeight  = metrics.getHeight();
0105      int      hbuffer      = (int) (imageWidth  * 0.1);
0106     int      vbuffer      = (int) (imageHeight * 0.2);
0107     Graphics imageContext = null;
0108
0109         imageToShow  = this.createImage(
0110                             imageWidth  + (2* hbuffer),
0111                             imageHeight + (2* vbuffer));
0112         imageContext = imageToShow.getGraphics();
0113         imageContext.setFont(  this.getFont());
0114         imageContext.setColor( this.getBackground());
0115
0116         imageContext.fillRect( 0, 0, imageWidth  + (2* hbuffer),
0117                                      imageHeight + (2* vbuffer));
0118
0119         imageContext.setColor( this.getForeground());
0120         imageContext.drawString( message, hbuffer,
0121                                  imageHeight + vBuffer);
0122         imageContext.dispose();
0123     } // End prepareImage.
```

An overview of this method is that it prepares an off-screen image which is a little higher and wider than the *message* which is to be rendered onto it, then fills the entire image with the background color of the component before rendering the *message* onto the image, using the component's foreground color. The implementation of the method commences, on lines 0101 and 0102, by obtaining the FontMetrics which are associated with **this** component, using the same techniques which were used in the *getPreferredSize()* method previously explained. Having obtained the metrics, lines 0103 and 0104 determine the width and height required for the *message* to be rendered. Lines 0105 and 0106 then add 10% to the width and 20% to the height to ensure that there is a border around the rendered text.

Line 107 declares, but does not construct, an instance of the Graphics class called *imageContext*. Graphics instances (known in some other environments as *graphics contexts*) can be thought of as drawing engines. Whenever some graphical output needs to be placed onto a *drawable*, a component's window or an Image, it has to be done using a Graphics context. The context encapsulates a large amount of information, including the color to be used, the font to be used and the drawing mode to be used, which consequently do not need to be explicitly specified when the context's drawing methods are used.

Object → java.awt.Graphics

```
public Graphics();

public Graphics create()
public Graphics create( int x, int y, int width, int height);

public void dispose()
```

Graphics instances are not normally constructed by calling the constructor but by using the getGraphics() method of java.awt.Component or java.awt.Image. The two create() methods make a copy of the Graphics instance, with output possibly constrained (clipped) to the area specified. Graphics instances should always be explicitly destroyed, by calling dispose(), as soon as they are no longer needed.

```
public void setColor( Color newColor)
public Color getColor()
public void setFont( Font newFont)
public Font getFont()
```

Attribute setting and inquiry methods for the Color and Font to be used.

```
public void setClip( int x, int y, int width, int height)
public void setClip( Shape toThis);
public Shape getClip()
```

A clip Shape (rectangle) is the area of the image or window to which drawing is constrained.

```
public void drawLine( int fromX, int fromY, int toX, int toY)
public void drawRect( int atX, int atY, int width, int height)
public void drawOval( int atX, int atY, int width, int height)
public void drawString( String toDraw, int baseX, int baseY)
public void drawImage( Image toDraw,
                       int atX, int atY, ImageObserver observer);
```

Various drawing operations which will produce output on the drawable.

The implementation of the method commences, on lines 0109 to 0111, with the construction of the *imageToShow*. The arguments to the createImage() method, which is inherited from the AWT Component class, are the width and height (in pixels) of the image to be created. Having constructed the Image instance, on line 0112, a Graphics context, called *imageContext*, is obtained for it by using its getGraphics() method. This context is then prepared, on lines 0113 and 0114 for use by setting its font and color resources to the font and background color resources of the *DynaLabel* instance.

Object → java.awt.Image

```
public Image()

public int getWidth(  ImageObserver observer)
public int getHeight( ImageObserver observer)
public Graphics getGraphics()
```

Image instances are not normally constructed by calling the Image() constructor but by using the createImage() method of java.awt.Component or java.awt.Toolkit. The two inquiry methods require an ImageObserver instance as an argument because the Image could be loading across the Internet and the *observer* can be interrogated to determine if loading is complete and has been successful. The java.awt.image.ImageObserver interface is implemented by Component and hence any AWT or JFC instance can be used. The getGraphics() method returns a Graphics instance which can be used to draw on the Image.

Lines 0116 and 0117 call the *imageContext*'s fillRect() method, the four arguments to this method are the upper left *x* and *y* coordinates and the width and height of the rectangle which is to be filled with the current color of the Graphics context. The actual arguments specified in this call are the upper left of the image and its entire width and height. The effect is to ensure that the *imageToShow* is completely filled with the *DynaLabel*'s background color.

The frame of reference for the Graphics coordinate system has the origin (0,0) at the top left of the drawable, with *x* values increasing to the right and *y* values increasing downwards.

Line 0119 then changes the color attribute of the *imageContext* to the *DynaLabel*'s foreground color so that the subsequent call of its drawString() method will render the text in the foreground color. The drawString() call, on lines 0120 and 0121, specifies the String to be drawn, *message*, and the *x*, *y* coordinates of the lower left of the rendered string. The effect is to render the *message* string onto the *imageToShow*, which has been previously constructed so as to be large enough to accommodate it, allowing for the horizontal and vertical buffer around it.

Graphics instances take a significant amount of system resources and it is regarded as good practice to explicitly dispose of them as soon as they are no longer required. This is accomplished in this method, on line 0122, by calling the *imageContext*'s dispose() method. The *prepareString()* method is called, if it is needed, at the start of the *showMessage()* method, whose implementation commences as follows.

```
0126    private void showMessage() {
0127
0128    Graphics context      =  this.getGraphics();
0129    Insets   insets       =  this.getInsets();
0130    int      windowWidth  =  this.getSize().width;
0131    int      windowHeight =  this.getSize().height;
0132
0133    float proportion =  1.0F/((float) END_PHASE) * thePhase;
0134
0135    int imageWidth  = 0;
0136    int imageHeight = 0;
0137
0138    int voffset     = 0;
0139    int hOffset     = 0;
0140
0141        if ( imageToShow == null) {
0142           this.prepareImage();
0143        } // End if.
0144
0145        imageWidth  = imageToShow.getWidth(  this);
0146        imageHeight = imageToShow.getHeight( this);
0147
0148        if ( clearWindow || (thePhase == START_PHASE)) {
0149           context.setColor( this.getBackground());
0150           context.fillRect( 0, 0, windowWidth, windowHeight);
0151           context.setColor( this.getForeground());
0152           this.paintBorder( context);
0153           clearWindow = false;
0154        } // End if.
0155
0156        graphics.setClip( insets.left, insets.top,
0157                    (windowWidth  - insets.right  - insets.left),
0158                    (windowHeight - insets.bottom - insets.top));
```

The method commences by declaring and obtaining a Graphics *context* which can be used to draw onto the component's window. Line 0129 then obtains the *insets* associated with the window. Insets describe the amount of space at each edge of the window that is reserved for its border. Lines 0130 and 0131 then determine the width and height of the window. The getSize() method is inherited from the AWT Component class and returns an instance of the Dimension class, whose public width and height attributes are accessed to obtain the information.

Line 0133 initializes the **float** *proportion* variable to a value between 0.0 and 1.0, depending upon the current value of *thePhase*. So if *thePhase* is equal to *START_PHASE*, *proportion* will have the value 0.0 and if it has the value *END_PHASE* it will have the value 1.0; intermediate values of *thePhase* will cause *proportion* to have a suitable intermediate value. Lines 0135 and 0136 declare **int**eger variables for the width and height of the *imageToShow*, but these cannot be initialized at this stage, as it is possible that the image has not yet been constructed. The local declarations conclude, on lines 0138 and 0139, with two **int**eger variables to record the vertical and horizontal offsets of the *imageToShow* relative to the window.

The first step of the implementation of the method, on lines 0141 to 0143, is to call the *prepareImage()* method if the *imageToShow* has a **null** value. This will be the case if this is the first time the method has ever been called or if the *setMessage()* or *setShowStyle()* methods have been called. The result is that, following this first step, it is certain that the *imageToShow* does reference an actual image, so lines 0145 and 0146 can initialize the values of the *imageWidth* and *imageHeight* local variables. The image's *getWidth()* method returns a Dimension instance whose public attributes are accessed as before. The argument to *getWidth()* is an instance of any class which implements the ImageObserver interface, and as the AWT Component class implements this interface any AWT or JFC component can be used. In this class the *imageToShow* has been completely constructed in the *prepareImage()* method, but in other situations it may have to be loaded across the Internet, on a separate thread of control, which can take some time. In these circumstances the ImageObserver can be interrogated to find out whether the image has completely loaded, but here this possibility can be safely ignored.

Lines 0148 to 0154 then clear the entire window if the *clearWindow* flag indicates that it is required, or at the start of every animation cycle as indicated by the value of *thePhase*. The clearing of the window to its background color, on lines 0149 to 0151, uses the same techniques which were used to clear the *imageToShow* to the component's background color in the *prepareImage()* method as previously described. Once the window has been cleared its border is redrawn by calling the *paintBorder()* method, inherited from the JFC JComponent class, passing the Graphics context as an argument. As the window has now been cleared the flag is reset to prevent it from happening again.

The final step in this part of the *showMessage()* method is to *clip* the Graphics *context* so that it cannot overdraw the border. By default the Graphics *context* can draw anywhere upon the window but its *setClip()* method can be used to restrict this to a specified rectangular area. The four arguments to the setClip() call, on lines 0156 to 0158, define the rectangle contained within the top, left, right and bottom *insets* which define the extent of the window's border. The remainder of the method is implemented as follows.

```
0160        if ( (this.getShowStyle()  == STATIONARY)  ||
            ((this.getShowStyle() == SLIDE_IN) &&
             (thePhase > MID_PHASE)            )   ){
         voffset = (int) ((1.0F/((float) END_PHASE) * MID_PHASE)
                         *  (float) (windowWidth + imageWidth));
       } else {
          voffset =  (int) ( proportion *
                            (float) (windowWidth + imageWidth));
       } // End if.

       hOffset =  (winHeight - imageHeight)/2;

       context.drawImage( imageToShow, winWidth - voffset,
                                       hoffset, this);

       context.dispose();
    } // End showMessage.
```

The first part of this fragment calculates the vertical offset (*voffset*) of the location of *imageToShow*. Figure 10.3 illustrates the considerations which are required for a *DynaLabel* whose *showStyle* is *CONTINUAL* when *thePhase* has the values *START_PHASE*, *MID_PHASE* and *END_PHASE*. The upper diagram illustrates the requirement when *thePhase* has the value *START_PHASE*, so *proportion* will have the value 0.0, and indicates that the left-hand location of *imageToShow* should be placed at the extreme right of the window. The lower diagram illustrates the requirement when *thePhase* has the value *END_PHASE*, so *proportion* will have the value 1.0, and indicates that the right-hand location of *imageToShow* should be placed at the extreme left of the window. Alternatively, this can be expressed as the left-hand side of *imageToShow* should be placed at a distance equal to the width of the image beyond the left of the window. Intermediate values of *thePhase* should cause the left-hand side of the image to be placed somewhere between the *x* location *windowWidth* and the *x* location minus *imageWidth*. The middle diagram also shows that when *thePhase* has the value *MID_PHASE* the *message* will be centered within the extent of the window.

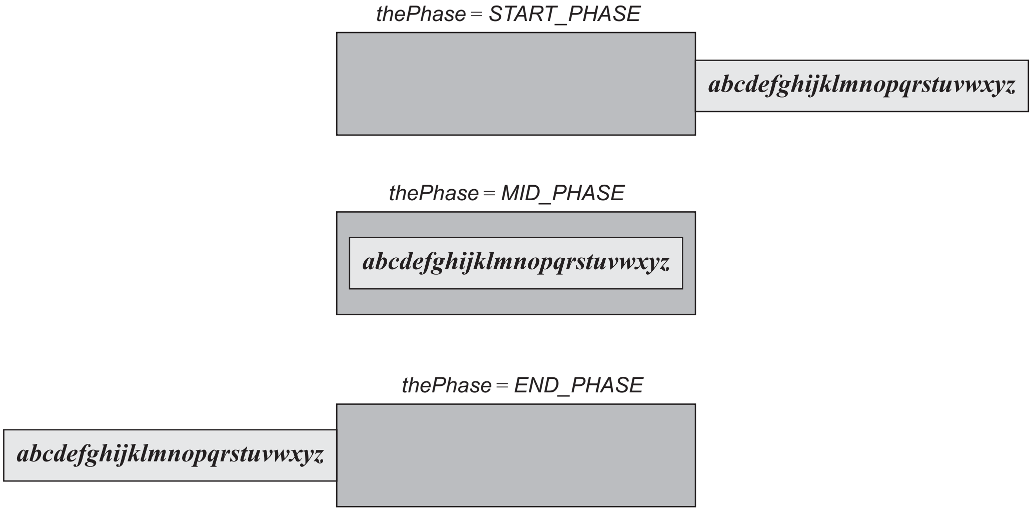

thePhase = START_PHASE
abcdefghijklmnopqrstuvwxyz
thePhase = MID_PHASE
abcdefghijklmnopqrstuvwxyz
thePhase = END_PHASE
abcdefghijklmnopqrstuvwxyz

Figure 10.3 *DynaLabel showMessage()* horizontal offset considerations.

Lines 0166 and 0167 achieve this requirement initializing the value of *voffset* to the appropriate *proportion* of the sum of the *windowWidth* and *imageWidth* values. However, if the *DynaLabel*'s *showStyle* is *SLIDE_IN* and *thePhase* is beyond the *MID_PHASE* then the image should remain centered within the window, and this is accomplished with the **if** structure condition on lines 0161 and 0162 and the alternative computation of *voffset* on lines 0163 to 0164. This version of the calculation is also appropriate at all times when the *showStyle* is *STATIONARY*, as indicated in the first term of the **if** condition on line 0160.

Having decided upon the vertical offset, line 0170 computes the horizontal offset (*hoffset*) before the *imageToShow* is drawn onto the window in lines 0172 to 0174. The four arguments to the Graphics *context* drawImage() method are the image which is to be copied, the *x* and *y* location where the upper left corner of the image is to be placed and an ImageObserver. So, for example, if *thePhase* has the value *START_VALUE*, *proportion* will have the value 0.0 and so *voffset* will have the value 0. Hence the upper left corner of *imageToShow* will be placed at *x* location *winWidth* and a *y* location causing it to be vertically centered. Figure 10.3 indicates that this is appropriate and as none of the image is contained within the *context*'s clip rectangle nothing will be drawn on the screen. Having copied the *imageToShow*, containing the *message*, onto the window in the appropriate location the final step of the *showMessage()* method is to dispose of the *context*.

Continuing the example started in the previous paragraph. The *showMessage()* method will be called again on the next iteration of the *run()* method's non-terminating loop with *thePhase* incremented from 0 to 1. This will cause the location where the left-hand side of the image is to be copied to move slightly to the left, and on the next iteration more to the left, and so on. After a few iterations the image will be within the bounds of the clip rectangle and the leftmost part of the image will start to appear at the right of the window. As the image is subsequently copied onto the screen, each time a little more to the left, it will overlay the image that was copied in the previous iteration.

Eventually, assuming that the width of the image is less than the width of the clip rectangle, the entire image will be copied onto the window. On the next iteration it will be copied a little more to the left, and as the image has a 10% vertical buffer where the string is not rendered it is hoped, but not guaranteed, that the visible string will be overlaid; this is illustrated in Figure 10.4. This potential fault in the implementation will be addressed in an end of chapter exercise.

The class diagram given in Figure 10.1 was incomplete, as it did not include three further methods whose use may not be so obvious. The AWT Component class introduces the setForeground(), setBackground() and setFont() methods and, if any of these methods are called, then the *message* should be redisplayed with the new attribute. Consequently these methods must be overridden in the *DynaLabel* class, as follows.

```
0207    public void setForeground( Color newBackground) {
0208       super.setForeground( newBackground);
0209       this.setShowStyle( this.getShowStyle());
0210    } // End setForeground
0211
0212    public void setBackground( Color newBackground) {
0213       super.setBackground( newBackground);
0214       this.setShowStyle( this.getShowStyle());
0215    } // End setBackground
0216
0217    public void setFont( Font newFont) {
```

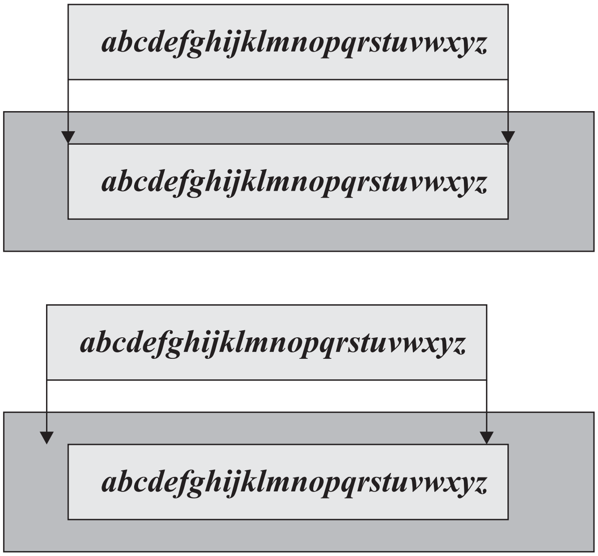

abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz

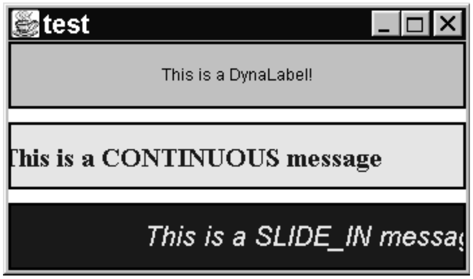
Figure 10.4 *DynaLabel showMessag()* horizontal overlaying.

```
      super.setFont( newFont);
      this.setShowStyle( this.getShowStyle());
   } // End setFont
```

Each of these methods commences by calling the inherited, **super**, version of the method that it is overriding and then calls *setShowStyle()*, passing the current *showStyle()* as an argument to cause the image to be reconstructed, as previously explained.

A partial demonstration of the *DynaLabel* is contained within the *DynaLabelDemo* class, which extends the JApplet class and whose *init()* action constructs three *DynaLabel* instances, one for each of the possible *showStyle*s, each with a different *message*, foreground and background colors and different *showSpeed*s, as follows.

```
    public void init() {
       this.setLayout( new GridLayout( 3, 1, 10, 10));

       DynaLabel demo1  = new DynaLabel();
       demo1.setBorder( new LineBorder( Color.black, 2));
       demo1.setShowStyle( DynaLabel.STATIONARY);

       DynaLabel demo2 = new DynaLabel(
                         "This is a CONTINUOUS message");
       demo2.setFont( new Font( "Serif", Font.BOLD, 12));
       demo2.setBorder( new LineBorder( Color.black, 2));
       demo2.setBackground( Color.yellow);
       demo2.setForeground( Color.blue);
       demo2.setShowStyle(  DynaLabel.CONTINUAL);

       DynaLabel demo3 = new DynaLabel(
                                 "This is a SLIDE_IN message");
```



This is a DynaLabel!

This is a CONTINUOUS message

This is a SLIDE_IN messa

Figure 10.5 *DynaLabelDemo* demonstration applet.

```
        demo3.setFont( new Font( "SansSerif", Font.ITALIC, 16));
        demo3.setBorder( new LineBorder( Color.black, 2));
        demo3.setBackground( Color.blue);
        demo3.setForeground( Color.yellow);
        demo3.setShowSpeed( 20);
        demo3.setShowStyle( DynaLabel.SLIDE_IN);

        this.add( demo1);
        this.add( demo2);
        this.add( demo3);
     } // End init.
```

An image of this applet shortly after it has been started is shown in Figure 10.5. The upper *DynaLabel*, *demo1*, is showing the default message with the default, *STATIONARY*, animation style using the default colors and font. The middle *DynaLabel*, *demo2*, is showing a *CONTINUAL* message using a serif font in blue on yellow at the standard speed. The lower *DynaLabel*, *demo3*, is showing a *SLIDE_IN* message using a sans serif font in yellow on blue at a slower speed, and so has not traversed as far as the middle label. All three instances have been awarded the same size by the GridLayout manager. This size will be the largest of the preferred sizes, probably obtained from the *demo3* instance.

This artifact gives some indication that the *DynaLabel* implementation seems to be working. But many of the methods of the class, for example *setShowSpeed()* or *setMessage()*, have not been demonstrated. The next part of the chapter will show how the *DynaLabel* class can be imported into a JavaBean tool, which will allow a more complete demonstration to be accomplished.

10.3 JavaBeans

The JavaBeans White Paper, details of which can be found in the Appendix, describes a JavaBean as:

> ...a reusable software component that can be manipulated visually in a builder tool.

This definition does not limit JavaBeans to visible user interface components, but in this section the *DynaLabel* class will be re-engineered so as to make it a more useful JavaBean. The definition of a bean can be divided into a number of major concerns as follows:

- ♦ *Introspection* – so that a builder tool can make inferences about a bean.
- ♦ *Customization* – so that the tool user can configure a bean while using the tool.

- *Event handling* – so that messages can be passed between beans.
- *Properties* – which allow customization to be performed.
- *Persistence* – so that the state of a bean can be saved from the tool and loaded into an artifact.

Some of these considerations have already been implicitly supplied in the artifacts that have already been developed. For example, the importance of event generation and listening has been emphasized since Chapter 1 and the customization of an instance of the JLabel class was discussed extensively in Chapter 3. The *DynaLabel* class already has many of the properties of a bean, and hence can be imported into a bean-aware tool and visually manipulated.

The introspection mechanism, at its simplest, works by analyzing the prototypes of a class's public methods and making inferences from them concerning the attributes which are supported. For example given following pairs of prototypes from the *DynaLabel* class.

```
public void setShowSpeed( int newSpeed)
public int  getShowSpeed()
public void setShowStyle( int newStyle)
public int  getShowStyle()
public void   setMessage( String newMessage)
public String getMessage()
```

A bean tool can infer that there are two **int**eger *properties* called *showSpeed* and *showStyle* and a String property called *message* associated with each *DynaLabel* instance. The tool can have no knowledge of the semantic meaning of these properties (that is what effect making changes to the values of these properties will have) nor of the range of valid values. However, it can offer a properties editor that allows the values to be shown and set, and which also allows the effect of changing the properties to be investigated.

> *The bean term property describes an attribute of a class whose existence and characteristics can be inferred by using the introspection mechanism; hence not all attributes will be properties.*

Before this can be demonstrated, using the Sun *BeanBox* tool which is distributed as a part of the Sun *Bean Development Kit* (BDK), the *DynaLabel* class needs to be packaged in a *Java Archive File* (`*.jar`) so that it can be imported into the BeanBox.

The first stage in preparing for the archiving of the *DynaLabel* class is to prepare a *manifest file* for the archive. A manifest file contains information that allows the environment that is using the archive to establish exactly what is contained within it. In particular, a bean-aware tool needs to know which classes in the archive are beans. The manifest file for the *DynaLabel* bean, called `DynaLabel.mf`, would be as follows.

```
Name: dynalabel/DynaLabel.class
Java-Bean: True
```

The first line starts with the jar keyword `Name:` and is followed by the full path to locate the class file containing the bean. As the *DynaLabel* class is contained within the *dynalabel* package, standard Java naming conventions produce the path `dynalabel/DynaLabel.class`, as shown. Notice that Unix-style slashes (`/`) must be used to describe the path even if the archive is being prepared in a Windows environment, which would normally use MS-DOS-style backslashes (`\`). The second and last line in the file starts with the jar keyword `Java-Bean:`, and the value `true` indicates that the class just named is a bean.

With the manifest file and the `DynaLabel.class` class file both in a subdirectory called `dynalabel` of the current working directory the jar archive file, to be called `DynaLabel.jar`, can be constructed and placed into the `dynalabel` directory using the jar tool distributed as part of the JDK. This is a command-line-driven tool and the command line to prepare the *DynaLabel* archive would be as follows.

```
>jar -cfm dynalabel\DynaLabel.jar dynalabel\DynaLabel.mf
dynalabel\DynaLabel.class
```

This command has been split over two lines on the page for typographic reasons only. When it is entered at the command prompt it should be typed as one continuous line. The flags given to the `jar` command are `c` for *c*reate a new archive, `f` to indicate that the first term of the command (`dynalabel\DynaLabel.jar`) is the location of the jar *f*ile, and `m`, which indicates that the first file in the list is a *m*anifest. The `jar` command can also be used to examine the contents of a jar file to confirm that the correct files have been archived into it. To examine the contents of the file just created, the command and its output would be as follows.

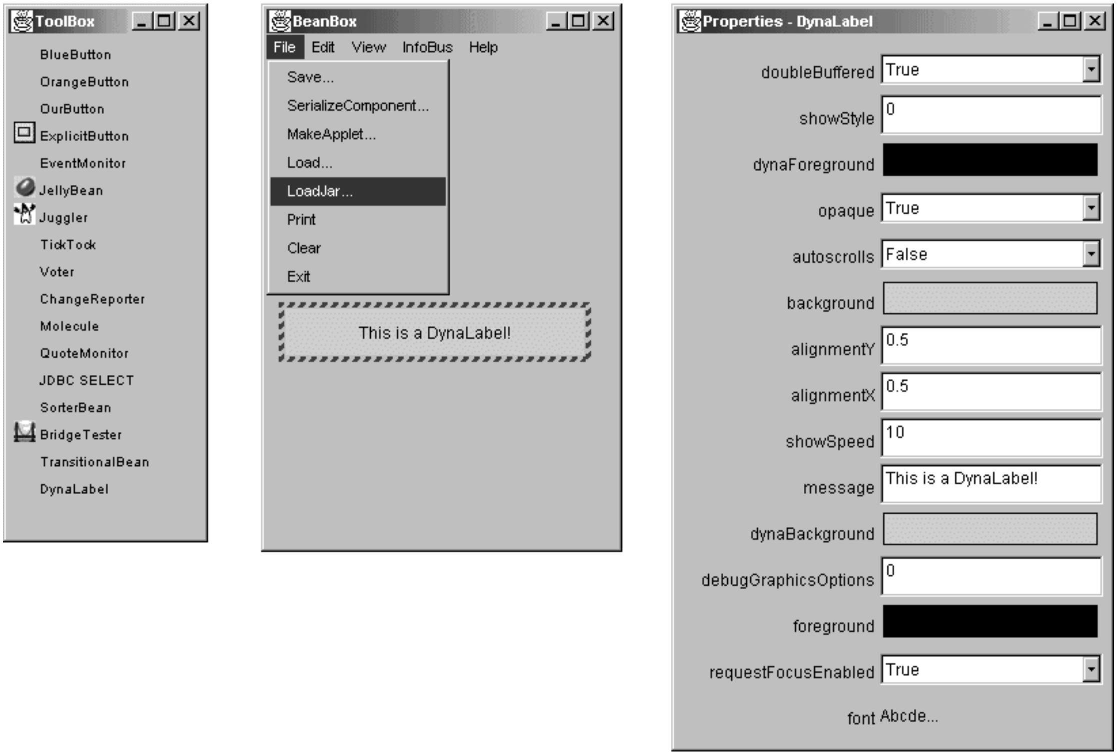
```
>jar -tf dynalabel\DynaLabel.

META-INF/MANIFEST.MF
dynalabel/DynaLabel.class
```

The flags to the `jar` command this time are `t` to *t*est the file and `f` to identify the java archive, as before. The first line of output is the manifest file, which has been renamed, added to and moved to a standard location and name so that the tool knows where to find it. The second line is the class file, which is the only other file contained within the archive. Jar files will be introduced in more detail in Chapter 12.

Having prepared the jar file it can be imported into the BDK BeanBox. Figure 10.6 illustrates the BeanBox when the archive file has been loaded and a *DynaLabel* instance has been placed on the work area.

The BeanBox has three windows visible. On the left is the *ToolBox* window, which contains a list of all the beans that are available for use; the *DynaLabel* bean is the last bean on the list. In the middle is the *BeanBox Work Area* with a *DynaLabel* instance placed upon it. On the right is the *Properties Sheet*, which is currently showing the properties which have been inferred about the component selected on the work area. It is showing the *DynaLabel* properties, as this is the currently selected bean as shown by the diagonal hashed box around it on the work area.

The BeanBox was configured into this state by first using the BeanBox *File* menu *LoadJar...* option, shown in the middle illustration in Figure 10.6. The `dynalabel\DynaLabel.jar` file was selected using the standard file selection dialog posted from the menu, and resulted in the *DynaLabel* bean being added to the end of the list in the toolbox. The *DynaLabel* bean was then selected in the toolbox by clicking upon it, causing the cursor to change to a cross-hair indicating that a bean was ready to be dropped on to



doubleBuffered True
showStyle 0
dynaForeground
opaque True
autoscrolls False
background
alignmentY 0.5
alignmentX 0.5
showSpeed 10
message This is a DynaLabel!
dynaBackground
debugGraphicsOptions 0
foreground
requestFocusEnabled True
font Abcde...

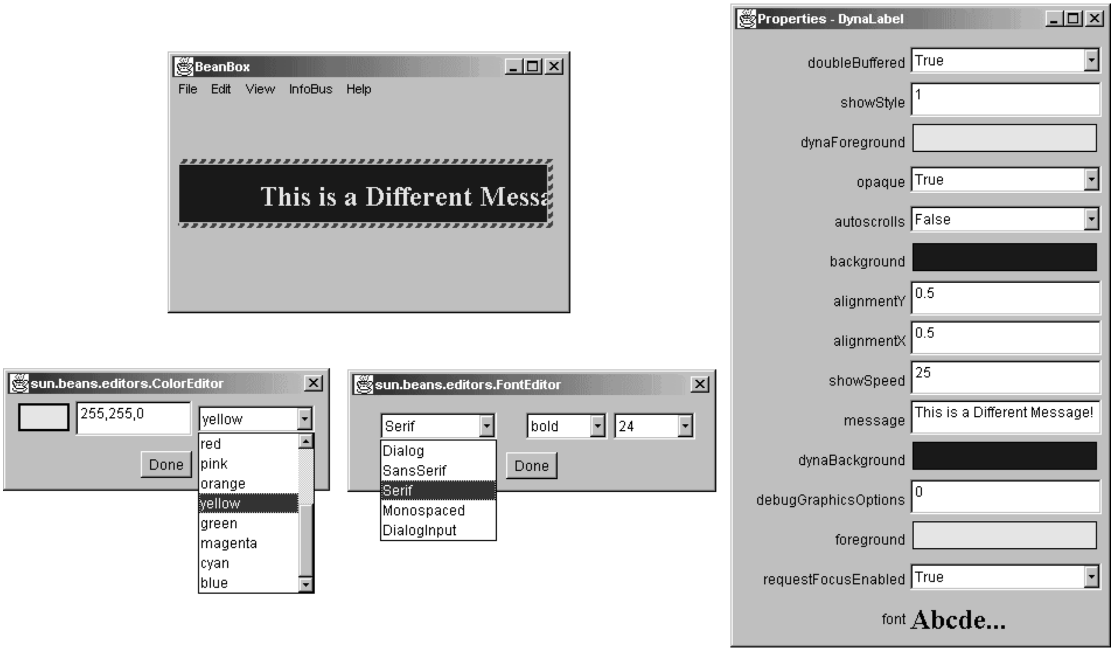
Figure 10.6 BeanBox with a *DynaLabel* instance selected.

the work area. Clicking on the work area caused the BeanBox to call the *DynaLabel* default constructor to prepare a *DynaLabel* instance, which is visible in the work area, and to reconfigure the properties sheet window so that it is connected to the instance.

The properties sheet does not list all the properties that the BeanBox was able to infer about the *DynaLabel* instance, only those that it has editors available for. For example, it was able to infer that the *DynaLabel* instance has border and toolTipText properties, but as it was unable to find editors for these properties it did not display them. It has, however, been able to provide editors for the *message*, *showSpeed* and *showStyle* properties, but has only been able to recognize *showStyle* as an **int**eger and not as a set of manifest values. It has also recognized a number of properties inherited from JComponent some of which, such as font or background, are immediately useful; others, such as alignmentY or doubleBuffered, are not particularly useful.

The editors on the properties sheet can be used to interactively set the values of the properties of the selected *DynaLabel* instance on the work area, with the consequences of these changes becoming immediately visible. Figure 10.7 shows the work area and the properties sheet after a number of changes have been made; it also shows the default color and font editors which are supplied by the BDK.

The *DynaLabel* instance has had its font changed using the font editor shown at the lower left. This dialog was posted by double-clicking upon the font editor feedback area of the properties sheet, shown as "*Abcde ...*". It supplies three pull-down menus allowing the three components of the font to be selected, with the results of each change becoming immediately visible in the work area. Pressing the "*Done*" button at the bottom of the



ColorEditor: 255,255,0 yellow

FontEditor: Serif bold 24

doubleBuffered	True
showStyle	1
dynaForeground	
opaque	True
autoscrolls	False
background	
alignmentY	0.5
alignmentX	0.5
showSpeed	25
message	This is a Different Message!
dynaBackground	
debugGraphicsOptions	0
foreground	
requestFocusEnabled	True
font	Abcde...

Figure 10.7 BeanBox with an edited *DynaLabel* instance.

dialog dismisses it from the desktop. Likewise, double-clicking on the foreground or background editor feedback area posts the color dialog, shown to the right of the font dialog. This dialog is not as sophisticated as the custom color editor developed in Chapter 3, allowing a color to be chosen by name from the pull-down menu at the right or by editing the *rgb* values in the text area. The effects of any changes to the attribute are immediately visible in the work area and are echoed as solid areas of color on the properties sheet and on the left of the editor dialog.

> *A dialog is a top-level window which is posted to inform the user of some occurrence, to ask the user a simple question or (as in this case) to supply the user with a specialized control. In UK English usage a useful distinction can be made between a dialog (a transient window posted from an artifact) and a dialogue (a sequence of interactions between the artifact and user).*

The editors provided for the *message*, *showSpeed* and *showStyle* properties are the text input areas alongside the name of the property. Typing a different *message* into the message editor area changes the message displayed in the *DynaLabel* instance when the <ENTER> key is pressed.

The *showSpeed* and *showStyle* text input areas require an integer value to be entered, but typing anything other than an integer is not reported as an error and has no effect upon the bean. This editor is more or less acceptable for the *showSpeed* property, but only the values 0, 1 and 2 are acceptable for the *showStyle* property. Any other integer value is not rejected by the bean, but causes it to have no animated behavior whatsoever, displaying the *DynaLabel* as a solid area of background color.

Figure 10.7, compared with Figure 10.6, shows that the message editor has been used to change the *message* in the bean and that its foreground and background colors have been changed, as has its font. The animation style has been set to 1 (*CONTINUAL*) and the speed slowed from 10 to 25. The instance has also been resized on the work area by dragging one of the corners while it had mouse focus. These changes allow a more convincing demonstration of the *DynaLabel* implementation, increasing the confidence that it is operating correctly.

This description of the use of the *DynaLabel* within the BeanBox indicates some of the potential power of the tool, but also exposes the limitations of importing a raw bean into the tool. Specifically, the properties sheet should suppress some of the properties which are currently showing and possibly reveal some which are currently not shown. It should also supply more intelligent and sophisticated property editors for some of its resources. The next section of this chapter will supply this capability.

10.4 BeanInfo classes

The list of properties that should be shown for a *DynaLabel* on the BeanBox property sheet is: *message*, background, foreground, font, *showSpeed* and *showStyle*; all of the other properties shown in Figure 10.6 should be suppressed. This can be accomplished by the provision of a *bean info class* which, using a strictly enforced naming convention, must be called *DynaLabelBeanInfo* and which extends the *SimpleBeanInfo* class. Overriding a method called getPropertyDescriptors(), which returns an array of PropertyDescriptor instances, will explicitly indicate which properties should be presented on the property sheet. The implementation of the *getPropertyDescriptors()* method in the *DynaLabelBeanInfo* class, is as follows.

```
public class DynaLabelBeanInfo extends SimpleBeanInfo {

private final static Class dynaLabelClass = DynaLabel.class;

   public PropertyDescriptor[] getPropertyDescriptors() {
      try {
          PropertyDescriptor backgroundDescriptor =
                new PropertyDescriptor("background",
                                       dynaLabelClass);
          PropertyDescriptor foregroundDescriptor =
                new PropertyDescriptor("foreground",
                                       dynaLabelClass);
          PropertyDescriptor fontDescriptor =
                new PropertyDescriptor("font",
                                       dynaLabelClass);
          PropertyDescriptor messageDescriptor =
                new PropertyDescriptor("message",
                                       dynaLabelClass);
          PropertyDescriptor showSpeedDescriptor =
                new PropertyDescriptor("showSpeed",
                                       dynaLabelClass);
          PropertyDescriptor showStyleDescriptor =
                new PropertyDescriptor("showStyle",
                                       dynaLabelClass);
      } catch (IntrospectionException exception) {
```

```
0035           throw new RuntimeException( exception.toString());
0036       } //End try/catch.
0037       PropertyDescriptor descriptors[] =
0038                 { backgroundDescriptor, foregroundDescriptor,
0039                   messageDescriptor, fontDescriptor,
0040                   showSpeedDescriptor, showStyleDescriptor};
0041       return descriptors;
0042    } // End getPropertyDescriptors.
```

The *DynaLabelBeanInfo* commences, on line 0012, by obtaining, from the Java run-time environment, an instance of the Class class which describes the *DynaLabel* class. Instances of the Class class are used by the Java run-time environment to obtain details about the attributes and other resources of a class, and will be used by this bean info class to obtain information about the properties and their methods. Within the *getPropertyDescriptors()* method a PropertyDescriptor instance is constructed for each of the properties which is to be made visible to the tool.

For example, lines 0016 to 0018 construct a PropertyDescriptor, called *background Descriptor*, which will contain the information needed for the background attribute to be presented as a bean property. The arguments to the constructor are the name of the property, as a String, and the Class instance from which it should infer the names of the read and write methods. The constructor will attempt to locate a pair of methods called getBackground() and setBackground() within the class's Class object and will throw an IntrospectionException if they cannot be found. Consequently, all six calls of the PropertyDescriptor constructor, one for each of the six properties which are to be exposed, are contained within a **try/catch** structure whose **catch** clause, on line 0035, propagates the *exception* as an instance of the RuntimeException class, which need not be noted in the *getPropertyDescriptors()* prototype.

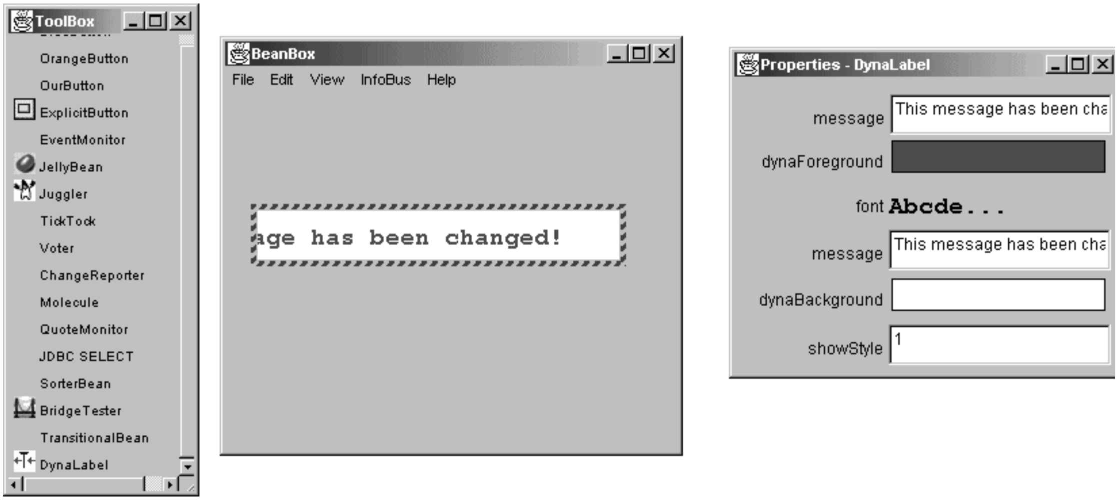
If an exception is not thrown than all six descriptors could be obtained and, on lines 0037 to 0040, an array of PropertyDescriptors, called *descriptors*, is constructed and, on line 0041, returned from the method. This class will have to be added to the `DynaLabel.jar` archive and will be used by the BeanBox, as the archive is loaded and the *DynaLabel* bean prepared, to indicate explicitly the list of properties that are to be shown in the properties sheet. Figure 10.8 shows the revised appearance of the properties sheet, and the rest of the BeanBox, with the bean info class available.

This version of the bean info class contains two further methods, the first of which informs the bean tool which of the properties is regarded as being the most fundamental and so should be placed at the top of the properties sheet. This method is called *getDefaultPropertyIndex()* and is implemented as follows.

```
0045      public int getDefaultPropertyIndex() {
0046          return 3;
0047      } // End getDefaultPropertyIndex
```

The method returns the location in the array returned from *getPropertyDescriptors()* of the property which should be positioned in the most prominent location. The value 3 returned from this method identifies the *message* property that is at the third location in the array. The other method supplied in this bean info class is called getIcon() and is used by the tool to obtain an icon for the bean in the Tool Box, as can be seen in Figure 10.8. The implementation of this method is as follows.

```
0049    public Image getIcon( int iconKind) {
```



message: This message has been cha
dynaForeground:
font: Abcde...
message: This message has been cha
dynaBackground:
showStyle: 1

Figure 10.8 BeanBox with *DynaLabeLBeanInfo* revealed properties.

```
0050        if ( iconKind == BeanInfo.ICON_MONO_16x16 ) {
0000           return loadImage("DynaLabelMonoIcon16.gif");
0051        } else if ( iconKind == BeanInfo.ICON_COLOR_16x16 ) {
0000           return loadImage("DynaLabelIcon16.gif");
0051        } else if ( iconKind == BeanInfo.ICON_MONO_32x32 ) {
0000           return loadImage("DynaLabelMonoIcon32.gif");
0051        } else if ( iconKind == BeanInfo.ICON_COLOR_32x32 ) {
0000           return loadImage("DynaLabelIcon32.gif");
0059        } else {
0060           return null;
0061        } // End if.
0063     } // End getIcon.
```

The argument to this method identifies one of the four possible icon types which bean tools may ask for (mono 16×16, color 16×16, mono 32×32, color 32×32). The manifest icon names indicate a two-color (monochrome) or full-color icon and the size, either 16 or 32 pixels square. The body of the method is implemented as an **if/else if...** structure containing a branch for each possible icon type. Within each branch the inherited SimpleBeanInfo loadImage() method is called. This will return either the identity of the icon constructed from the `gif` image file named in its argument or **null** if the icon could not be constructed. Line 0060 will return a **null** icon if the argument does not identify one of the valid manifest icon types. The `gif` icon files named in the method will have to be included in the *jar* archive and may be used by the bean tool to illustrate the bean in the toolbox.

10.5 Manifest property editor classes

The java.beans.Property editor **interface** defines a protocol by which a specialized property editor can be specified in a bean info class and used by a bean tool on the property sheet. The java.beans.PropertyEditorSupport supplies a simple implementation of the interface that can be easily extended to supply an editor for manifest values. In the first part of this section this process will be illustrated by the provision of an editor class, called

DynaLabelShowStyleEditor, which will allow the three possible styles of animation to be chosen from the BeanBox property editor.

In order to provide a property editor for a set of manifest values only three methods from PropertyEditorSupport need to be overridden. The first two of these are *getTags()* and *setAsText()*, implemented as follows.

```
0010  package dynalabel;
0011
0012  import java.beans.*;
0013
0014  public class DynaLabelShowStyleEditor
0015                  extends PropertyEditorSupport {
0016
0017  private final static String tagNames[] =
0018          {"Stationary", "Continual", "Slide In"};
0019
0020     public String[] getTags() {
0021        return tagNames;
0022     } // End getTags.
0023
0024
0025     public void setAsText( String tagName) {
0026        if ( tagName.equals( tagNames[ DynaLabel.STATIONARY])) {
0027           setValue( new Integer( DynaLabel.STATIONARY));
0028        } else if ( tagName.equals(
0029                              tagNames[ DynaLabel.CONTINUAL])) {
0030           setValue( new Integer( DynaLabel.CONTINUAL));
0031        } else if ( tagName.equals(
0032                                tagNames[DynaLabel.SLIDE_IN])) {
0033           setValue( new Integer( DynaLabel.SLIDE_IN));
0034        } // End if.
0035     } // End setAsText
```

The class's implementation commences on line 0017 and 0018 by declaring an array of Strings, called *tagNames*, which list the names of the three possible show styles, as will be presented to the user on the property editor. The bean tool can become aware of these names by calling the *getTags()* method, declared on lines 0020 to 0022.

The *setAsText()* method, declared on lines 0025 to 0035, will be called by the bean tool after the user has chosen one of the options offered, passing one of the Strings obtained by *getTags()* as an argument. The method is implemented as a three-way selection corresponding to the three possible String values. Each branch calls the inherited setValue() method, passing as an argument an Integer instance whose value corresponds to the appropriate manifest value from the *DynaLabel* class. It is the call of the setValue() method which sets the value of the associated resource in the instance being configured, as will be explained later.

The only remaining method that needs to be overridden in order to complete this editor is getJavaInitializationString(). This method should return a String that can be evaluated to produce the value to which the attribute of the instance being edited should be set. The implementation of this method is as follows.

```
0040     public String getJavaInitializationString() {
```

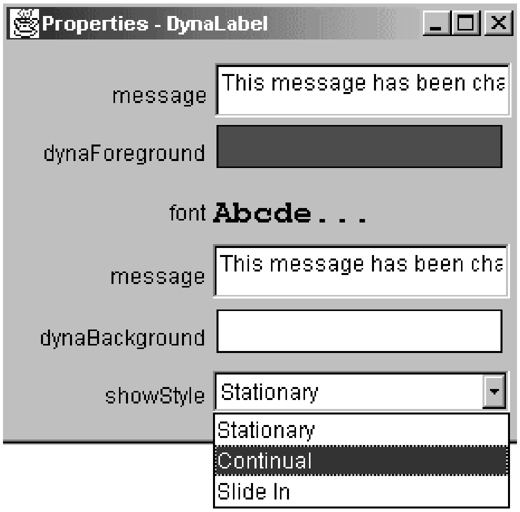
```
0041        switch( ((Number) this.getValue()).intValue()) {
0042
0043        default:
0044        case DynaLabel.STATIONARY :
0045                 return "dynalabel.DynaLabel.STATIONARY";
0046        case DynaLabel.CONTINUAL  :
0047                 return "dynalabel.DynaLabel.CONTINUAL";
0048        case DynaLabel.SLIDE_IN   :
0049                 return "dynalabel.DynaLabel.SLIDE_IN";
0050        } // End switch.
0051     } // End getJavaInitializationString.
```

The *getValue()* method, called on line 0041, returns the Integer object created in the *setAsText()* method and it is reconverted to an **int**. The value obtained is used as a selector expression of a **switch** structure, each **case** of which returns a String identifying one of the the **public** manifest value from the *DynaLabel* class.

With the *DynaLabelSetShowStyleEditor* class compiled into the *DynaLabel* package of classes the *DynaLabelBeanInfo* class has to be modified to inform the BeanBox that a specialized editor is available. The appropriate change is as follows.

```
0031           PropertyDescriptor showStyleDescriptor =
0032                 new PropertyDescriptor("showStyle",
0033                                         dynaLabelClass);
0034           showStyleDescriptor.setPropertyEditorClass(
0035                             DynaLabelShowStyleEditor.class);
```

The *showStyleDescriptror* PropertyDescriptor is constructed, as before, on lines 0031 to 0033 and has its setPropertyEditorClass() method called on lines 0034 to 0035. The argument to this call is the Class class of the editor to be used for this property. With this change made, the *DynaLabelBeanInfo* class recompiled and the archive remade to include it and the *DynaLabelSetShowStyleEditor* class, it will be used by the BeanBox, as shown in Figure 10.9.



Properties - DynaLabel

message: This message has been cha

dynaForeground

font: Abcde...

message: This message has been cha

dynaBackground

showStyle: Stationary

Stationary
Continual
Slide In

Figure 10.9 BeanBox with specialized *showStyle* editor.

The properties sheet has installed a pull-down menu with three choices on it, identifying the three possible animation styles. Selecting one of the styles from the menu will cause a message to be sent to the *DynaLabel* instance on the work area, calling its *setShowStyle()* method with the appropriate argument, and which will change the animation style of the instance. Figure 10.10 attempts to explain the processes involved in this scenario.

The properties sheet is informed by the *DynaLabelBeanInfo* class that there is a specialized editor available for the *showStyle* property and constructs an instance of it. It then calls its *getTags()* method and, as it does not return a **null** value, it knows that this is an editor for manifest values and constructs a pull-down menu containing the Strings in the array returned which it presents as the property editor user interface. When the user chooses one of the values the editor's *setAsText()* method is called with the String from the menu choice passed as an argument. The *setAsText()* method calls its own setValue() method, passing as an argument an Integer instance; it also dispatches a PropertyChangeEvent to its registered listener, which is the BeanBox.

The BeanBox knows from the receipt of the PropertyChangeEvent that a change of the showStyle is needed and calls the editor's *getJavaInitializationString()* to obtain an expression which it can evaluate to produce the **int** value which it passes as an argument to the *DynaBean* instance's *setShowStyle()* method. This is a general purpose technique which relies upon information being passed between the various components as Strings, the reasons for which will become more apparent when the *CustomColorPanel* from Chapter 3 is re-engineered for use within a properties sheet.

> *This class can be used as a template for producing a manifest value editor for any bean. The appropriate changes would have to be made to all three methods, but the changes are straightforward and should not be troublesome.*

10.6 Custom property editor classes

This section will introduce the techniques by which a custom property editor can be implemented. It will do this be re-engineering the *CustomColorPanel* from Chapter 3 and install it as the editor for the *DynaPanel* foreground and background properties. The appearance of the editor, an instance of the *DynaLabelColorEditor* class, is shown in Figure 10.11. There is one major difference in the appearance of this editor from that in Chapter 9: the "*OK*" button at the bottom of the editor is not required in this implementation and has been omitted.

The editor was posted onto the desktop by double-clicking upon the feedback area to the right-hand side of the *foreground* and *background* labels in the properties sheet. It operates in an identical manner to the *CustomColorPanel*: moving one of the sliders will change the color represented, and this change is immediately echoed in the feedback area of the editor panel, the feedback area on the properties sheet and the *DynaLabel* instance on the work area.

As instances of the *DynaLabelColorEditor* class do not have an "*OK*" button the class need not implement the ActionListener interface in order to listen to it. However, it still has to implement the ChangeListener interface in order to listen to the events generated by the sliders. It also has to implement the PropertyEditor interface in order to be used by

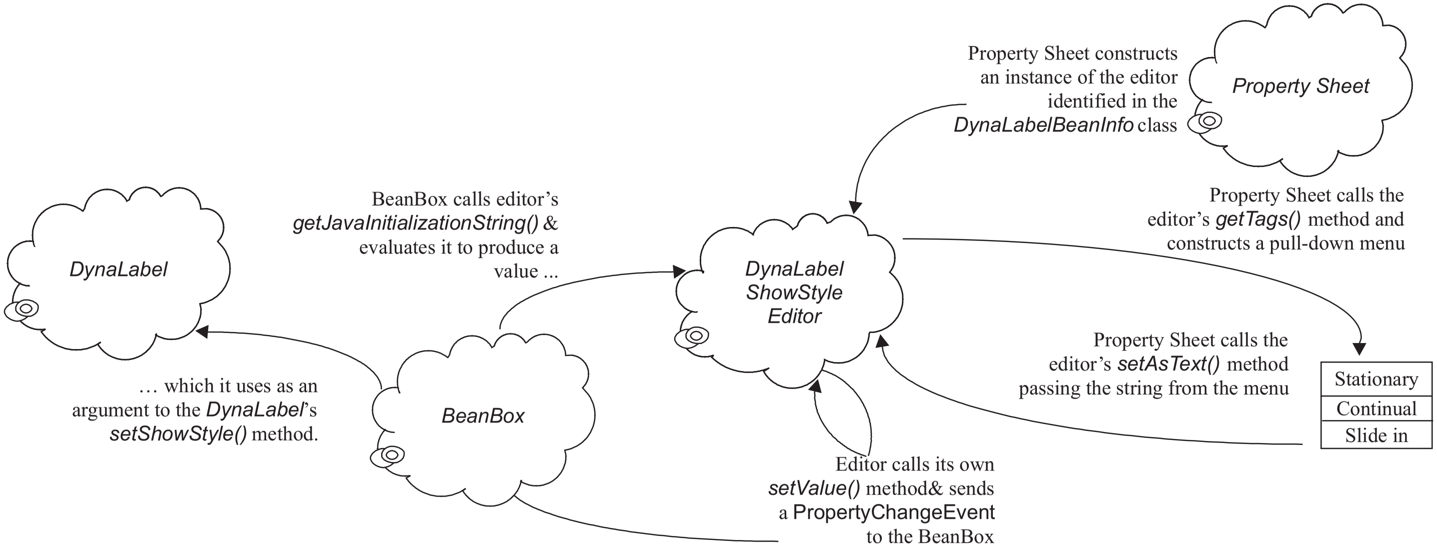
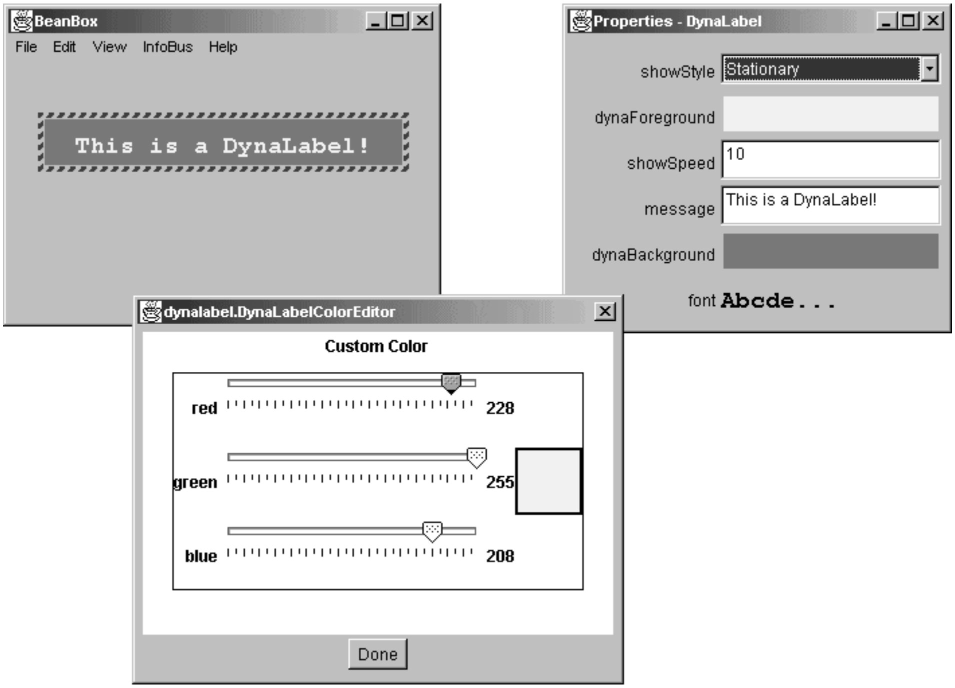

Property Sheet
Property Sheet constructs an instance of the editor identified in the *DynaLabelBeanInfo* class
Property Sheet calls the editor's *getTags()* method and constructs a pull-down menu
Stationary
Continual
Slide in
Property Sheet calls the editor's *setAsText()* method passing the string from the menu
DynaLabel *ShowStyle* *Editor*
Editor calls its own *setValue()* method& sends a PropertyChangeEvent to the BeanBox
BeanBox calls editor's *getJavaInitializationString()* & evaluates it to produce a value ...
BeanBox
… which it uses as an argument to the *DynaLabel*'s *setShowStyle()* method.
DynaLabel

Figure 10.10 Operation of specialized *showStyle* editor.



showStyle Stationary
dynaForeground
showSpeed 10
message This is a DynaLabel!
dynaBackground
font Abcde . . .

red 228
green 255
blue 208

Figure 10.11 Operation of specialized *customColor* editor.

the properties sheet as an editor. This is shown at the top of its class diagram, given in Figure 10.12.

The *stateChanged()* method must be supplied in order to implement the ChangeListener interface, and all other methods, apart from *getPreferredSize()*, must be supplied in order to implement the PropertyEditor interface. The first set of methods is concerned with informing the properties sheet that this editor is a custom editor and not a manifest editor. Hence *getTags()* will return **null**, *supportsCustomEditor()* will return **true** and *getCustomEditor()* will return its own identity (**this**). The properties sheet will call the *getCustomEditor()* method to obtain the editor and mount it within a dialog panel with a "*Done*" button at the bottom when the user requests it. The *getPreferredSize()* method is supplied so that the editor can ask for a suitable size as it is laid out on the dialog panel.

The *getValue(), setValue(), getAsText()* and *setAsText()* methods are supplied to support the *beanColor* attribute which will always indicate the current value of the Color being edited. The *getJavaInitializationString()* serves the same purpose as in the previous editor, providing an *initializationString* which can be evaluated to produce an instance of the Color class whose value is the same as the *beanColor* attribute and which can be passed to the *DynaLabel* instance's *setForeground()* or *setBackground()* methods. The *isPaintable()* method will return **true**, indicating that this editor will take responsibility for the feedback area alongside the label on the properties sheet and the *paintValue()* method will be called, when required, to allow it to provide the feedback.

All PropertyEditors are sources of PropertyChangeEvents, and the last two methods, *addPropertyChangeListener()* and *removePropertyChangeListener()*, are supplied to register and de-register listeners. As the *addPropertyChangeListener()* does not throw an

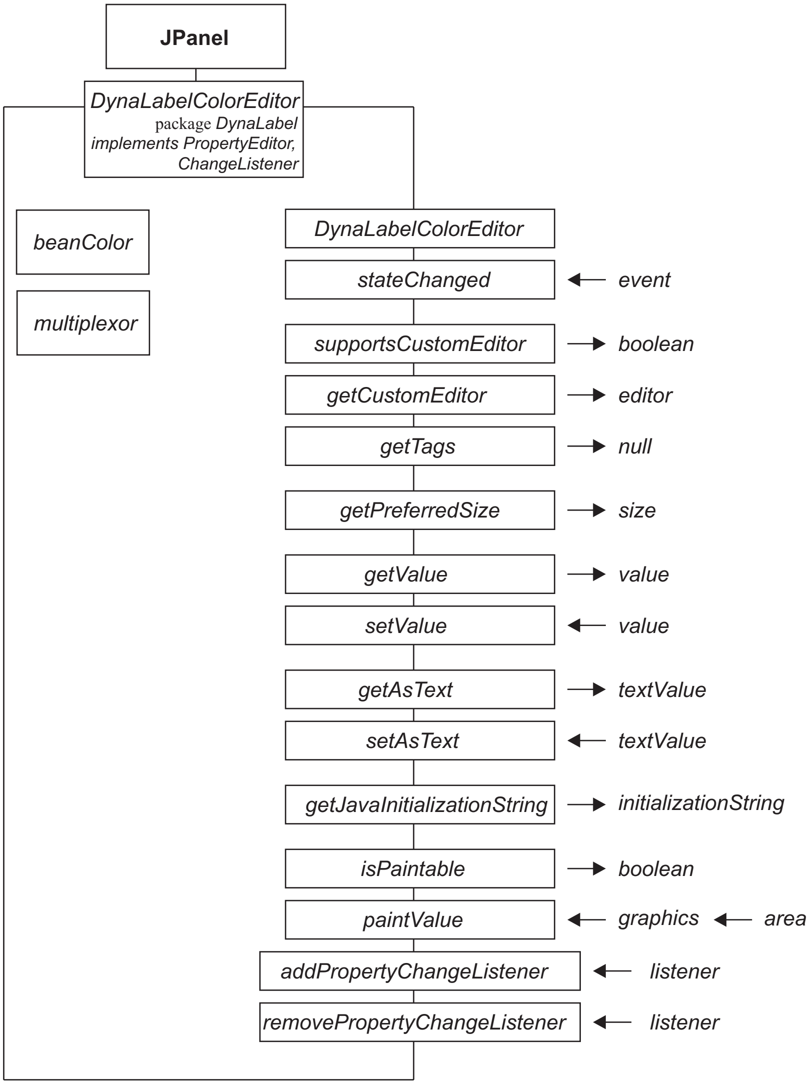


Figure 10.12 *DynaLabelColorEditor* class diagram.

exception it can be inferred (by the developer) that this is a multiplexing event source, and the *multiplexor* attribute, an instance of the PropertyChangeSupport class, facilitates maintaining a list of PropertyChangeEventListeners.

The implementation of this design commences, as follows, with the class declaration and the declaration of its attributes.

```
public class DynaLabelColorEditor
                       extends    JPanel
                       implements PropertyEditor,
                                  ChangeListener {

```

```
0024
0025  private Color beanColor = null;
0026  private PropertyChangeSupport multiplexor =
0027                        new PropertyChangeSupport( this);
```

The *beanColor* attribute, declared on line 0025, is used to record the value of the color chosen by the user. It need not be given a value upon declaration, as the properties sheet will supply it with a value from the *DynaLabel* instance whose resources are being edited as the dialog is placed on the desktop. The *multiplexor* is declared, on lines 0026 and 0027, as an instance of the PropertyChangeSupport class and its constructor identifies the editor (**this**) which it is to support. Using an instance of the PropertyChangeSupport class removes the need for this class to explicitly maintain a list of listeners and to inform each one individually whenever a PropertyChange event needs to be generated, as will be explained in detail later.

The constructor does not differ greatly from the *CustomColorPanel* constructor, given in Chapter 3 and whose layout design is given in Figure 3.21, apart from the omission of the "*OK*" button and the JPanel which it was mounted upon. The *stateChanged()* method also does not differ significantly; its implementation within the *DynaLabelColorEditor* class is as follows.

```
0137     public void stateChanged( ChangeEvent event){
0138
0139     int red   = redSlider.getValue();
0140     int green = greenSlider.getValue();
0141     int blue  = blueSlider.getValue();
0142
0143        beanColor = new Color( red, green, blue);
0144        feedbackArea.setBackground( beanColor);
0145
0146        redValue.setText(   (new Integer( red)).toString());
0147        greenValue.setText( (new Integer( green)).toString());
0148        blueValue.setText(  (new Integer( blue)).toString());
0149        this.setAsText( red + "," + green + "," + blue);
0150     } // End StateChanged.
```

This implementation differs on line 0143, where the Color value constructed from the values of the three sliders is used to change the color recorded in the *beanColor* attribute. The only other change is on line 0149, where the *setAsText()* method is called with an argument constructed from the three slider values; the significance of this step will be explained below. The implementation of the *DynaLabelColorEditor* class continues as follows.

```
0153      public boolean supportsCustomEditor() {
0154         return true;
0155      } // End supportsCustomEditor.
0156
0157      public Component getCustomEditor() {
0158         return this;
0159      } // End getCustomEditor.
0160
0161      public String[] getTags() {
0162         return null;
0163      } // End getTags.
0164
0165      public Dimension getPreferredSize() {
0166         return new Dimension(350,250);
0167      } // End getPreferredSize
```

The first of these methods, *supportsCustomEditor()*, returns **true** to indicate that this class will supply a custom editor, and the identity of the editor can obtained by calling *getCustomEditor()*. The properties sheet will call this latter method and place the Component returned within a dialog panel with a "*Done*" button below it, configured to dismiss the panel when it is pressed. The *getPreferredSize()* method is supplied so that the editor can indicate how large it would like to be. The remaining method in this fragment, *getTags()*, has to be supplied to satisfy the PropertyEditor interface and is implemented as a method which returns **null**. The next two methods are implemented as follows.

```
0170      public void setValue( Object newValue) {
0171
0172      Color newColor = (Color) newValue;
0173         redSlider.setValue(   newColor.getRed());
0174         greenSlider.setValue( newColor.getGreen());
0175         blueSlider.setValue(  newColor.getBlue());
0176         this.stateChanged( new ChangeEvent( this));
0177      } // End setValue.
0178
0179      public Object getValue() {
0180         return beanColor;
0181      } // End getValue
```

These two methods set and retrieve the value of the Color that is being edited by this class, stored in *beanColor*. These methods are mandated by the PropertyEditor class and so may be used to set and get an arbitrary value. Consequently they are implemented as methods requiring, or returning, an Object instance and must employ casting to change the instance to Color.

The *setValue()* method will be called by the properties sheet as the editor is posted onto the desktop, passing as an argument the current foreground or background color of the *DynaLabel* instance which is being edited. By the time the user sees the editor it should be showing, on the sliders and on the feedback area, the current color which is being edited. This method accomplishes this, on line 0172, by casting the Object argument *newValue* into a *Color* instance called *newColor*. Having done this (lines 0173 to 0175) set the value of the three sliders on the editor's interface by calling their setValue() method, passing the appropriate component value from the *newColor* instance. To ensure that all other aspects of the editor instance are in a consistent state, line 0176 sends a ChangeEvent to the *stateChanged()* method. The argument to the ChangeEvent constructor is the identity, **this**, of the source of the event.

The *stateChanged()* method, as already described in detail, will update the values on the numeric labels and the color displayed on the editor's feedback area. It will also call the editor's *setColorAsText*() method, which will be described shortly, to complete the change of value. The effect of this method is that when the user posts the editor from the properties sheet the interface, and all attributes which support the interface, will be set so as to reflect the value of the attribute in the bean that it is editing.

The next two methods, *setAsText()* and *getAsText()* provide an interface to setting, or retrieving, the Color value being edited, in *beanColor*, as a String. The String is expected to be in the format "*rrr,ggg,bbb*"; that is, red, green and blue integer values in the range 0 to 255 separated by commas. The implementation of these two methods is as follows.

```
0180      public void setAsText( String colorString)
0181                            throws IllegalArgumentException {
0182
0183      int comma1  = colorString.indexOf(',');
0184      int commma2 = colorString.indexOf(',', comma1 +1);
0185
0186         if ( comma1 < 0 || commma2 < 0) {
0187             throw new IllegalArgumentException( colorString);
0188         } // End if.
0189         try {
0190             int red   = Integer.parseInt(
0191                             colorString.substring( 0, comma1));
0192             int green = Integer.parseInt(
0193                             colorString.substring(
0194                                            comma1+1, commma2));
0195             int blue  = Integer.parseInt(
0196                             colorString.substring( commma2+1));
0197             beanColor= new Color( red, green ,blue);
0198             multiplexor.firePropertyChange("", null, null);
0199         } catch ( NumberFormatException exception) {
0200             throw new IllegalArgumentException( colorString);
0201         } // End try/catch.
0202      } // End setAsText
0203
0204      public String getAsText() {
0205         return  beanColor.getRed()   + "," +
0206                 beanColor.getGreen() + "," +
0207                 beanColor.getBlue();
0208      } // End getAsText.
```

The *setAsText()* method commences by using the String indexOf () method to determine the location within the string of the two commas. If there are no commas in *colorString* then the value of *comma1* will be set to a negative value by line 0183; and likewise *comma2* by line 0184, if there are fewer than two commas. Should either of these two variables have a negative value an IllegalArgumentException with the *colorString* as its reason is thrown on line 0187.

If no exception is thrown then execution of the method continues with the **try** block starting on line 0189. The first part of this block, on lines 190 to 196, attempts to extract the **int**eger values from the *colorString*, using the String substring() method and the locations of the commas as previously determined. If the substrings cannot be converted to **int** values a NumberFormatException will be thrown and propagated from the *setAsText()* method as an IllegalArgumentException, on line 0200. Hence if line 0197 is reached the three integer values have been successfully extracted and they are used to construct a new Color instance which is assigned to *beanColor*.

This is the technique which will be used to change the Color being used for the editor, should it ever need to happen, by the BeanBox, so its listeners should be notified of the change. Notification is accomplished, on line 0198, by calling the *multiplexor*'s firePropertyChange() method. As all that needs to be done, in this circumstance, is to send a PropertyChange event, no informative arguments need to be supplied to the method.

Upon receipt of the PropertyChange event the BeanBox, which is listening to this editor, will call this editor's getJavaInitializationString() method to obtain the changed value in a format which it can make use of, just as described for the *showStyle* editor in Figure 10.11. This notification step of the *setAsText()* method is the reason why the *stateChanged()* called this method in response to the user moving one of the sliders.

The *getAsText()* method is considerably simpler and returns a String indicating the current value of *beanColor* in the appropriate format. This method is called as part of the getJavaInitializationString() method whose implementation is as follows.

```
0210      public String getJavaInitializationString() {
0211         return "new java.awt.Color(" + this.getAsText() + ")";
0212      } // End getJavaInitializationString.
```

The String returned from this method, when evaluated, will result in a call of the java.awt.Color() constructor with the arguments "*rrr,ggg,bbb*", as provided by *getAsText()*. The result of the evaluation will be a Color instance representing the same color as *beanColor*, and the new instance will be passed to the *DynaLabel* on the work area as an argument to its *setForeground()* or *setBackground()* method. The effect of this will be that the color indicated by the user moving the slider will be installed as a property of the *DynaLabel*. The action should also result in the area alongside the label in the properties sheet changing color, and this is accomplished by the next two methods.

```
0215      public boolean isPaintable() {
0216         return true;
0217      } // End isPaintable.
0218
0219      public void paintValue( Graphics context, Rectangle area) {
0220          context.setColor( beanColor);
0221          context.fillRect( area.x,     area.y,
0222                            area.width, area.height);
0223      }  // End paintValue.
```

The *isPaintable()* method indicates to the properties sheet whether the editor is able to take responsibility for the feedback area alongside the label, the value **true** returned indicating that this editor is willing to do so. Painting of the area is accomplished with a call of *paintValue()* (allowing the editor to *paint* the edited *value* onto the properties sheet) the arguments to the method being a Graphics *context* which can be used and the extent of the *area* available. All that this editor needs to do is to fill the whole extent of the *area* with the current *beanColor*, and this is accomplished, on lines 0220 to 0222, using techniques which were explained in the description of the *DynaLabel* earlier in this chapter. The equivalent action in a customized font editor would be responsible for rendering the "*Abcde...*" string in the chosen font, as can be seen in Figure 10.8.

The last two methods, *addPropertyChangeListener()* and *removePropertyChangeListener()*, support the list held by the *multiplexor* and are implemented as follows.

```
0226      public void addPropertyChangeListener(
0227                                PropertyChangeListener listener) {
0228         multiplexor.addPropertyChangeListener( listener);
```

```
    } // End addPropertyChangeListener.

    public void removePropertyChangeListener(
                            PropertyChangeListener listener) {
       multiplexor.removePropertyChangeListener( listener);
    } // End removePropertyChangeListener.
```

The *multiplexor* is an instance of the PropertyChangeSupport class and, as its name suggests, it takes care of maintaining a list of listeners with its addPropertyChangeListener() and removePropertyChangeListener() methods. It will also send a PropertyChange event to each listener when its firePropertyChange() method is called, as it was in the *setTextValue()* method above. The details of how the PropertyChangeSupport class accomplishes these actions need not be discussed; all that needs to be known is how to use it, as illustrated in this class.

Having constructed and compiled the *DynaLabelColorEditor* class the *foregroundDescriptor* and *backgroundDescriptor* instances in the *DynaLabelBeanInfo* class will have to have their setPropertyEditorClass() methods called to inform them of the identity of the class (*dynalabel.DynaLabelColorEditor*.class) of the specialized editor. The jar archive will then have to be remade to contain the amended *DynaLabelBeanInfo* class and the *DynaLabelColorEditor* class and re-imported into the BeanBox. Double-clicking upon the foreground or background feedback areas of the properties sheet should then post an instance of the specialized color editor, within a dialog panel, as shown in Figure 10.11.

The operation of the *DynaLabelColorEditor* is illustrated in Figure 10.13. As with other object interaction diagrams, time advances from the top of the screen, and this shows how the user obtains the three confirmations that interaction with the sliders is having an effect upon the *DynaLabel* instance in the BeanBox work area.

Unlike the adaptation of the manifest editor example for use with other beans, or with other properties, it not as straightforward to adapt this example for reuse. Some parts of it, for example those concerned with the multiplexor, *can be reused, but the majority of it will have to be hand-crafted.*

10.7 Serializing a bean

One of the defining characteristics of a bean, listed in Section 10.3, was *persistence*, described as the ability of a bean to be saved from a bean-aware tool, such as BeanBox, and then loaded into an artifact in the same state as when it was saved. The same techniques can be used to move a bean from one machine to another via the Internet. This capability relies upon a Java facility known as *object serialization*. In this part of the chapter the *DynaLabel* class will be revisited so as to make it serializable, and then a demonstration applet, which restores an instance of a serialized bean, will be presented.

A serialized bean contains sufficient information to allow an object to be reconstructed in the same state as when a bean instance was serialized and written to the stream. To do this it is necessary for the values of all significant attributes of the instance to be written to the stream and for a method to be provided which restores the instance into a viable state after reading it from the stream.

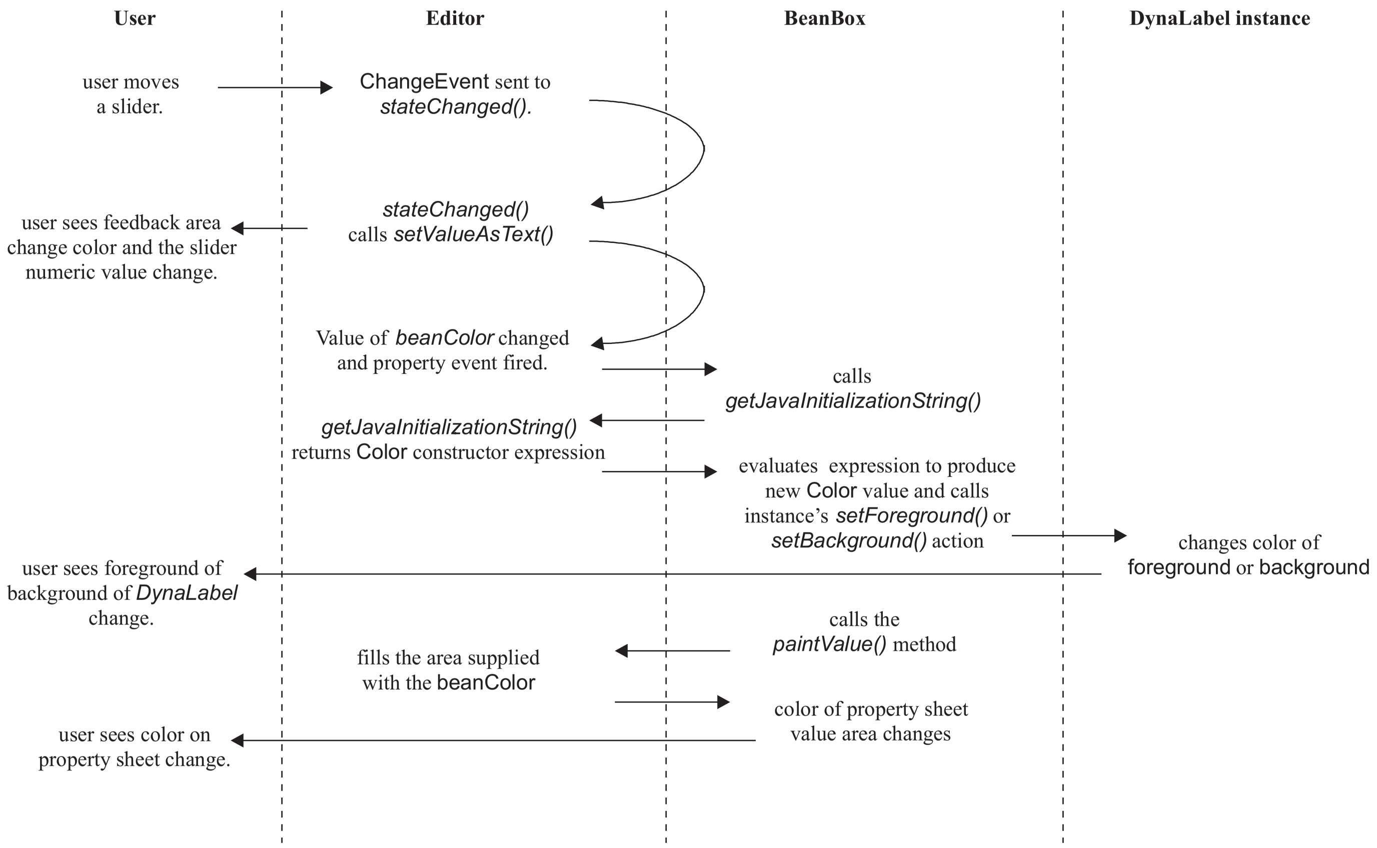


Figure 10.13 Operation of custom color editor.

The first change required to the *DynaLabel* implementation is to indicate that it implements the *Serializable* interface. This interface is contained in the java.io package of classes, so this package should be added to the importations at the start of the file.

```
import java.io.*;

public class DynaLabel extends    JComponent
                       implements Runnable,
                                  Serializable  {
```

The *Serializable* interface, contained within the java.io package, mandates no methods which have to be provided by a class implementing the interface, but merely serves as an indicator to Java that instances of this class are capable of being serialized. The next change that is required is to decide which of the attributes of the class need to be stored in the serialized file. Not all attributes should be saved, only those which are essential to restore the state of the instance; and not all attributes are capable of being serialized, as only attributes of classes which are themselves serializable can be stored.

java.io.Serializable interface

This interface mandates no methods but serves as a marker to indicate that the class has been constructed so as to be serializable. Any attributes of the class which should not be stored in the serialized stream should be declared with the modifier **transient**. Any attributes which are of a class which is not itself serializable will cause an exception to be thrown when the operation is attempted.

In this example the *imageToShow* attribute, of the Image class, cannot be stored as the Image class does not implement the *Serializable* interface. However, this does not prevent *DynaLabel* instances from being serialized, as the *imageToShow* will be reconstructed from the *message* attribute in the *prepareImage()* method upon deserialization. Any attributes of the class which should not be stored when the instance is serialized should be declared with the modifier **transient**. The revised declaration of the *DynaLabel* attributes is as follows.

```
private transient Thread     itsThread;
private           String     message = null;
private transient Image      imageToShow = null;

private transient Dimension  optimalSize = null;

private           int        showSpeed = 10;
private           int        showStyle = STATIONARY;
private transient int        thePhase  = START_PHASE;
private transient boolean    clearWindow = true;
```

The *imageToShow* attribute is declared transient for reasons which have already been explained. The Thread attribute cannot be serialized, as the Thread class does not implement the *Serializable* interface, because a reference to a Thread instance is a reference to an internal object of the Java run-time environment and this will be meaningless when the instance is deserialized. The *optimalSize* Dimension instance is marked transient as the size stored in *optimalSize* is related to the font which is currently being used. If the instance is deserialized in a different environment the characteristics of the font being used may be different, so the size recorded would be invalid. The other two transient attributes, *thePhase* and *clearWindow*, will have their values established when the *imageToShow* attribute is recreated in the *prepareImage()* method. The class-wide (**static**) attributes do not need to be declared as **transient** as, being attributes of the class and not of the instance, they will not be stored as part of the state of the object.

Only a minimal set of attributes should be serialized, as this will reduce the size of the serialized stream, which will reduce the size of the serialized file and also reduce the time taken for the serialized instance to be transmitted across the Internet.

In addition to non-transient attributes introduced by the *DynaLabel* class, all inherited non-transient attributes will also be stored when the instance is serialized. This is possible as the AWT Component class implements the Serializable interface, so all classes extended from it are also Serializable. However, explicitly marking an extended component as being Serializable is a useful documentation aid, indicating that the designer of the class has considered which added attributes should be declared **transient**.

These changes are all that is required for *DynaLabel* instances to be capable of being serialized. The BeanBox work area *File* menu, as shown in Figure 10.6, contains an option labeled *SerializeComponent...*. Activating this option leads to a standard file save dialog which allows the host machine's file store to be navigated and a file name for the serialized file to be selected or entered. By convention it is expected that a serialized object will be stored in a file which has a `.ser` extension. For the purposes of this example it will be assumed that a *DynaLabel* instance is stored in the *dynalabel* package directory, in a file called `demoser.ser`.

Although serialization of an object can be accomplished automatically, without any changes required to the *DynaLabel* class, this is not the case for the deserializing an instance. In order to deserialize a *DynaLabel* instance a method called *readObject()* will have to be supplied, implemented as follows.

```
0200     private void readObject( ObjectInputStream inStream)
0201                  throws IOException, ClassNotFoundException {
0202        inStream.defaultReadObject();
0203        this.getPreferredSize();
0204        itsThread = new Thread( this);
0205        itsThread.start();
0206     } // End readObject.
```

Curiously this method has to be declared as **private** even though it will be called from outside the class. The reasons for this are related to the intimate nature of deserialization and are a rare exception to the Java visibility rules. The argument to the method is an instance of the ObjectInputStream class, contained within the java.io package of classes, and which provides a method called defaultReadObject(), which is called as the first step of the implementation of the method on line 0202. Calling this method will effect the deserialization of the non-transient attributes of the *DynaLabel* instance. These were written to the stream by the BeanBox with a call of the corresponding ObjectOutputStream defaultWriteObject() method.

Object → OutputStream → Java.io.ObjectOutputStream and
Object → InputStream → java.io.ObjectInputStream

```
(ObjectOutputStream) public void writeObject(Object toWrite)
```

```
                                           throws IOException
(ObjectOutputStream) public void defaultWriteObject(Object toWrite)
                                           throws IOException
(ObjectInputStream) public Object readObject()throws
          OptionaldataException, ClassNotFoundException, IOException
(ObjectInputStream) public Object defaultReadObject()throws
             ClassNotFoundException, NotActiveException, IOException
```

Objects are serialized with a call of writeObject() and deserialized with a call of readObject(). If these methods are overridden in a Serialized class then the defaultWriteObject() and defaultWriteObject() methods should be used.

The remainder of the implementation of the method is concerned with restoring the *DynaLabel* instance into a viable state. This involves ensuring that the non-transient attributes are reestablished with suitable values. The first step is to call the *getPreferredSize()* method to restore the value of the *optimalSize()* Dimension attribute. (This method returns an instance of the Dimension class, but Java allows this value simply to be disposed of without any explicit action required by the developer.) The remaining two lines of the method recreate the Thread and call its start() method, this will result in the calling of the *DynaLabel*'s *run()* method which will, in time, call the *showMessage()* method. At the start of this method it will be noticed that the *imageToShow* attribute has a **null** value and this will cause the *prepareImage()* method to be called, which will recreate the *imageToShow* Image instance and also initialize the value of *thePhase* and *clearWindow*.

The effect of calling this method is to restore a *DynaLabel* instance so that its attributes have a set of values that are equivalent to the values of the instance which were originally serialized. It is possible that this attempt will throw exceptions of various classes, as noted in the prototype of the method, and these will be propagated from the method.

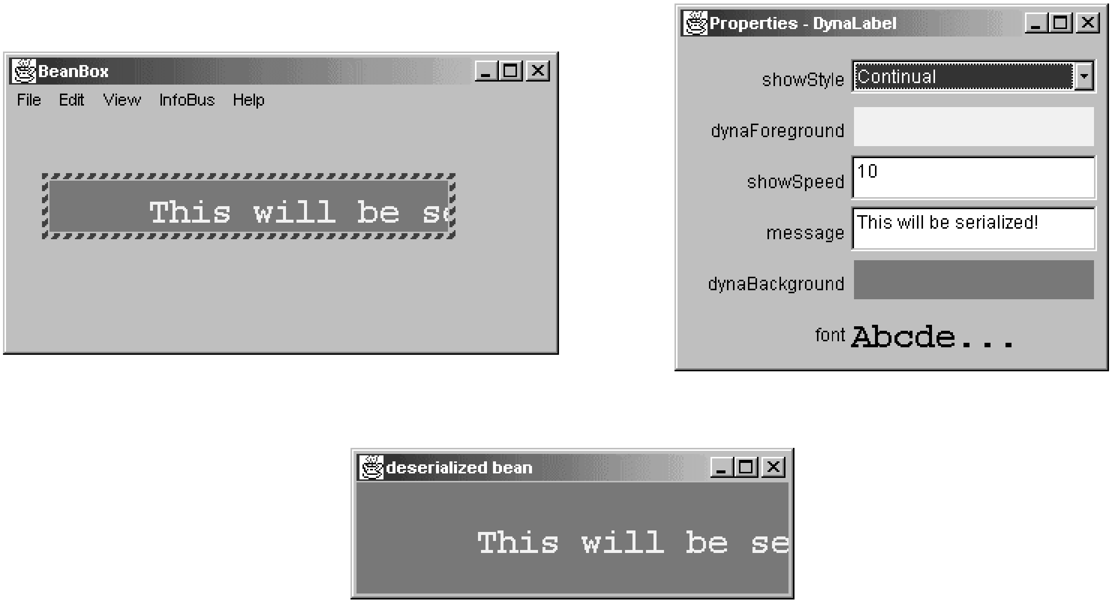
To demonstrate that this method will effectively deserialize a *DynaLabel* instance, an applet, called *DynaLabelDeserialDemo*, whose *init()* method is implemented as follows, can be used.

```
package dynalabel;

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

public class DynaLabelDeserialDemo extends JApplet {


    public void init() {

    ObjectInputStream inStream     = null;
    DynaLabel         deSerialized = null;

       try {
          inStream = new ObjectInputStream(
                         new FileInputStream(
                                "dynalabel/demoser.ser"));

0029             deSerialized = (DynaLabel) inStream.readObject();
0030
0031             inStream.close();
0032          } catch ( Exception exception) {
0033             throw new RuntimeException( exception.toString());
0034          } // end try/catch.
0035
0036          this.getContentPane().add( deSerialized);
0037       } // End init.
```

The *init()* action declares two local variables: an instance of the ObjectInputStream class, called *inStream*, to deserialize the *DynaLabel* instance *deSerialized* from. The first step of the *init()* method is to create the *inStream* instance, passing as an argument to the ObjectInputStream constructor an anonymous instance of FileInputStream which is connected to the file `demoser.ser` in the *dynalabel* directory.

If no exception is thrown, the next step, on line 0029, is to call the *inStream* readObject() method. This method will return an instance of the Object class, which must be cast to *DynaLabel*. Internally this will cause the Java run-time system to call the *DynaLabel readObject()* method to initialize the *DynaLabel* object, as previously described. This part of *init()* concludes by calling the *inStream* close() action to close the stream and hence the file.

If no exceptions are thrown then *deSerialized* will have been initialized to a state equivalent to that when the DynaLabel instance was serialized from the BeanBox. The *main()* method of the *DynaLabelDeserialDemo* is as expected. Figure 10.14 shows the BeanBox with a *DynaLabel* instance about to be serialized and the *DynaLabelDeserialDemo* artifact showing the *deSerialized* instance.



showStyle Continual

showSpeed 10

message This will be serialized!

font Abcde...

Figure 10.14 Serializing and deserializing.

10.8 Bound and constrained properties

Of the five defining characteristics of a bean listed in Section 10.3, *introspection*, *customization*, *event handling*, *properties* and *persistence*, all, apart from event handling, have been illustrated by the *DynaLabel* class. To describe event handling in the context of a bean, two significant aspects of bean properties that have not been considered so far, bound and constrained properties, will have to be introduced. A *bound property* will send a notification, by means of a PropertyChangeEvent, whenever the value of the property has changed to any registered PropertyChangeListeners. The attributes of the PropertyChangeEvent will identify which property has changed, what its previous value was and what its new value is. The listeners will then be able to take any actions required by the change.

For example, a *DynaLabel* instance may be connected to a news feed on the Internet, and a change of the *message* it displays indicates that some new news headline has been received. A PropertyListener, listening to changes of a bound *message* property, may use this information to alert the user, in some way, to the changed headline.

A *constrained property*, which must also be a bound property, has Vetoable ChangeListeners listening to it which are not only notified that a change of a property is proposed, also by means of a PropertyChangeEvent, but can also veto the proposed change by throwing a PropertyVetoException. Should such an exception be thrown in response to a proposed change, the bean will not implement the change. For example, the foreground and background properties of a DynaLabel instance may be both connected to a VetoableChangeListener that will throw a PropertyVetoException if the contrast between the proposed color combination is insufficient for comfortable viewing or if some inadvisable combination of colors is proposed.

This section will illustrate the use of constrained, and hence bound, properties, by implementing a *DynaBeanColorVetoer* class which will veto any proposed changes to the foreground or background color if the contrast between them is insufficient. While doing this the mechanisms which the BeanBox can use to transmit events between beans will be introduced.

EventListener → java.beans.PropertyChangeListener interface

```
public void propertyChange( PropertyChangeEvent event)
```

The propertyChange() method will be called after a bean property has been changed in order to allow the environment to take any appropriate actions.

EventListener → java.beans.VetoableChangeListener interface

```
public void vetoableChange( PropertyChangeEvent event)
                                    throws PropertyVetoException
```

The vetoableChange() method will be called when a bean property change is proposed in order to permit or deny it. The throwing of the PropertyVetoException indicates that the change has been vetoed.

The *DynaBeanColorVetoer* class will implement both the *PropertyChangeListener* and the *VetoableChangeListener* interfaces, requiring them to supply *propertyChange()* and *vetoableChange()* methods. The declaration of this class, and its context, is as follows.

```
package dynalabel;

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

public class DynaLabelColorVetoer extends Object
                       implements PropertyChangeListener,
                                  VetoableChangeListener {
```

The *propertyChange()* method, mandated by the *PropertyChangeListener* interface, will be called in response to the *DynaLabel setForeground()* and *setBackground()* methods after the change has been effected, as long as the change is not vetoed by the *vetoableChange()* method whose description will follow. The implementation of this method is as follows.

```
   public void propertyChange(PropertyChangeEvent event) {
   if ( (event.getPropertyName().equals( "foreground")) ||
        (event.getPropertyName().equals( "background")) ){

         System.out.println( event.getPropertyName() +
                    " changed from "              +
                    ((Color) event.getOldValue()) +
                    " to "                        +
                    ((Color) event.getNewValue()) );
      } // End if.
   } // End propertyChange.
```

The method is called with a single argument of the PropertyChangeEvent class which has four attributes: the source of the event, the propertyName (of class String), the oldValue (of class Object) and the newValue (of class Object). So when this method is called from a *DynaLabel* instance which has just changed its foreground color, the source will be the identity of the instance, the propertyName will be "*foreground*", the oldValue will be an instance of the Color class containing the color which the instance has just changed from and newColor will be the color it has just changed to.

Object → EventObject → java.beans.PropertyChangeEvent

```
public PropertyChangeEvent( Object source,   String propertyName
                            Object oldValue, Object newValue);

public String getPropertyName()
public Object getOldValue()
public Object getNewValue()
```

A PropertyChangeEvent is used by the PropertyChangeListener propertyChange() method and by the VetoableChangeListsner vetoableChange() method. The four attributes are the *source* of the event, the name of the property which is about to change, or which has changed, and its old and new values. A getSource() method is inherited from EventObject.

The implementation of the method merely reports on the terminal window that the foreground or background property has changed and states the old and new values. An example of the output it produces might be as follows.

```
foreground changed from java.awt.Color[r=100,g=100,b=100]
to java.awt.Color [r=0,g=0,b=0]
```

The only other method in the *DynaBeanColorVetoer* class is the *vetoableChange()* method, mandated by the VetoableChangeListener interface, which has the responsibility of throwing a PropertyVetoException if the contrast between the proposed new foreground color and the existing background color, or vice versa, is insufficient. The implementation of this method is as follows.

```
public void vetoableChange(PropertyChangeEvent event)
                            throws PropertyVetoException {

if ( (event.getPropertyName().equals( "foreground")) ||
     (event.getPropertyName().equals( "background")) ){

   Color backColor;
   Color foreColor;
   int   backIntensity;
   int   foreIntensity;
   int   contrast;

      if (event.getPropertyName().equals( "foreground")) {
         backColor = ((Component)
                       event.getSource()).getBackground();
         foreColor = (Color) event.getNewValue();
      } else {
         foreColor = ((Component)
                       event.getSource()).getForeground();
         backColor = (Color) event.getNewValue();
      } // End if.

      backIntensity = ( backColor.getRed() +
                        backColor.getGreen() +
                        backColor.getBlue()) /3;
      foreIntensity = ( foreColor.getRed() +
                        foreColor.getGreen() +
                        foreColor.getBlue()) /3;
      contrast = Math.abs( backIntensity - foreIntensity);

      if ( contrast < 25) {
      String explanation = event.getPropertyName() +
          " color change refused, insufficient contrast.";
         throw new PropertyVetoException(
                                     explanation, event);
      } // End if.
   } // End if.
} // End vetoableChange.
```

This action will only respond to PropertyChangeEvents associated with changes of the foreground and background colors and the **if** guard, on lines 0037 and 0038, ensures this. The local variables are then declared, on lines 0040 to 0044, and lines 0046 to 0049 will initialize the value of *backColor* to the background color of the source Component and *foreColour* to the proposed new foreground color if the *event*'s propertyName indicates that this is a proposed change of the foreground property. Otherwise, lines 0051 to 0053 will initialize the value of *foreColor* to the foreground color of the source Component and *foreColour* to the proposed new background color if it is a proposed change of the background property.

The effect of this **if** structure is that *foreColor* and backColor will contain the colors that will be used by the *DynaLabel* if the change is not vetoed. Lines 0056 to 0061 then initialize the *backIntensity* and *foreIntensity* local variables to the average intensity of their respective Color's red, green and blue composite values. Having done this, line 0062 computes the absolute difference between the two intensities and stores this in the local variable *contrast*.

The last step of this action will throw a PropertyVetoException if the *contrast* between the proposed color combination is insufficient. The arguments to the PropertyVeto Exception constructor are an *explanation* and the PropertyChangeEvent passed to the *vetoableChange* method.

Object → Throwable → Exception → java.beans.PropertyVetoException

```
public PropertyVetoException( String              explanation,
                              PropertyChangeEvent event);

public PropertyChangeEvent getPropertyChangeEvent()
```

A PropertyVetoException is thrown by the vetoableChange() method of the VetoableChangeListener interface in order to deny a proposed change. The explanation may be used to inform the user why the change could not be implemented.

The methods in the *DynaBeanColorVetoer* class will be called from a *DynaLabel* class instance whenever the *setForeground()* and *setBackground()* methods are called. In order for this to happen the *DynaLabel* class with have to have additional attributes, and methods, to allow it to be a source of PropertyChangeEvents which can be dispatched to PropertyChangeListeners and VetoableChangeListeners. These changes are similar to the registration of PropertyChangeListeners by the *DynaLabelColorEditor* class and are implemented as follows.

```
0041  private transient PropertyChangeSupport propertyMultiplexor =
0042                             new PropertyChangeSupport( this);
0043
0044  private transient VetoableChangeSupport vetoMultiplexor =
0045                             new VetoableChangeSupport( this);
```

The declaration of *propertyMultiplexor*, of the PropertyChangeSupport class, on lines 0041 to 0042 is essentially identical to the declaration of the *multiplexor* attribute in the *DynaLabelColorEditor* class, as previously described. It is declared with the modifier **transient** as it should not be serialized. The declaration of the *vetoMultiplexor* is very similar and provides support for the registration of *VetoableChangeListeners*. These multiplexors require methods to add and remove listeners to and from them, implemented as follows.

```
0274     public void addPropertyChangeListener(
0275                               PropertyChangeListener listener) {
0276        propertyMultiplexor.addPropertyChangeListener( listener);
0277     } // End addPropertyChangeListener.
0278
0279     public void removePropertyChangeListener(
0280                               PropertyChangeListener listener) {
0281      propertyMultiplexor.
0282                removePropertyChangeListener( listener);
0283      } // End removePropertyChangeListener.
0284
0285      public void addVetoableChangeListener(
0286                              VetoableChangeListener listener) {
0287         vetoMultiplexor.addVetoableChangeListener( listener);
0288      } // End addVetoableChangeListener.
0289
0290      public void removeVetoableChangeListener(
0291                               VetoableChangeListener listener) {
0292         vetoMultiplexor.removeVetoableChangeListener( listener);
0293      } // End removeVetoableChangeListener.
```

With these in place the *setForeground()* and *setBackground()* methods can be adapted so as to fire a PropertyChangeEvent to any registered VetoableChangeListeners before the change is effected and to fire a PropertyChangeEvent to any registered PropertyChangeListeners after the change has been made, if it is not vetoed first. The re-implementation of setForeground() would be as follows.

```
0107     public void setForeground( Color newForeground)
0108                             throws PropertyVetoException {
0109
0110     Color oldForeground = this.getForeground();
0111
0112        vetoMultiplexor.fireVetoableChange("foreground",
0113                                          oldForeground,
0114                                          newForeground);
0115
0116        super.setForeground( newForeground);
0117        this.setShowStyle( this.getShowStyle());
0118
0119        propertyMultiplexor.firePropertyChange("foreground",
0120                                             oldForeground,
0121                                             newForeground);
0122     } // End setForeground.
```

The method commences, on line 0112 to 0114, by firing a PropertyChangeEvent, via the *vetoMultiplexor*'s fireVetoableChange() method, to any registered VetoableChange Listeners. If the listener vetoes the proposed change it will throw a PropertyVetoException which, as noted in the prototype, will be propagated from the method. The throwing of the exception will also prevent lines 0116 and 0117, which effect the change of the *DynaLabel* foreground resource, from executing and hence effecting the veto. Having changed the value of the property, lines 0019 to 0121 fire a PropertyChangeEvent, via the *propertyMultiplexor*'s firePropertyChange() method, to any registered PropertyChange Listeners notifying them that the property has just changed.

However, this method cannot be implemented in this manner, as it is making a significant change to the AWT Component setForeground() method, by the addition of the **throws** clause. In order to get around this problem the name of the method, and of the resource in the bean info file, has been changed to *setDynaForeground()*, as follows.

```
0107     public void setDynaForeground( Color newForeground)
0108                              throws PropertyVetoException {
```

To complement this change a *getDynaForeground()*, implemented as follows, has to be supplied. In addition, the *setBackground()* and *getBackground()* methods have similarly been replaced with similar *setDynaBackground()* and *getDynaBackground()* methods.

```
0124     public Color getDynaForeground() {
0125        return this.getForeground();
0126     } // End getDynaForeground.
```

With these changes to the *DynaLabel* class made its bean info class, *DynaLabelBeanInfo* needs to be updated so that the BeanBox tool becomes aware of the changed nature of the properties. For example, the revised declaration, and configuration, of the *foregroundDescriptor* is as follows.

```
0020        PropertyDescriptor foregroundDescriptor =
0021            new PropertyDescriptor("dynaForeground",
0022                                                 dynaLabelClass);
0023            foregroundDescriptor.setPropertyEditorClass(
0024                                       DynaLabelColorEditor.class);
0025            foregroundDescriptor.setBound( true);
0026            foregroundDescriptor.setConstrained( true);
```

The changes call the Descriptor's setBound() and setConstrained() methods, with the argument **true**, to inform the BeanBox of the nature of the *dynaForeground* property.

Object → FeatureDescriptor → PropertyDescriptor → java.beans.PropertyDescriptor

```
public PropertyDescriptor( String propertyName,
                           Class  toIntrospect)
                                 throws IntrospectionException
```

The constructor will use the JavaBean introspection rules to locate the appropriate read and write methods for the *propertyName* in the Class instance *toIntrospect*. If a suitable pair of methods cannot be located then an IntrospectionException will be thrown. (More complex constructors can be used if the appropriate methods do not conform to introspection rules.)

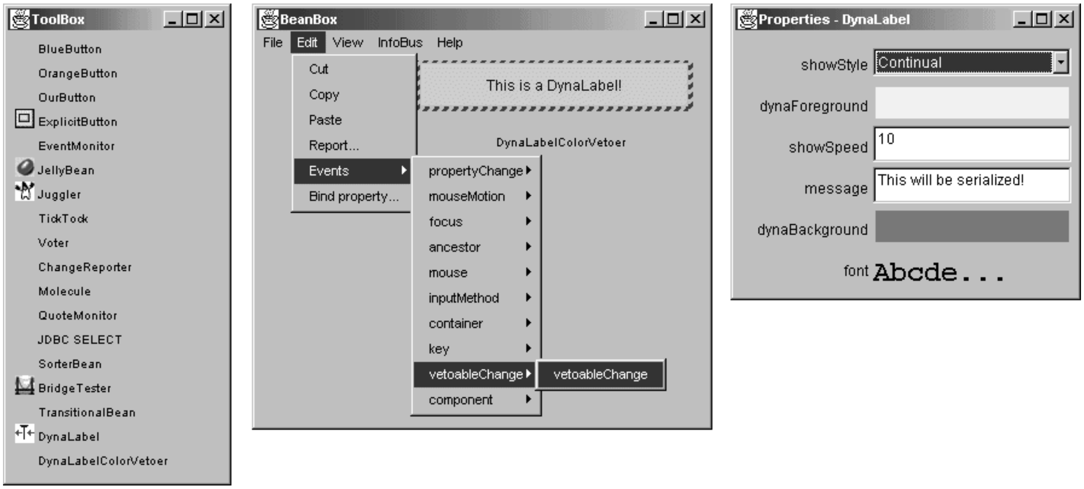
```
public void    setPropertyEditorClass( Class itsEditor)
public Class   getPropertyEditorClass()
public void    setBound( boolean yesOrNo)
public boolean isBound()
public void    setConstrained( boolean yesOrNo)
public boolean isConstrained ()
```

Attribute-setting and getting methods for the editor to be used in the properties sheet and for the bound and constrained nature of the property.

```
public Class  getPropertyType()
public Method getWriteMethod()
public Method getReadMethod()
```

Inquiry methods for the nature of the property and for its read and write methods.

Having made all of these changes, extending the manifest file to indicate that the *DynaBeanColorVetoer* is a bean and remaking the jar archive file, the BeanBox can be used to demonstrate the mechanisms. Figures 10.15 and 10.16 show the BeanBox, with



showStyle Continual

dynaForeground

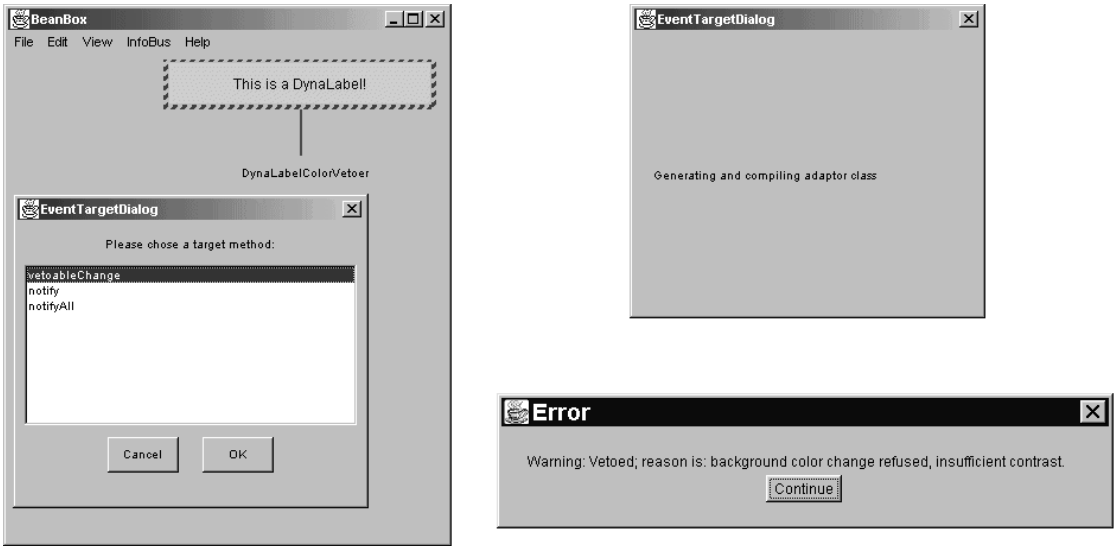
showSpeed 10

message This will be serialized!

dynaBackground

font Abcde...

Figure 10.15 Constrained property event handling (1).



Generating and compiling adaptor class

Warning: Vetoed; reason is: background color change refused, insufficient contrast.

Figure 10.16 Constrained property event handling (2).

the changed archive loaded, being used to investigate property change and veto change listening, The left-hand image on Figure 10.15 shows the ToolBox with the *DynaBean* and *DynaBeanChangeVetoer* beans loaded from the archive and available for use. The right-hand image shows the revised properties sheet with the revised labels for the foreground and background properites visible.

The work area, shown in the middle image, has had a *DynaLabel* instance and a *DynaBeanChangeVetoer* instance dropped on it. The *DynaBeanChangeVetoer* instance is an example of an invisible bean, which will not appear as a component on a user interface, and is illustrated on the work area as a rectangle containing just the name of the class. The *DynaLabel* instance has been selected and the *Edit* menu *Events...* option selected. Introspection on the *DynaLabel* bean has revealed to the BeanBox tool the events that this instance is a source of. One of these possibilities is vetoableChange, and this option has been selected to reveal a further cascading menu which indicates that there is a single category of events also shown as vetoableChange. If the *mouse* option on the menu had been chosen then this would have revealed a number of possible categories of event, including mousePressed, mouseReleased and mouseClicked.

Activating the vetoableChange event on the cascading menu leads to the situation shown at the top right of the left-hand image of Figure 10.16. The thick line originating from the *DynaLabel* instance can be dragged and used to connect the event source, the *DynaLabel* instance, to the *target* event listener, the *DynaBeanChangeVetoer* instance, as shown. Introspection of the *DynaBeanChangeVetoer* instance will reveal the possible methods in the target bean which can be called in response to the receipt of the event. These are all the methods that take no arguments, those that take an Object as a single argument and those which take an argument of the event class being dispatched, *PropertyChangeEvent*. These methods are presented, for the user to chose from, on the dialog shown at the top of the left-hand image in Figure 10.16.

The appropriate method to connect to in this situation is *vetoableChange()*, as previously described. Selecting this method in the *EventTargetDialog* and then clicking the "*OK*" button causes the working dialog message, shown at the top right of Figure 10.16, to be displayed. The BeanBox is writing the source code for an *adapter class* which will be able to receive the *PropertyEvent* generated by the *DynaBean* and pass it as an argument to the *DynaBeanChangeVetoer*'s *vetoableChange()* method. It will then compile the class it has written, load it into the BeanBox, create an instance of it, and register it as the *DynaBean*'s *VetoableChangeListener*. The effect is the same as if the *DynaBean ChangeVetoer* instance had been registered directly as the *DynaBean*'s *Vetoable ChangeListener*.

The same process was followed to register the *DynaBeanChangeVetoer* instance's *propertyChange()* method as the destination of the events generated by the *DynaLabel*'s propertyChange capability. When the *dynaForeground* or *dynaBackground DynaBean ColorEditor* is used the consequence of the call of the *DynaLabel*'s *setDynaForeground()* or *setDynaBackground()* call is for the *vetoableChange()* method of the *DynaBeanChange Vetoer* instance to be called before the change is effected. Should the contrast between the proposed *foreground* and *background* properties be insufficient, the *Error* dialog, shown at the bottom right of Figure 10.16, will be posted to alert the user that the change has not taken place. The message shown on the *Error* dialog contains the *explanation* obtained from the PropertyVetoException thrown by the *DynaBeanChangeVetoer*'s *vetoableChange()* method. What is not shown in Figure 10.16 are the messages output to the terminal window every time the proposed change is not vetoed and the

DynaBeanChangeVetoer's *propertyChange()* method is called. An example of the format of the messages has already been given when the implementation of the *propertyChange()* was described. In an alternative, and more realistic, scenario, a color change on one component on an interface should be propagated to all other components on an interface in order that a consistent appearance can be maintained. Responsibility for such propagation would lie with the *propertyChange()* method.

10.9 Other beans considerations

The description of the introspection protocols given above applied only to *simple* properties. There are a number of other kinds of properties that can also be introspected. The first of these are **boolean** properties, which can be identified by the existence of a pair of methods such as:

```
public void    setEnabled( boolean yesOrNo)
public boolean isEnabled()
```

This pair of methods allows an inference to be made that there is a **boolean** property called *enabled* associated with the bean. The BeanBox properties sheet will automatically supply a pull-down menu with the options "*True*" and "*False*" to edit the value of these properties.

The remaining kind of property is an indexed property which can be inferred from the existence of a set of methods such as:

```
public void      setMessages( String[] newMessages)
public String[]  getMessages()
public void      setMessages( int thisOne, String newMessage)
public String    getMessages( int thisOne)
```

This set allows an inference to be made that there is an indexed property called *messages*; the four methods allow all, or just one, of the *messages* to be set or retrieved.

All of the properties discussed so far have been read–write properties, and it is only read–write properties that can be supported on a BeanBox property sheet. It is also possible for a property to be read-only, having an inferred method to read its value but none to retrieve it, or write-only, having an inferred method to set it but none to retrieve it. Some parts of the BeanBox, which have not been presented in this chapter, allow such properties to be manipulated.

The Web site which supports this book contains details of another JavaBeans case study describing a bean, the classes that support it and the protocols involved. The address of the Web site is contained in the Appendix.

Summary

- A JavaBean is a component architecture which is intended to allow specialized components to interoperate effectively and also to be used by bean-aware software tools.
- Properties of a bean include introspection, customization, event handling, properties and persistence.
- Introspection can be implicit by a set of rules applied to the names of a bean's methods or, preferably, can be explicit by the provision of a systematically named BeanInfo file.
- A bean, and its supporting classes and other resources, must be packaged into a Java archive (jar) file to be made available to a bean-aware tool.

- ♦ The properties shown on a bean-aware properties sheet will be restricted to those that are revealed by introspection.
- ♦ A bean environment supplies default editor components for many property classes, and if a suitable editor cannot be located will not place the property onto the property sheet. However, in many cases these editors are inadequate.
- ♦ A specialized property editor for manifest values can be simply implemented by extending the java.beans.PropertyEditorSupport class.
- ♦ A more complex specialized property editor can be produced by implementing the PropertyEditor interface.
- ♦ The serializable interface mandates no methods but indicates that a class has been constructed with regard to the possibility of it being stored in a persistent state or migrated across a network connection.
- ♦ Any attributes which should be stored as part of the persistent state should be declared with the modifier **transient**.
- ♦ Serialization may require the implementation of readObject() and writeObject() methods, although the supplied defaultReadObject() and defaultWriteObject() methods are sometimes sufficient.
- ♦ A bound property is one whose class dispatches a PropertyChangeEvent to a PropertyChangeListener's propertyChange() method after its value has changed.
- ♦ A constrained property, which should also be a bound property, is one whose class dispatches a PropertyChangeEvent to a VetoableChangeListener's vetoableChange() method before its value has changed. The listener can veto the change by throwing a PropertyVetoException.

Exercises

10.1 Investigate the behavior of a *DynaLabel* instance when a large italic font is used. In some circumstances the bit blasting of the off-screen image onto the on-screen window does not completely remove the previous image. Devise, implement and investigate various ways of curing this fault.

10.2 Extend the *DynaLabel* class to support other animated behavior, for example continual animation from left to right or animating one character of the message into position at a time. When this has been accomplished extend the *DynaLabelShowStyleEditor* to support these changes, repackage the bean and load it into a bean-aware tool.

10.3 Design, implement, package and test a *DynaLabelShowSpeedEditor* class which supplies a JSlider to control the speed of the animation.

10.4 Extend the *DynaLabel* class so that it supports an array of messages showing each one in turn. Make the methods to support the messages bean introspection-compliant and investigate what default support is provided for them in the BeanBox.

10.5 Revisit the *TimeInput* component from Chapter 7 and provide it with a *faceStyle* attribute which determines the decoration on the clock face, for example plain, 4 numeral, 12-numeral or roman-numeral styles. Package the *TimeInput* component in a jar file together with a suitable BeanInfo and *TimeInputFaceStyleEditor* class, import it into the BeanBox and demonstrate its effectiveness.

10.6 Consider any further bean support which is essential, or desirable, for the *TimeInput* component and then implement it in a bean in a suitable manner and demonstrate it in the BeanBox.

10.7 Many Web sites contain collections of public domain beans. Some addresses are given in the Appendix. Visit one of these Web sites, obtain the jar file of a bean that interests you, and investigate it within the BeanBox.

10.8 Revisit any of the specialized components that have been introduced in previous chapters and re-implement them as beans.

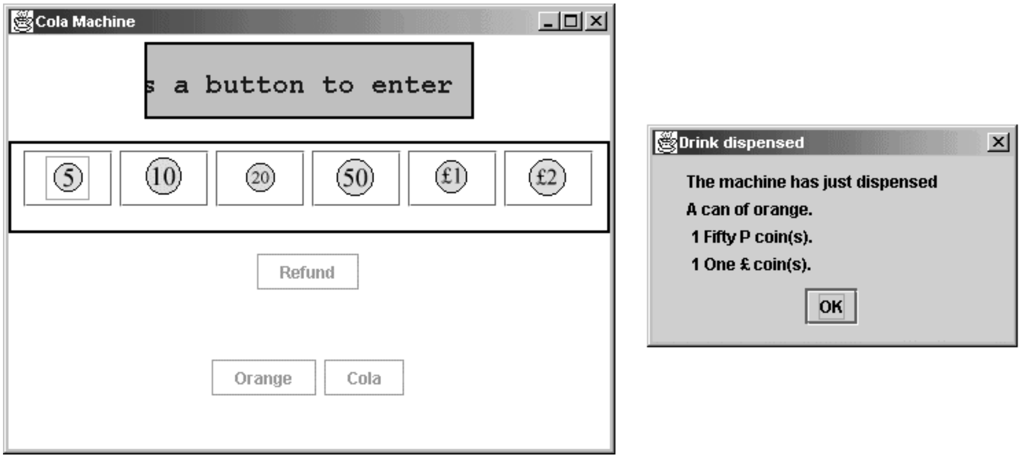
11

The *ColaMachine*: introducing Internationalization

11.1 Introduction

This chapter will consolidate the techniques of designing for usability using STDs, first introduced in Chapter 1, by illustrating the design and development of a complex artifact. The example chosen is a simulation of a soft drink vending machine, which will be known as a *ColaMachine*. The appearance of the artifact, when it first becomes visible to the user, is illustrated in the left-hand image in Figure 11.1. The component parts of the interface will be described in detail below. The right-hand image shows how the artifact simulates the dispensing of a soft drink and returning the change, if any, to the user.

The implementation shown in Figure 11.1 is configured for use in the UK, showing representations of the coins that are available in the UK and displaying the messages to the user in British-style English. The first implementation of the artifact will only be capable of this configuration, but once the design and implementation of the various parts of the simulation have been described, parts of it will be revisited so as to make it suitable for international use. In order to do this any parts of the artifact that are dependent upon the *locale* in which it is to operate will have to be identified and removed into a *resource file*. The mechanisms by which alternative resource files for different



```
s a button to enter
```

```
The machine has just dispensed
A can of orange.
 1 Fifty P coin(s).
 1 One £ coin(s).
```

Figure 11.1 *ColaMachine*: visual appearance.

locales can be supplied and automatically used by the simulation will then be described and illustrated.

Partly because Java was intended almost from the outset for global use, most noticeably to provide interactive content on the Web, its mechanisms to support its use in different locales are among the most powerful provided by any development environment. The development strategy being used in this chapter, first producing the artifact and then retrofitting it with international capability, is not recommended. It would be more advisable to build for international use from the outset. However, this would further complicate the description of its essential functionality, in the first part of this chapter, so this will not be attempted.

A context diagram for the complete artifact is given in Figure 11.2. The central object, an instance of the *ColaMachine* class identified as *colaMachine*, provides the translation layer, listening to the interactive interface components shown on the left of the diagram, providing feedback on the operations on the *messagePanel* and various dialogs at the top and bottom of the diagram and making use of the application functionality supplied by the *CanHopper* and *CoinHopper* instances shown at the right. The description of the construction of this artifact will commence with the hoppers shown at the right of the diagram.

11.2 The *CanHopper* and *CoinHopper* classes

A *hopper* is a pile of real-world objects, such as coins or cans, which can be refilled from the top and can release objects from the bottom. They are occasionally visible in some real-world soft drink vending machines so that users can see whether the hopper containing the drink of their choice is sold out. A class hierarchy diagram for the hopper classes is given in Figure 11.3. The diagram shows that a specialized exception class, called *HopperException*, which extends the Java Exception class, will be supplied. Instances of this exception will be thrown by instances of the two hopper classes, *CanHopper* and *CoinHopper*, if an attempt is made to dispense from an empty hopper.

The implementation of *HopperException* class need only override one constructor and is implemented as follows. The *reason* argument to the constructor will be displayed by the run-time environment if the exception is ever raised, indicating exactly which fault occurred.

```
package hoppers;

public class HopperException extends Exception {

   public HopperException( String reason) {
      super( reason);
   } // End HopperException.
} // End class HopperException.
```

A class diagram for the *CanHopper* class is given in Figure 11.4. The diagram shows that the class has two encapsulated attributes, a String *description* of what is stocked and the number of *cans* currently in the hopper. Both constructors must supply a String describing *whatType* of can is being stored. The second constructor can also indicate the *numberOfCans* in the hopper, the first constructor will use a default *numberOfCans*. The *restock()* method adds the *numberOfCans* supplied as an argument to those held in the hopper. The *isEmpty()* method will return **true** if the number of *cans* in the hopper is zero and the *dispense()* action will reduce the value of this attribute by 1, throwing a

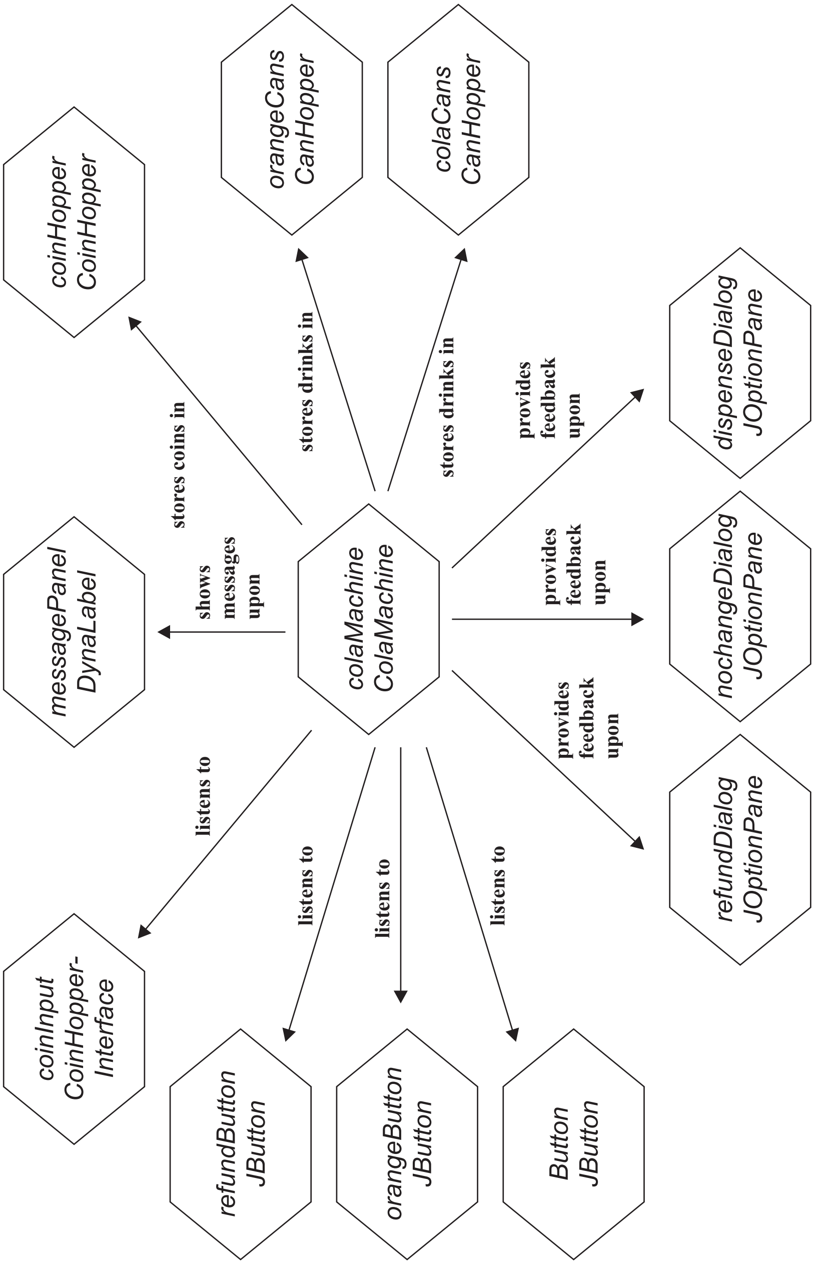


Figure 11.2 *ColaMachine*: context diagram.

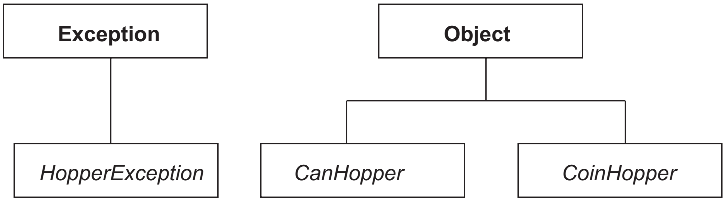


Figure 11.3 *ColaMachine*: hopper classes hierarchy diagram.

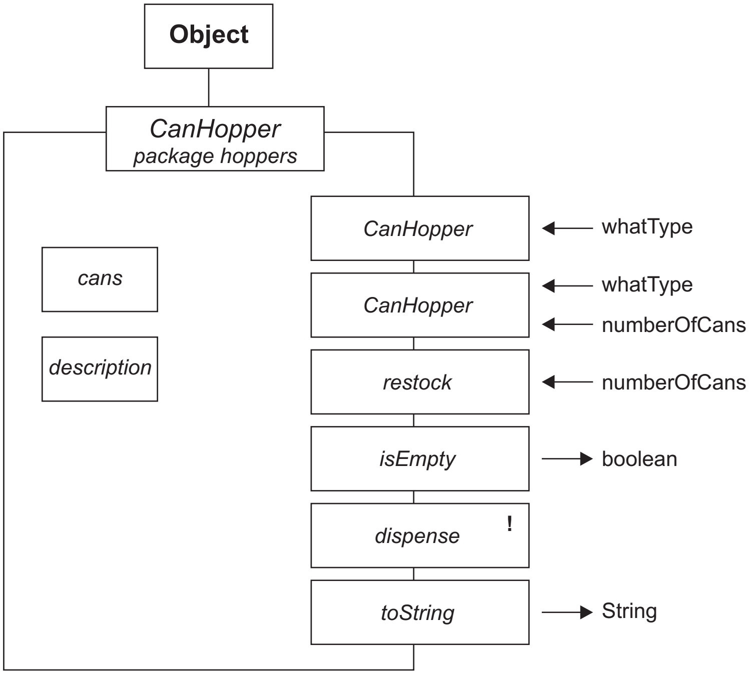


Figure 11.4 *CanHopper*: class diagram.

HopperException if it is called when the hopper is empty. Finally, the *toString()* method returns a description of the state of the hopper. The implementation of this design, presented without comment, is as follows.

```
package hoppers;

public class CanHopper extends Object {

private final static int DEFAULT_RESTOCK_LEVEL = 25;

private String description = null;
private int    cans        = 0;

   public CanHopper( String whatType) {
      this( whatType, DEFAULT_RESTOCK_LEVEL);
   } // End constructor.

   public CanHopper( String whatType,
                     int    numberOfCans) {
      description = new String( whatType);
      cans        = numberOfCans;
   } // End principal constructor.

   public void restock( int numberOfCans) {
      cans += numberOfCans;
   } // End restock

   public boolean isEmpty() {
      return cans == 0;
   } // End isEmpty.

   public void dispense() throws HopperException {
      if ( this.isEmpty()) {
         throw new HopperException(
                      "Attempt to dispense from an " +
               "empty " + description + " can hopper");
      } else {
        cans-;
      } // End if.
   } // End dispense.

   public String toString() {
      return "A can hopper dispensing " + description +
             " with " + cans + " cans.";
   } // End toString.
} // End class CanHopper.
```

This is not a strictly accurate model of a real-world can hopper, mainly because it has no limit upon the number of cans which it can store. However, it is sufficient for the purpose of this chapter. No demonstration client for this class will be presented, although if it were being produced as a part of a development project one should be supplied and used to indicate that the implementation is correct before construction continues. The *CoinHopper* is a more accurate model of a real-world hopper; its class diagram is given in Figure 11.5.

The diagram shows that there is limit (*MAXIMUM_COINS*) on each of the *coin stacks* which are contained in the hopper. The other manifest value, *RESET_LEVEL*, is used by the *resetHopper()* method to indicate the number of coins which should be left in each stack when the machine is serviced. The argument to the single constructor, *values[]*, is an array of **int** whose length indicates the number of stacks which are required and each individual element indicates the value of the denomination of coin stored in each stack. Hence to initialize a *CoinHopper* for use in the UK, as indicated in Figure 11.1, the argument would contain {5, 10, 20, 50, 100, 200}. To initialize a coin hopper for use in the USA, assuming a mechanism to accept one-dollar bank notes, the argument would contain {5, 10, 25, 100}. The *values[]* argument is stored in the *coinValues[]* attribute and the *coins[]* array attribute is initialized to the length of the *values[]* argument.

The *depositCoin()* method simulates the addition of a coin to the hopper, and the argument *coinType* identifies which denomination of coin and hence which stack. Assuming that the stack is not full the coin will be added to it. If the stack is full then the value of

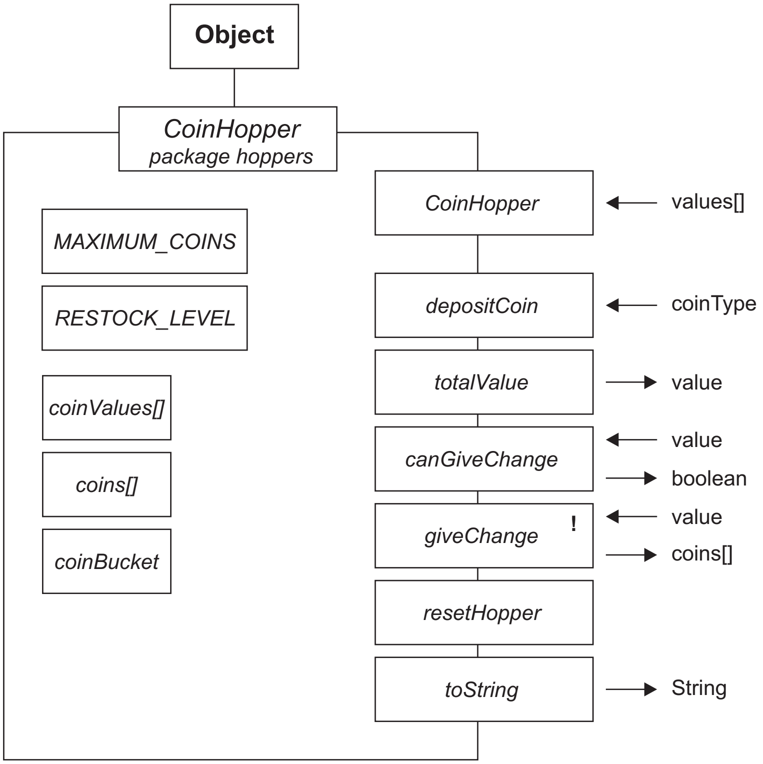


Figure 11.5 *CoinHopper*: class diagram.

coinType is added to the *coinBucket* attribute. Hence to simulate the insertion of a UK 50p coin the value of *coinType* would be 3, identifying which element of the *coins[]* array to add it to. If this stack is not full, i.e. if the value of the third element in the array is less than *MAXIMUM_COINS*, then the coin will be added to the stack, i.e. if the value of the third element in the array will be incremented. If the stack is full, then, in the real world, the coin is allowed to fall into a collection bucket, from where it can never be returned by the machine in change given to the user.

The *totalValue()* inquiry method returns the total value of all coins held in the stacks and in the bucket. The *canGiveChange()* method will return **true** if it is possible for the stacks to return a combination of coins whose value is equal to that of the value argument, and **false** otherwise, but will not change the state of the stacks. The *giveChange()* method should be called immediately after *canGiveChange()* has indicated that *value* can be obtained from the stacks. However, it is implemented defensively and will throw a *HopperException* if the value of change requested cannot be provided. The object returned from this method is an array of **int** whose length is equal to the number of stacks and the value of each element indicates the number of coins of each denomination returned.

The *resetHopper()* method is called from the constructor and empties the hopper, leaving *RESTOCK_LEVEL* of coins in all stacks, apart from the maximum denomination stack, which is left empty. Finally, the *toString()* method supplies a description of the state of the hopper. The implementation of this class, as far as the end of the constructor, is as follows.

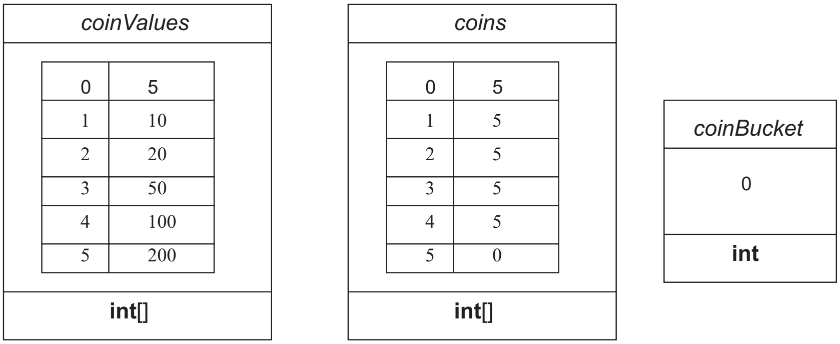
```
0010  package hoppers;
0011
0012  public class CoinHopper extends Object {
0013
0014  private static final int MAXIMUM_COINS = 20;
0015  private static final int RESTOCK_LEVEL = 5;
0016
0017  private int[] coinValues = null;
0018  private int[] coins      = null;
0019  private int   coinBucket = 0;
0020
0021     public CoinHopper( int[] values) {
0022        super();
0023        coinValues = values;
0024        coins = new int[ coinValues.length];
0025        this.resetHopper();
0026     } // End CoinHopper constructor.
```

The two manifest values are declared on lines 0014 and 0015 and the three instance attributes on lines 0017 to 0020. The constructor commences on line 0022 by calling its **super**, Object, constructor and then stores the *values* array supplied as an argument in the *coinValues* attribute. Line 0024 then constructs the *coins* array with a length equal to that of the *coinValues* array and line 0025 initializes the state of this array by calling *resetHopper()*, whose implementation is as follows.

```
0113     protected void resetHopper() {
0114          for( int thisStack = 0;
0115                   thisStack < coinValues.length -1;
0116                   thisStack ++ ){
0117            coins[ thisStack] = RESTOCK_LEVEL;
0118         } // End for.
0119         coins[ coins.length -1] =0;
0120         coinBucket = 0;
0121     } // End resetHopper.
```

The effect of this action is to initialize, or reset, all of the coin stacks apart from the highest denomination to the *RESTOCK_LEVEL*. Both the highest denomination stack and the *coinBucket* are emptied by being set to zero. Figure 11.6 contains a visualization of a newly constructed *CoinHopper* configured for use in the UK. The implementation of the *depositCoin()* and *totalValue()* methods is as follows.

```
0029     public void depositCoin( int coinType) {
0030        if ( coins[ coinType] < MAXIMUM_COINS) {
0031           coins[ coinType]++;
0032        } else {
0033           coinBucket += coinValues[ coinType];
0034        } // End if.
0035     } // End depositCoin.
0036
0037
0038     public int totalValue() {
0039
0040     int theValue =0;
```



	coinValues
0	5
1	10
2	20
3	50
4	100
5	200

	coins
0	5
1	5
2	5
3	5
4	5
5	0

Figure 11.6 *CoinHopper*: initial UK state.

```
0041
0042          for( int thisStack = 0;
0043                 thisStack < coinValues.length;
0044                 thisStack++ ){
0045             theValue += coins[thisStack] *
0046                        coinValues[thisStack];
0047          } // End for.
0048          theValue += coinBucket;
0049          return theValue;
0050       } // End totalValue.
```

The argument, *coinType*, to the *depositCoin()* method indicates the denomination of coin which is to be deposited into the hopper. If the appropriate stack is not full, as determined in the **if** condition on line 0030, the number of coins stored in the stack is incremented on line 0031. Otherwise, if the appropriate stack is full, the value of the coin, obtained from the *coinValues* array, is added to the *coinBucket* on line 0033. Using the visualization in Figure 11.6, if a 50p coin were to be deposited the value of *coinType* would be 3 and the consequence of the action would be that the value of the *coins* array element 3 would be incremented from 5 to 6. If the value of this element of the *coins* array was 20 (*MAXIMUM_COINS*) then the consequence would have been to add the value stored in the third element of the *coinValues* array (50) to that in *coinBucket*.

The *totalValue()* method iterates through each stack in the array, multiplying the number of *coins* in each stack by the corresponding *coinValue* and adding this to *theValue*. The value of *coinBucket* is added to *theValue* before it is returned from the method, on line 0049. Using the visualization in Figure 11.6, the value returned from the method would be 5*5 + 5*10 + 5*20 + 5*50 + 5*100 + 0*200 + 0, which is 925 (pence). The implementation of *canGiveChange()* is as follows.

```
0053       public boolean canGiveChange( int value) {
0054
0055       int[] tempCoins      = new int[ coins.length];
0056       int   remainingValue = value;
0057       int   thisStack;
0058
0059          for( int thisStack = 0;
0060                 thisStack < coinValues.length;
0061                 thisStack++ ){
```

```
0062            tempCoins[ thisStack] = coins[ thisStack];
0063         } // End for.
0064
0065         for( int thisStack = coinValues.length -1;
0066                  thisStack >= 0;
0067                  thisStack-){
0068            while ( (tempCoins[ thisStack] > 0)                   &&
0069                    (remainingValue >= coinValues[ thisStack]) ){
0070               remainingValue -= coinValues[ thisStack];
0071               tempCoins[ thisStack]-;
0072            } // End while.
0073         } // End for.
0074         return remainingValue == 0;
0075     } // End canGiveChange.
```

This method will return **true** if it is possible for the *CoinHopper* to return a set of coins whose value is equal to the argument *value* and **false** otherwise. It is essential that this method does not actually make any changes to the state of the stacks, so lines 0055 and 0059 to 0063 make a copy of the *coins* array in the *tempCoins* array. The other initialization step, on line 0056, copies the argument *value* to the local variable *remainingValue*.

The major part of the action is implemented in the double loop commencing on line 0066. The outer **for** loop iterates through all the stacks, indexed by *thisStack*, commencing with the largest denomination stack. The **for** loop body considers each individual stack and the **while** loop will, on each iteration, deduct the value of the stack from *remainingValue* (line 0070) and remove the appropriate coin from *tempCoins* (line 0071). This iteration will continue so long as there are coins remaining in *thisStack* (line 0068) and the value of *remainingValue* is greater than the denomination of *thisStack* (line 0069). At the termination of the **for** loop, if it is possible for the change to be made then the value in *remainingValue* will be zero, and the **boolean** expression evaluating this, on line 0074, is returned as the value of the method.

This method can be partially validated by considering a request to make change equal to 926 (pence) from the *CoinHopper* illustrated in Figure 11.6. In this situation every coin will be removed from the *tempCoins* copy of the *coins* array, a total of 925 pence, leaving the value 1 in *remainingValue*, which will cause the method to return **false**. The implementation of *giveChange()*, as follows, is very similar to *canGiveChange()*.

```
0079     public int[] giveChange( int value) {
0080
0081     int[] returnedCoins  = new int[ coins.length];
0082     int   remainingValue = value;
0083
0084         if ( ! this.canGiveChange( value)) {
0085            throw new HopperException(
0086                           "Coin Hopper give change error " +
0087                           value + " requested, hopper contains +
0088                           this.toString());
0089         } // End if.
0090
0091         for( int thisStack = coinValues.length -1;
0092                  thisStack >= 0;
0093                  thisStack- ){
```

```
0094            while ( (coins[ thisStack] > 0)                 &&
0095                    (remainingValue >= coinValues[ thisStack]) ){
0096               coins[ thisStack]-;
0097               returnedCoins[ thisStack]++;
0098               remainingValue -= coinValues[ thisStack];
0099            } // End while.
0100         } // End while.
0101         return returnedCoins;
0102     } // End giveChange.
```

The action commences, on lines 0084 to 0089, by throwing a *HopperException*, with a full *reason*, if the *canGiveChange()* method indicates that it is not possible to return the *value* requested in the argument from the hopper. If the exception is not thrown then the double loop, commencing on line 0091, is controlled by exactly the same considerations as the *canGiveChange()* double loop described above. The only difference is that as each coin is removed from the *coins* array it is added to the *returnedCoins* array which was constructed on line 0081. This array is returned from the method, on line 0101, indicating the coins which are to be returned to the user. The remaining method, *toString()*, is implemented, without comment, as follows.

```
0116     public String toString() {
0117
0118     StringBuffer reply = new StringBuffer( "");
0119
0120         for( int thisStack = 0;
0121                  thisStack < coinValues.length;
0122                  thisStack++ ){
0123            reply.append( coins[ thisCoin] +
0124                              "   "            +
0125                           coinValues[ thisCoin] +
0126                              " coins.\n");
0127         } // End for.
0128         reply.append( "\n" + coinBucket +
0129                          " in the bucket.\n");
0130         reply.append( "\nTotal " +
0131                          this.totalValue() + ".\n");
0132         return reply.toString();
0133     } // End toString;
```

If this method were called, with the *CoinHopper* in the state shown in Figure 11.6, the String returned would be as follows.

```
5   5 coins.
5   10 coins.
5   20 coins.
5   50 coins.
5   100 coins.
0   200 coins

0 in the bucket.

Total 925.
```

As with the *canHopper* class, described above, a thorough demonstration of this class should be made before its correct implementation should be relied upon for the rest of the project. In the interests of keeping this book manageably short such a demonstration client will not be supplied, although one was produced and used while this chapter was being prepared. However, a partial demonstration of the class will be supplied when the *CoinHopperUI* class, which follows, is demonstrated.

11.3 The *CoinHopperUI* class

An instance of the *CoinHopperUI* class is visible on the *ColaMachine* interface illustrated in the left-hand image in Figure 11.1. The interface supplied contains a number of JButton instances, each of which displays an image representing the appearance of the coin it corresponds to. It might seem that as JButton instances generate ActionEvents when the user interacts with them these events should be propagated from the *CoinHopperUI* interface to whatever Object is interested in listening to it. However, the requirement in this situation is for the user to select one element from a number of choices offered, and this requirement is better met by dispatching an instance of the ItemEvent class. Hence the ActionEvents generated by the JButtons on the interface will be listened to by the interface instance itself and notification provided to its listener by means of an ItemEvent. The class diagram for the *CoinHopperUI* class is given in Figure 11.7.

The diagram indicates that the *CoinHopperUI* class extends JPanel, is a member of the hoppers package of classes and implements the ActionListener interface, in order to be able to listen to the ActionEvents generated by the JButtons it contains. It also has to

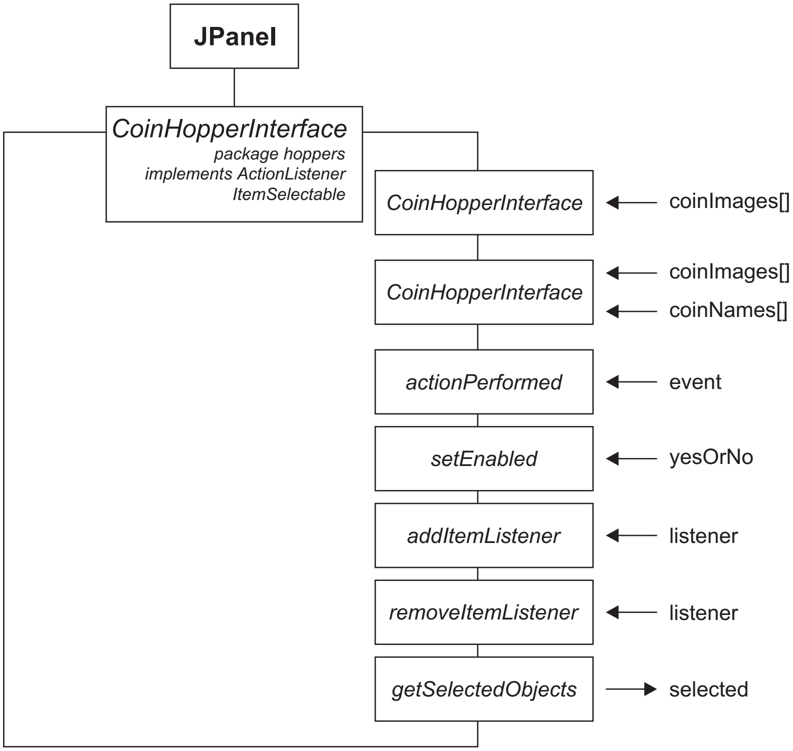


Figure 11.7 *CoinHopperUI*: class diagram.

implement the ItemSelectable interface in order that the handling of the ItemEvents which it generates can be correctly processed.

The first constructor takes an array of Icon, called *coinIcons*, which contains the images to be shown on each of the buttons. The second constructor also requires an array of String, called *coinNames*, which it will install as the *toolTipText* resource of the buttons. The *actionPerformed()* method listens to the ActionEvents generated by the buttons and dispatches ItemEvents to the list of ItemListeners maintained by the *addItemListener()* and *removeItemListener()* methods. The absence of an exception on the *addItemListener()* method indicates that instances of the *CoinHopperUI* class are event multicasters.

The *setEnabled()* method will, according to its **boolean** argument enable or disable all the buttons on the interface. The last method, *getSelectedObjects()*, is required to implement the ItemSelectable interface, but will be provided as a null method. The implementation of this design, as far as the end of the first constructor, is as follows.

```
0010  package hoppers;
0011
0012  import javax.swing.*;
0013  import javax.swing.border.*;
0014
0015  import java.awt.*;
0016  import java.awt.event.*;
0017
0018
0019  public class CoinHopperUI extends JPanel
0020                         implements ActionListener,
0021                                    ItemSelectable {
0022
0023  JButton[]    buttons     = null;
0024  ItemListener itsListener = null;
0025
0026    public CoinHopperUI( Icon[] coinIcons) {
0027         this( coinImages, null);
0028     } // End alternative constructor.
```

The package declaration, required importations and class declaration contain no surprises. Line 0023 declares, but does not initialize, an array of JButton called *buttons* and line 0024 an instance of the ItemListener class called *itsListener*, which will be used to contain the list of listeners to which ItemEvents will be dispatched. The first constructor indirects to the second, principal, constructor, passing **null** as the second argument. The principal constructor is implemented as follows.

```
0031     public CoinHopperUI( Icon[]   coinIcons,
0032                                 String[] coinNames) {
0033        super();
0034        buttons = new JButton[ coinImages.length];
0035        for ( int coinIndex =0;
0036                  coinIndex < coinImages.length;
0037                  coinIndex++ ) {
0038           buttons[ coinIndex] = new JButton(
0039                                   coinImages[ coinIndex]);
0040           if ( coinNames != null) {
0041              buttons[ coinIndex].setToolTipText(
0042                                     coinNames[ coinIndex]);
0043           } // End if.
0044
0045           buttons[ coinIndex].setBackground( Color.white);
0046           buttons[ coinIndex].setOpaque( true);
0047           buttons[ coinIndex].addActionListener( this);
0048           buttons[ coinIndex].setActionCommand(
0049                        new Integer(coinIndex).toString());
0050           this.add( buttons[ coinIndex]);
0051        } // End for.
0052
0053        this.setBorder( new LineBorder( Color.black, 2));
0054     } // End CoinHopperUI principal constructor.
```

The constructor commences, on line 0033, by calling the **super**, JPanel, constructor and continues by initializing the size of the *buttons* array to that of the number of Icons supplied in the *coinIcons* array. The **for** loop commencing on line 0035 then constructs and configures each JButton instance in turn. Lines 0038 and 0039 construct the JButton instance, using a constructor that specifies only the icon which it is to display. Lines 0040 to 0043 then install a *toolTipText* resource using the String passed in the *coinNames* array argument, if any. Lines 0045 and 0046 then configure the visual appearance of the button before line 0047 registers **this** instance being constructed as its own listener. Lines 0048 and 0049 install as the actionCommand resource of the JButton a String containing the ordinal location of the button. Thus the first button in the interface will have "0" installed as its actionCommand, the second "1", the third "2", and so on. The final step in the loop is to add() the JButton to **this** JPanel which, as it has a default FlowLayout, will lay out the buttons in a single row from left to right, as shown in Figure 11.1. The final step in the constructor establishes a two-pixel black line Border within the panel.

The implementation of the *setEnabled()* method calls the setEnabled() method of each JButton in the array, as follows.

```
0070     public void setEnabled( boolean yesOrNo){
0071        for ( int coinIndex =0;
0072                  coinIndex < buttons.length;
0073                  coinIndex++ ) {
0074            buttons[ coinIndex].setEnabled( yesOrNo);
0075        } // End if.
0076     } // End setEnabled.
```

The *addItemListener()* and *removeItemListener()* methods make use of facilities supplied by the AWTEventMulticaster class, as follows.

```
0079    public synchronized void addItemListener(
0080                                     ItemListener listener){
0081       itsListener = AWTEventMulticaster.add(
0082                                      itsListener, listener);
0083    } // End addItemListener.
0084
0085
0086    public synchronized void removeItemListener(
0087                                     ItemListener listener){
0088       itsListener = AWTEventMulticaster.remove(
0089                                     itsListener, listener);
0090    } // End removeItemListener.
```

The AWTEventMulticaster class simplifies the maintenance of a list of AWT listeners. Its add() method, called on line 0081 and 0082, will add the second ItemListener argument to the list of ItemListener instances stored in the first ItemListener argument and return the extended list. The effect of the lines in this example is to add another listener to those already stored in the itsListener ItemListener attribute of the *CoinHopperUI* class. Similarly the AWTEventMulticaster remove() method removes the ItemListener in the second argument from the list in the first argument returning the depleted list.

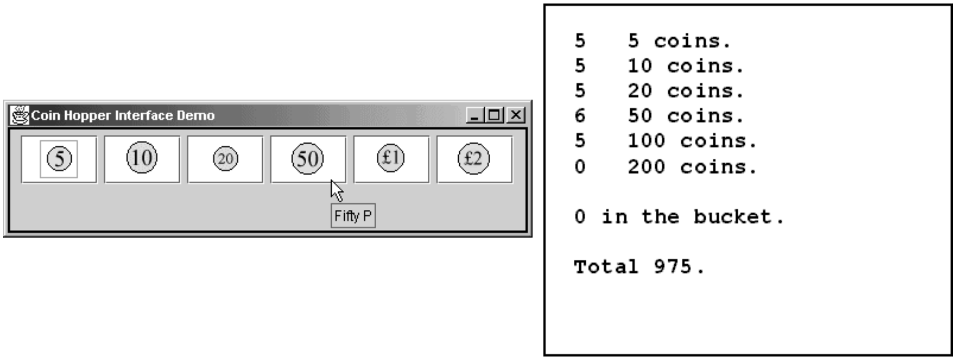
The AWTEventMulticaster add() and remove() methods should be used whenever an extended class has a need to maintain a list of AWT listeners. It not only simplifies the code which the developer has to produce but is also inherently thread safe. The *actionPerformed()* method of the *CoinHopperUI* class makes use of the list maintained when an event has to be dispatched, as follows.

```
0059     public void actionPerformed( ActionEvent event) {
0060        if ( itsListener != null) {
0061           itsListener.itemStateChanged(
0062                         new ItemEvent( this,
0063                               ItemEvent.SELECTED,
0064                     new Integer( event.getActionCommand()),
0065                             ItemEvent.ITEM_STATE_CHANGED));
0066        } // End if.
0067     } // End actionPerformed.
```

This method will be called every time the user presses one of the buttons on the panel and the *event*'s actionCommand resource will be a String containing the ordinal location of the button on the interface. This method will only do anything if there are any listeners stored in the *itsListener* ItemListener list, that is if its value is not **null**. The action taken is to call the itemStateChanged() method of the *itsListener* instance, passing as an argument an instance of the ItemEvent class constructed on lines 0062 to 0065. The consequence of this call will be that the itemStateChanged() method of every ItemListener stored in the list of listeners in *itsListener* will be called. Each will receive a copy of the ItemEvent passed to this class, although there is no guarantee of the sequence in which the listeners will be notified.

An ItemEvent has four attributes which must be supplied as arguments to the constructor on lines 0062 to 0065. The first argument, **this**, is the source of the event, and the second argument is the manifest value ItemEvent.SELECTED, indicating why the event was dispatched. The third argument should identify which of the choices in the source caused the event to be dispatched. In this example the String integer representation of the button's location, obtained from the ActionEvent *event*'s getActionCommand() method, and converted to an instance of the Integer class. The final argument, ItemEvent.ITEM_STATE_CHANGED, is an additional qualification of why the ItemEvent was dispatched.

So, using the example shown in Figure 11.1, if the user presses the 50p button this will cause an ActionEvent, with the actionCommand String "3", to be passed to this *actionPerformed()* method. This information will be propagated from this method in the item attribute of the ItemEvent as an Integer object with the value 3. The only remaining action in the *CoinHopperUI* class is *getSelectedObjects()*, which is mandated by the



```
5   5 coins.
5   10 coins.
5   20 coins.
6   50 coins.
5   100 coins.
0   200 coins.

0 in the bucket.

Total 975.
```

Figure 11.8 *CoinHopperUIDemonstration*, visual appearance.

ItemSelectable interface but does not have a sensible implementation in this situation. Accordingly its implementation is as follows.

```
  public Object[] getSelectedObjects(){
     return null;
  } // End getSelectedObjects.
} // End CoinHopperUI.
```

A demonstration client for this class is supplied in the *CoinHopperUIDemonstration* class, whose appearance is illustrated in Figure 11.8. The left-hand image shows the panel containing the six UK coins supported with the toolTipText for the 50p coin displayed. The right-hand image shows the output produced on the terminal window when the 50p button is pressed immediately after the interface has become visible. The output is obtained from a call of the *CoinHopper toString()* action and shows that there are now six 50p coins in the hopper. The implementation of the *CoinHopperUIDemonstration* class as far as the start of the *init()* method is as follows.

```
package hoppers;

import javax.swing.*;

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

public class CoinHopperUIDemo
                             extends JApplet
                          implements ItemListener {

private final static ImageIcon ukIcons[] =
   { new ImageIcon( "hoppers/fivep.gif",     "Five P"),
     new ImageIcon( "hoppers/tenp.gif",      "Ten P"),
     new ImageIcon( "hoppers/twentyp.gif",   "Twenty P"),
     new ImageIcon( "hoppers/fiftyp.gif",    "Fifty P"),
     new ImageIcon( "hoppers/onepound.gif",  "One £"),
     new ImageIcon( "hoppers/twopounds.gif", "Two £") };

private final static String ukNames[] =
0030                        { "Five P",  "Ten P", "Twenty P",
0031                          "Fifty P", "One £", "Two £"};
0032
0033  private final static int ukValues[] =
0034                              { 5, 10, 20, 50, 100, 200};
0035
0036  CoinHopperUI demoInterface = null;
0037  CoinHopper   demoHopper    = null;
```

The class declaration, on lines 0017 to 0019, indicates that the *CoinHopperUIDemonstration* class extends the JApplet class, in order to provide an environment to mount the artifact within, and implements the ItemListener interface in order to be able to listen to the ItemEvents that will be generated by an instance of the *CoinHopperUI* class.

Lines 0021 construct an array of ImageIcons, called *ukIcons*, containing the images representing the six coins visible on the artifact. The `gif` image files are all stored in the *hoppers* directory and each constructor also specifies a name for the coin should the image not be available. Lines 0029 to 0031 declare an array of Strings, called *ukNames*, corresponding to the six coins, which will be used to supply toolTipText resources for each of the JButtons on the artifact. The final class-wide resource, declared on lines 0033 and 0034, is an array of **int**, called *ukValues*, with values corresponding to the six coins and which will be passed to a *CoinHopper* constructor.

The declarations conclude, on lines 0036 and 0037, with two instance attributes: an instance of the *CoinHopperUI* class called *demoInterface* and an instance of the *CoinHopper* class called *demoHopper*. The class continues with the *init()* method, as follows.

```
0041     public void init() {
0042        theHopper     = new CoinHopper( ukValues);
0043        demoInterface = new CoinHopperUI( ukIcons,
0044                                                      ukNames);
0045        demoInterface.addItemListener( this);
0046        this.getContentPane().add( demoInterface);
0047     } // End init.
```

Line 0042 constructs an instance of the *CoinHopper* class, referenced by *demoHopper*, and initialized with the *ukValues* array. Lines 0043 and 0044 construct an instance of the *CoinHopperUI* class, referenced by *demoInterface*, and initialized with the *ukIcons* and *ukNames* arrays. The ItemListener resource of the *demoInterface* is then established as **this** instance of the *CoinHopperUIDemo* class being initialized, before the *demoInterface* is added to the applet's *contentPane()* on line 0046. The consequence of this initialization is that the artifact will become visible to the user, as shown in Figure 11.8, with the ItemEvents produced when the user presses one of the buttons passed to the *itemStateChanged()* method, implemented as follows.

```
0050     public void itemStateChanged( ItemEvent event) {
0051
0052     int theCoin = ((Integer) event.getItem()).intValue();
0053
0054         demoHopper.depositCoin( theCoin);
0055         System.out.println( demoHopper);
0056     } // End itemStateChanged.
```

The item attribute of the ItemEvent generated from the *demoInterface* contains the ordinal position of the coin that the user has chosen, as an instance of the Integer class. Line 0052 extracts this information from the event, the getItem() method returns an instance of the Object class which is cast to Integer, and then its intValue() method is used to obtain the value as an **int**, which is assigned to *theCoin*. Having done so, line 0054 deposits *theCoin* into the *demoHopper* by calling its *depositCoin()* method and the state of the hopper is then output to the terminal by means of line 0055.

This demonstration artifact can be used to provide initial confidence that the *CoinHopperUI* appears to be working and illustrates the technique by which it can be connected to a *CoinHopper*. A more complete demonstration can be obtained when these two classes are used as part of the *ColaMachine*.

11.4 The *ColaMachine* design

The state transition diagram describing the operation of the *ColaMachine* is given in Figure 11.9. The interface is first presented to the user in the *reset state*, where only the *coin entry* buttons are sensitive and the *dyna message* display is inviting the user to enter a coin, possibly warning that the exact change only must be used. This state is shown in the left-hand illustration in Figure 11.1.

From the *reset state* the user can only enter a coin, and if the value of the coin is greater than, or equal to, the cost of a can of drink the interface will transit into the *dispense state*. In this state the *coin entry* buttons are desensitized, the *refund* and *drink dispense* buttons are sensitized and the *dyna message* display is inviting the user to chose a drink.

Alternatively, if the value of the first coin entered in the *reset state* is less than the cost of a can of drink then the interface transits into the *refund state*. In this state only the *coin entry* and *refund* buttons are sensitive and the *dyna message* display will show the total value of the coins entered. If the user continues to enter coins and the total value of the coins is still less than the cost of a drink then the transition which leads back to the *reset state* is followed, causing the value displayed on the *dyna message* display to increment. When the user eventually enters sufficient coins the transition from the *refund state* to the *dispense state* will be followed.

From either the *refund state* or the *dispense state* it is possible for the user to press the *refund button*. To indicate this possibility the two states have been enclosed together in a round-edged rectangle from which the *refund pressed* transition leads to the *refund dialog* state. This dialog will simulate the return of coins equivalent to the value which has been entered and contains a button labeled *OK* which, when pressed, unposts the *refund dialog* and transits back to the *reset state*.

From the *dispense state* it is possible for the user to request a drink by pressing one of the drink *dispense buttons*. If it is not possible for the machine to provide the change required then the interface will transit to the *nochange dialog* state. The user will have been warned of this possibility, via the *dyna message* in the *reset state*, but it must be assumed that on occasion some users will not attend to it. The *nochange dialog* will explain the situation to the user and refund the value of the coins entered and, when the dialog's *OK* button is pressed, will transit back to the *reset state*.

If it is possible for change to be made when the user presses a drink *dispense button* the interface will transit into the *dispense dialog* state. Here the can of drink that the user has requested together with any change due is shown, as illustrated in the right-hand image in Figure 11.1. When the user presses the *OK* button on this dialog, if at least one of the drink hoppers has cans remaining the interface will transit to the *reset state*. When the

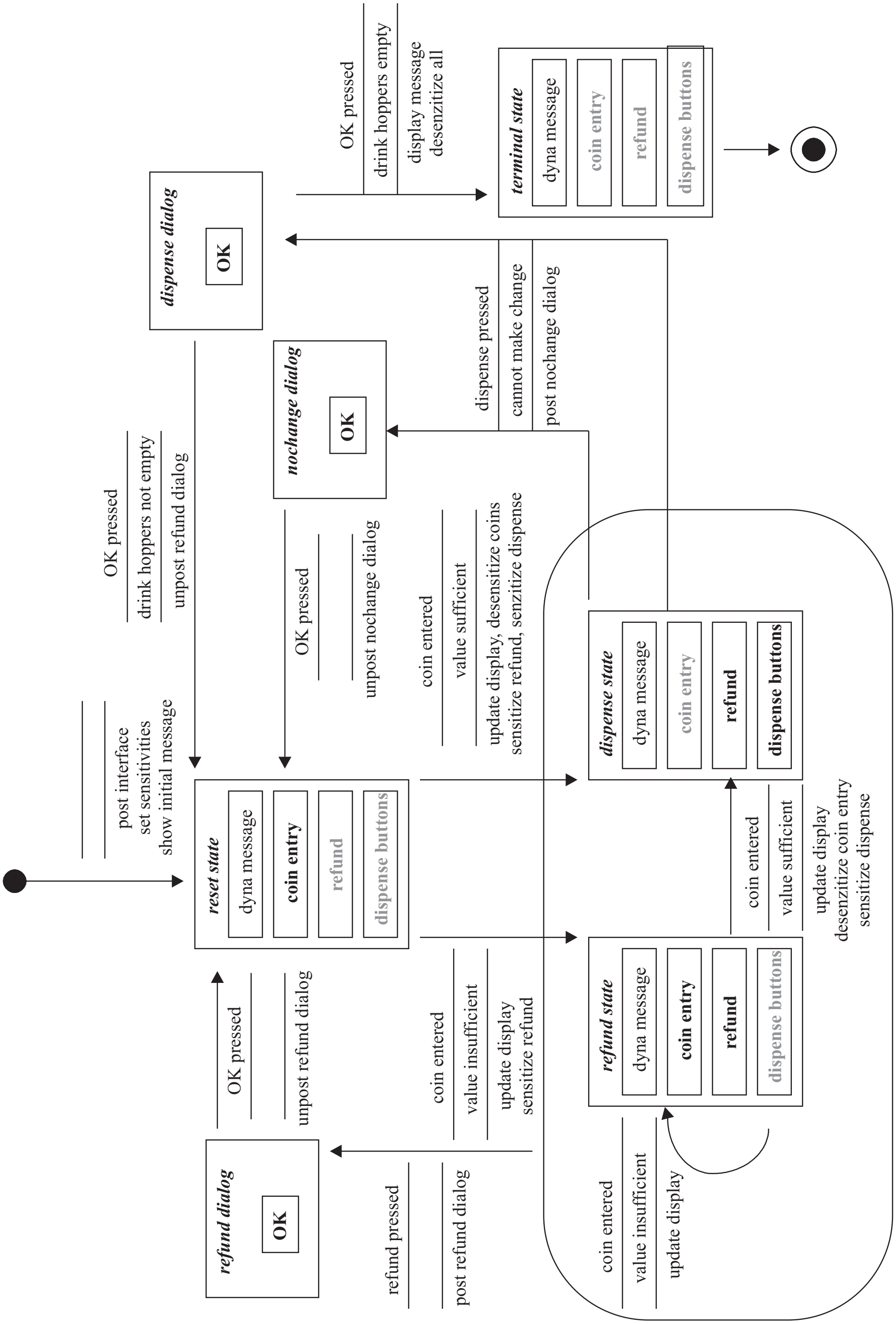
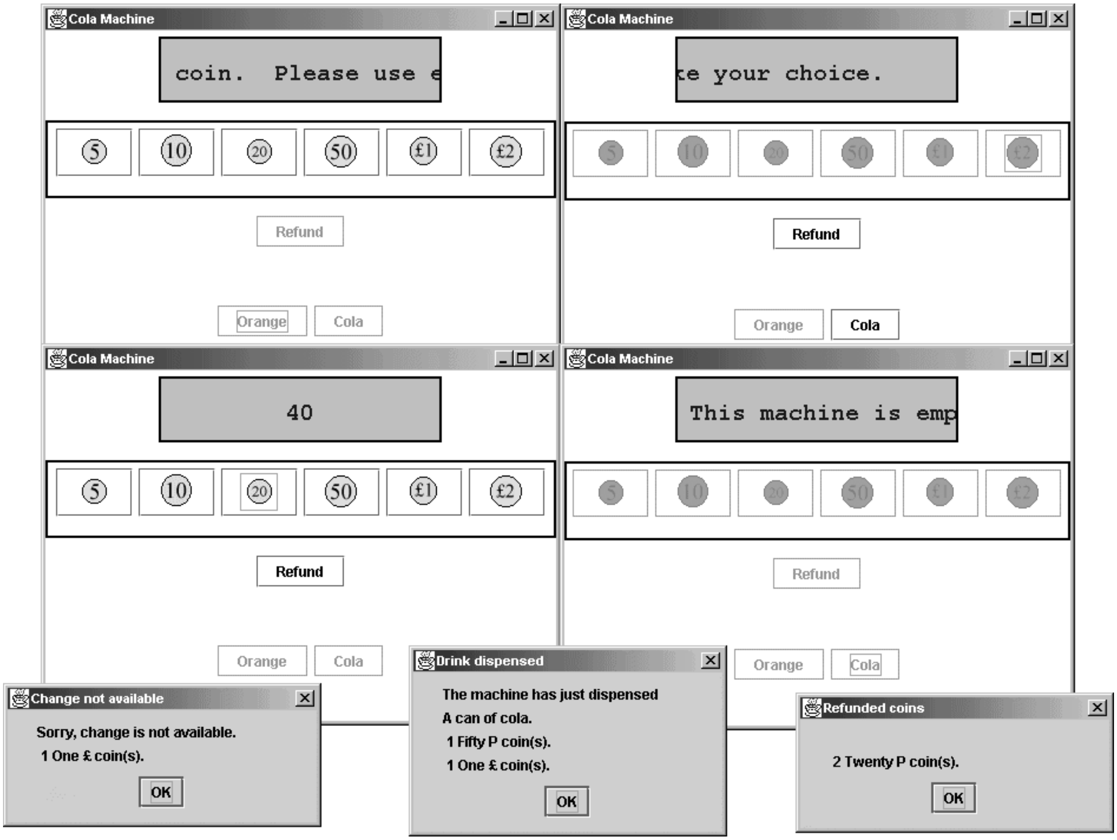


Figure 11.9 *ColaMachine*: state transition diagram.



Sorry, change is not available.
1 One £ coin(s).

The machine has just dispensed
A can of cola.
1 Fifty P coin(s).
1 One £ coin(s).

2 Twenty P coin(s).

Figure 11.10 *ColaMachine*: various appearances.

dispense state is subsequently shown to the user only those *dispense buttons* corresponding to non-empty hoppers will be sensitive, although this is not explicit in the design.

At some stage all drink hoppers will be empty and this will cause the transition to the *terminal state* to be taken when the *OK* button on the *dispense dialog* is pressed. In this state none of the controls is sensitive and the *dyna message* display explains that the machine is empty. This is effectively the termination of all possible interaction with the artifact and a transition to the terminal state is shown associated with it.

Figure 11.10 illustrates the appearances of the various states and dialogs with the interface configured for UK use. The image at the top left illustrates the *reset state* with the message asking the user to use the exact change displayed. The image on the bottom left is of the *reset state*, with 40p entered and only the reset button sensitive. The upper right image shows the *dispense state* with the *orange* can hopper empty and the image in the lower right shows the *terminal state*. The lower row of images shows the *nochange dialog*, the *dispense dialog* and the *refund dialog*.

All three of the transient dialogs shown on this STD will only operate effectively if they prevent the user from interacting with the underlying interface while they are active; that is they must be *modal* in their operation. The alternative possibility is for the dialogs to be posted but for the user still to be able to interact with the underlying interface; these are known as *non-modal* dialogs. In general, non-modal dialogs should be favored unless, as in this example, modal dialogs are essential to the operation of the artifact.

A context diagram for the *ColaMachine* artifact has already been supplied in Figure 11.2 and described in section 11.1. A layout management diagram for the interface is given in Figure 11.11. It shows that the JApplet *ColaMachine* has a four row by one column GridLayout policy. The upper cell is occupied by a JPanel, called *messagePanel* upon which a *DynaLabel* instance called *dynaMessage* is mounted and centered by the default FlowLayout policy. The two lower cells are also occupied by JPanels, called *refundPanel* and *drinkPanel*, whose default FlowLayout is used to position the *refundButton* and the two drink buttons, *orangeButton* and *colaButton*. The remaining cell, the second from the top, is occupied by an instance of the *CoinHopperUI* class called *coinEntry*.

The Grid layout manager will force all four cells to be the same size. The height of each cell will be determined by the height of the highest cell and this will, most probably, be the height required by the *coinEntry* component. The width of the interface shown in Figure 11.1 is determined by the width of the *coinEntry* component, but with a different set of coins, used in a different country, it might be determined by the width of the *dynaMessage*.

11.5 The *ColaMachine*: first implementation

This first implementation of the *ColaMachine* class will not explicitly implement internationalization considerations. However, some of the implementation decisions, most noticeably the declaration of all messages on the interface as manifest constants, have been taken with internationalization in mind. To this end these resources, which will change when the artifact is re-engineered, have been collected together at the end of the listing. In the description which follows their declarations will be introduced as and when they are referenced in the main part of the implementation.

The implementation of the *ColaMachine* class, as far as the start of its *init()* method, is as follows.

```
package colamachine;

import javax.swing.*;
import javax.swing.border.*;

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

import Hoppers.*;
import DynaLabel.*;

public class ColaMachine extends    JApplet
                         implements ItemListener,
                                    ActionListener {

private static final int INITIAL_STATE  = 0;
private static final int RESET_STATE    = 1;
private static final int REFUND_STATE   = 2;
private static final int DISPENSE_STATE = 3;
private static final int TERMINAL_STATE = 4;
private              int theState = INITIAL_STATE;
```

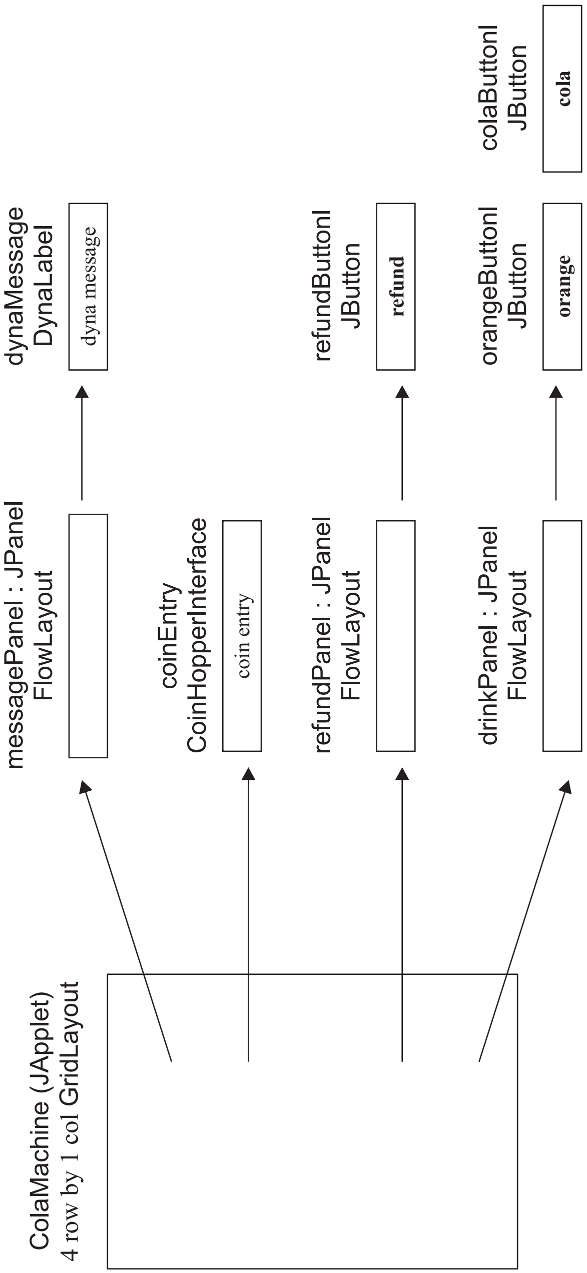


Figure 11.11 *ColaMachine*: layout management.

```
0031
0032  CoinHopperUI coinEntry    = null;
0033  CoinHopper   coinHopper   = null;
0034  int          enteredValue = 0;
0035
0036  DynaLabel  dynaMessage  = null;
0037
0038  JButton    refundButton = null;
0039
0040  CanHopper  orangeHopper = null;
0041  JButton    orangeButton = null;
0042  CanHopper  colaHopper   = null;
0043  JButton    colaButton   = null;
```

This class is contained within the *colamachine* package of classes, as declared on line 0010. Lines 0012 to 0019 supply the required importations and the class is declared on lines 0021 to 0023. The *ColaMachine* class extends the JApplet class in order to be able to mount the interface, implements the ItemListener interface in order to listen to the events dispatched from the *CoinHopperUI* and also implements the ActionListener interface in order to be able to listen to the events dispatched by the JButton instances.

Lines 0025 to 0029 declare the private class-wide manifest values which model the state of the interface, as given in Figure 11.9. Only the major states of the interface and not the transient dialog states need to be modeled, for reasons which will be explained shortly. Line 0030 declares an instance variable called *theState* which will always indicate the current state of the interface and is initialized to the value *INITIAL_STATE*.

The declarations continue, on lines 0032 to 0034, with the objects required for dealing with the entry and storage of coins. These are an instance of the *CoinHopperUI* class called *coinEntry*, a *CoinHopper* called *coinHopper* to store the coins and a primitive **int** called *enteredValue*, which will be used to maintain knowledge of the value of the coins which have been entered by the current user.

An instance of the *DynaLabel* class, called *dynaMessage*, is declared on line 0036 and will be used to display messages to the user at the top of the interface. Line 0038 declares the *refundButton* as an instance of the JButton class.

Lines 0040 to 0043 declare the objects needed to model the cans held by the machine. These are two *CanHopper*s, called *orangeHopper* and *colaHopper*, and two *JButton*s called *orangeButton* and *colaButton*. The *init()* method follows these declarations and commences as follows.

```
0047     public void init() {
0048
0049     JPanel messagePanel = null;
0050     JPanel refundPanel  = null;
0051     JPanel drinkPanel   = null;
0052
0053        this.getContentPane().setLayout(
0054                              new GridLayout( 4, 1, 0, 10));
0055        this.getContentPane().setBackground( Color.white);
0056
0057        dynaMessage = new DynaLabel();
0058        dynaMessage.setBackground( Color.lightGray);
0059        dynaMessage.setForeground( Color.blue);
```

```
0060        dynaMessage.setFont( new
0061                         Font( "Courier", Font.BOLD, 20));
0062        dynaMessage.setBorder( new
0063                             LineBorder( Color.black, 2));
0064
0065        messagePanel = new JPanel();
0066        messagePanel.setOpaque( false);
0067        messagePanel.add( dynaMessage);
0068        this.getContentPane().add( messagePanel);
```

The three JPanel instances, declared on lines 0049 to 0052, are declared as local variables of the *init()* method as their identity is only required during initialization. Lines 0053 to 0055 then configure the *ColaMachine* JApplet panel by establishing a four row by one column GridLayout for its contentPane and also setting its background color to white.

The next section of the *init()* method, from lines 0057 to 0068, constructs, configures and mounts the *dynaMessage* into the top of the interface. The *dynaMessage* is constructed on line 0057 using its default constructor. This would cause it to display a default message announcing that it is a *DynaLabel* instance, but, as will be explained, its *setMessage()* method will be called to change this before the user sees it. Lines 0058 to 0063 configure its *foreground*, *background* and *font*, as well as installing a border into it. This part of the *init()* method concludes by constructing and configuring the *messagePanel* JLabel instance, adding the *dynaMessage* to it and adding it to the *ColaMachine* in the upper cell of its GridLayout. The panel is configured, on line 0066, to be non-opaque so that the parts of the panel not occupied by the *dynaMessage* will allow the underlying background to show through.

The next part of the *init()* method is concerned with preparing the *coinEntry* and *coinHopper* parts of the artifact, as follows.

```
0070        coinEntry = new CoinHopperUI( coinIcons,
0071                                                coinNames);
0072        coinEntry.addItemListener( this);
0073        coinEntry.setBackground( Color.white);
0074        this.getContentPane().add( coinEntry);
0075        coinHopper = new CoinHopper( coinValues);
```

The *coinEntry* instance is constructed on lines 0070 and 0071 as an instance of the *CoinHopperUI* class, the arguments to the constructor being part of the resources declared at the bottom of the file, as follows.

```
0287  private final static ImageIcon coinIcons[] =
0288     { new ImageIcon( "hoppers/fivep.gif",      "Five P"),
0289       new ImageIcon( "hoppers/tenp.gif",        "Ten P"),
0290       new ImageIcon( "hoppers/twentyp.gif",  "Twenty P"),
0291       new ImageIcon( "hoppers/fiftyp.gif",    "Fifth P"),
0292       new ImageIcon( "hoppers/onepound.gif",    "One £"),
0293       new ImageIcon( "hoppers/twopounds.gif", "Two £") };
0294
0295  private final static int coinValues[] =
0296                            { 5, 10, 20, 50, 100, 200};
0297
0298  private final static String coinNames[] =
0299                          { "Five P",  "Ten P", "Twenty P",
0300                              "Fifty P", "One £", "Two £"};
```

These declarations are identical, apart from the identifiers, to those used in the *CoinHopperUIDemo* class, as described earlier. The *coinIcons* array will be used to supply the images used on the interface and the *coinNames* will supply the *toolTipText* resources for this part of the interface. The *init()* method continues, on lines 0073 and 0074, by registering **this** *ColaMachine* instance being initialized as its listener and then adding it into the second cell of the applet. This part of the *init()* method concludes by constructing the *coinHopper* and passing the array *coinValues* as an argument to its constructor. The *init()* method continues with the construction of the refund button as follows.

```
0077         refundButton = new JButton( REFUND_LABEL);
0078         refundButton.setBackground( Color.white);
0079         refundButton.setForeground( Color.black);
0080         refundButton.addActionListener( this);
0081         refundButton.setActionCommand( "refund");
0082         refundPanel  = new JPanel();
0083         refundPanel.setOpaque( false);
0084         refundPanel.add( refundButton);
0085         this.getContentPane().add( refundPanel);
```

The *refundButton* is constructed on line 0077 with the label which it will display specified by the manifest *REFUND_LABEL* string, whose declaration will be given shortly. The visual appearance of the button is then configured and **this** instance registered as its *actionListener* before it *actionCommand* is specified on line 0081. The *actionCommand* resource is used only within the Java source code, in the *actionPerformed()* method to decide which button was pressed, and so will not change between linguistic environments; consequently it is specified as the English word "*refund*". Having constructed and configured the *refundButton*, its panel, *refundPanel*, is constructed and configured before the button is added to the panel and, finally, on line 0085, the panel is added to the third cell on the interface. The declaration of the *REFUND_LABEL*, and the labels for the two remaining buttons, is as follows.

```
0320  private static final String REFUND_LABEL = "Refund";
0321  private static final String ORANGE_LABEL = "Orange";
0322  private static final String COLA_LABEL   = "Cola";
```

The next part of the *init()* method is somewhat similar, constructing and configuring two JButton instances and adding them to the *drinkPanel*, which is then added to the last location on the interface. It is implemented as follows.

```
0087         orangeHopper = new CanHopper( ORANGE_LABEL);
0088         orangeButton = new JButton( ORANGE_LABEL);
0089         orangeButton.setBackground( Color.white);
0090         orangeButton.setForeground( Color.black);
0091         orangeButton.setActionCommand( "orange");
0092         orangeButton.addActionListener( this);
0093
0094         colaHopper = new CanHopper( COLA_LABEL);
0095         colaButton = new JButton( COLA_LABEL);
0096         colaButton.setBackground( Color.white);
0097         colaButton.setForeground( Color.black);
0098         colaButton.setActionCommand( "cola");
```

```
0099        colaButton.addActionListener( this);
0100        colaButton.setEnabled( false);
0101
0102        drinkPanel = new JPanel();
0103        drinkPanel.setOpaque( false);
0104        drinkPanel.add( orangeButton);
0105        drinkPanel.add( colaButton);
0106        this.getContentPane().add( drinkPanel);
0107
0108        this.setResetState();
0109     } // End init.
```

Lines 0088 to 0092 construct and configure the *orangeButton*, an instance of the JButton class. Before this, on line 0087, the *orangeHopper canHopper* is constructed using the constructor which requires a string to describe the nature of the cans being dispensed and which will stock the hopper with the default (25) number of cans. Lines 0094 to 0100 do the same for the *colaHopper* and the *colaButton* before both buttons are added to the newly constructed *drinkPanel*, after which it is added to the interface on line 0106.

The final step in the *init()* method, on line 0108, is to call the *ColaMachine setResetState()* method to effect the transition from the initial to the reset state, as shown on the STD in Figure 11.9. This method is implemented as follows.

```
0112     private void setResetState(){
0113        if ( coinHopper.canGiveChange(
0114                       coinValues[ coinValues.length -1] -
0115                                              CAN_COST) ){
0116          dynaMessage.setMessage( RESET_MESSAGE);
0117        } else {
0118          dynaMessage.setMessage( RESET_MESSAGE +
0119                                  CORRECT_CHANGE_MESSAGE);
0120        } // End if.
0121        dynaMessage.setShowStyle( DynaLabel.CONTINUAL);
0122        coinEntry.setEnabled(    true);
0123        refundButton.setEnabled( false);
0124        orangeButton.setEnabled( false);
0125        colaButton.setEnabled(   false);
0126        enteredValue = 0;
0127        theState = RESET_STATE;
0128     } // End setResetState.
```

The various manifest values referred to in this implementation are declared as follows.

```
0303  private static final int CAN_COST = 50;
0304
0305  private static final String RESET_MESSAGE =
0306                       " Press a button to enter a coin. ";
0307  private static final String CORRECT_CHANGE_MESSAGE =
0308                         " Please use exact change only. ";
```

The first step of the *setResetState()* method is to configure the *dynaMessage* to inform the user that the machine is ready for use. This will either be the message "*Press a button to enter a coin.*" or the compound message "*Press a button to enter a coin. Please use exact change only.*" if it cannot be guaranteed that the correct change can always be given. This decision is taken on the basis of the worst-case scenario, which is a request to make change from the highest denomination coin. The Java phrase on line 0114 will resolve to the value of the highest value coin with the cost of a single can subtracted from it on line 0115. The resulting value is passed to the *coinHopper canGiveChange()* method which will return **true** or **false** and so determine which branch of the **if** structure is taken.

Whichever phrase is installed into the *dynaMessage* it is *CONTINUAL*ly animated on line 0121, as indicated in Figure 11.1. Lines 0122 to 0125 then establish the required pattern of sensitivities for the interactive components before the *enteredValue* is set to zero and finally, on line 127, *theState* is updated to *RESET_STATE*. In this state only the *coinEntry* component is active and will dispatch an ItemEvent to the *ColaMachine*'s *itemStateChanged()* method, as follows, when the user simulates the entry of a coin.

```
0158     public void itemStateChanged( ItemEvent event) {
0159
0160     int theCoin = ((Integer) event.getItem()).intValue();
0161
0162         coinHopper.depositCoin( theCoin);
0163         enteredValue += coinValues[ theCoin];
0164         dynaMessage.setMessage(
0165                  new Integer( enteredValue).toString());
0166         dynaMessage.setShowStyle( DynaLabel.STATIONARY);
0167
0168         if ( enteredValue >= CAN_COST ) {
0169            this.setDispenseState();
0170         } else if ( theState == INITIAL_STATE) {
0171            this.setRefundState();
0172         } // End if.
0173     } // End itemStateChanged.
```

The first part of this method is comparable to the *itemStateChanged()* method in the *CoinHopperUIDemo* class as previously explained. The ordinal location of the coin selected by the user is extracted into *theCoin*, on line 0160, and *theCoin* is then deposited into the *coinHopper* on line 0162. The value of *theCoin*, obtained from the *coinValues* array, is then added to the *enteredValue*, on line 0163, and the updated value is displayed, in a stationary manner, on the *dynaMessage* on lines 0164 to 0166.

The deposit of a coin may cause a transition to the *dispense state* or to the *refund state*, depending upon the updated *enteredValue*. This decision is made in the **if** structure on lines 0168 to 0172 which will call the appropriate state setting method. The third possible *coin entry* transition, which leads from the *refund state* back to the *refund state*, need not be explicitly effected in this method. The implementation of these two state setting methods is as follows.

```
0130     private void setRefundState(){
0131        refundButton.setEnabled( true);
0132        theState = REFUND_STATE;
0133     } // End setRefundState.
0134
0135     private void setDispenseState(){
0136        dynaMessage.setMessage(
0137                (new Integer( enteredValue).toString()) +
0138                                    DISPENSE_PROMPT);
0139         dynaMessage.setShowStyle( DynaLabel.CONTINUAL);
0140         coinEntry.setEnabled(    false);
0141         refundButton.setEnabled( true);
0142         if ( ! orangeHopper.isEmpty()) {
0143            orangeButton.setEnabled( true);
0144         } // End if.
0145         if ( ! colaHopper.isEmpty()) {
0146            colaButton.setEnabled(   true);
0147         } // End if.
0148         theState = DISPENSE_STATE;
0149      } // End setDispenseState.
```

The transition to the *refund state* need only set the sensitivity of the *refundButton* and this is effected on line 0131. The transition to the *dispense state* is more complex and commences, on lines 0137 to 0139, by installing a continually animated message of the form "*50 entered, please make your choice*", as can be partly seen in Figure 11.10. The *enteredValue* attribute provides the first part of this message, configured as a String on line 0137 and is followed by the manifest String *DISPENSE_PROMPT,* declared as follows.

```
0336 private static final String DISPENSE_PROMPT =
0337                          " entered, please make your choice.";
```

The *setDispenseState()* continues, on lines 0140 and 0141, by desensitizing the *coinEntry* component and sensitizing the *refundButton*. Lines 0142 to 0147 then set the sensitivities of the *orangeButton* and *colaButton*, only if their respective *CanHopper*s are not empty.

In these two states the three JButton instances are sensitive, and when any one of them is pressed an ActionEvent will be dispatched to the *actionPerformed()* method, implemented as follows.

```
0175      public void actionPerformed( ActionEvent event) {
0176
0177         if ( event.getActionCommand() == "refund") {
0178            this.refundCoins();
0179         } else {
0180            this.dispenseCan( event.getActionCommand());
0181         } // End if.
0182      } // End actionPerformed.
```

The first branch of the **if** structure will call the private *refundCoins()* method, implemented as follows, if the *event*'s actionCommand attribute indicates that the *refundButton* was pressed.

```
0186      private void refundCoins() {
0187
0188      String refundedCoins = getRefundString( enteredValue);
0189
0190         JOptionPane.showMessageDialog( this,
0191                                        refundedCoins,
0192                                        REFUND_DIALOG_TITLE,
0193                                    JOptionPane.PLAIN_MESSAGE,
0194                                        null);
0195         this.setResetState();
0196      } // End refundCoins.
```

The first step in this method is to initialize *refundedCoins* to the *String* returned from the *getRefundString()* method when it is passed *enteredValue* as an argument. The implementation of this method is as follows.

```
0249     private String getRefundString( int forThisAmount) {
0250
0251     int[]  coinsToReturn = null;
0252     String refundString  = "";
0253
0254        try {
0255           coinsToReturn = coinHopper.giveChange(
0256                                          forThisAmount);
0257        } catch ( HopperException exception) {
0258           // Do Nothing!
0259        } // End try catch.
0260        for ( int thisCoin = 0;
0261                  thisCoin < coinsToReturn.length;
0262                  thisCoin++) {
0263           if ( coinsToReturn[ thisCoin] > 0 ) {
0264              refundString = refundString.concat( "\n " +
0265                                 coinsToReturn[ thisCoin] +
0266                                                     " " +
0267                                     coinNames[ thisCoin] +
0268                                          COINS_LITERAL);
0269           } // End if.
0270        } // End for.
0271        return refundString;
0272     } // End getRefundString.
```

The purpose of this method is to obtain the coins which will make up the value of the argument *forThisAmount* from the *coinHopper* and then express these as a String which can be shown to the user. The first step is to call the *coinHopper giveChange()* method passing onward the argument *forThisAmount*. This method, explained in Section 11.3, will return an array of **int** whose length is equal to the number of coin stacks in the *coinHopper*, with the value in each element indicating the number of coins to return. This method may throw a *HopperException* which is caught but not attended to. The **int** array returned from *giveChange()* is stored in the local variable *coinsToReturn*.

The second part of the *getRefundString()* method expresses the contents of the *coinsToReturn* array as a String. It does this by iterating through the entire extent of the array and, if the element currently being considered is not zero, catenates a phrase onto the end of the *refundString*. The phrase commences, on line 0265, with the number of coins and is followed, on line 0267, by the name of the coin, obtained from the *coinNames* array. The final part of the phrase, on line 0268, is obtained from the *COINS_LITERAL*, declared as follows.

```
0329  private static final String COINS_LITERAL = " coin(s).";
0330
0331  private static final String REFUND_DIALOG_TITLE =
0332                                             "Refunded coins";
0333  private static final String NOCHANGE_DIALOG_TITLE =
0334                                       "Change not available";
0335  private static final String DISPENSE_DIALOG_TITLE =
0336                                              "Drink dispensed";
```

An example of a String which can be produced by this method is visible in the illustrations at the bottom of Figure 11.11. In the middle illustration the method was called to provide £1.50p in change and accomplished this by supplying a £1 coin and a 50p coin.

Within the *refundCoins()* method the String returned from the call of the *getRefundString()* method is referenced by the local variable *refundedCoins* and used as an argument to the JOptionPane showMessageDialog() call, on line 0191. The JOptionPane class provides a large number of utility methods that allow a modal dialog to be automatically placed on the desktop, centered within the parent window, using a single method call. In this example a message dialog is being used as it contains a single *OK* button which, when pressed, dismisses the dialog from the desktop.

A call of the JOptionPane showMessageDialog() method requires five arguments which are, in sequence: the parent window to place the dialog over, in this example **this** applet's window; the String to be displayed in the message area of the dialog (the *refundedCoins* string obtained from *getRefundString()*); a title for the dialog's frame (in this example the manifest *REFUND_DIALOG_TITLE* whose declaration has already been given); a JOptionPane manifest value which informs the dialog which standard icon to display to the left of the message (in this example the manifest value PLAIN_MESSAGE indicates that none of the standard icons should be displayed); and a final argument which can supply a specialized icon to be displayed (in this example the value **null** again indicates that no icon should be shown).

The effect of the showMessageDialog() call within the *refundCoins()* method is to post a dialog such as that shown in the bottom right of Figure 11.10 onto the desktop. As the dialog is modal it will remain posted, preventing any further interaction with the artifact until it has been attended to and dismissed. Flow of control will remain suspended at the point where the dialog was posted from and will continue with the next instruction when it is dismissed. In this example, on line 0195, this is a call of the *setResetState()* method, which effects the transition from the *refund dialog* state to the *reset state*, as shown on the STD in Figure 11.9.

All the dialog states in the *ColaMachine* are implemented by means of *JOptionPane* dialogs. This implies that they are all transient, with flow of control suspended until they are dismissed, and this is why they do not have to be modeled as manifest states in the state modeling aspects of the *ColaMachine* implementation. The techniques for posting non-modal dialogs are more complex and have already been introduced in previous chapters.

If the user presses one of the drink dispense buttons, rather than the refund button, then, as shown on line 0180 of the *actionPerformed()* method given above, the *dispenseCan()* method will be called. The first part of the implementation of this method is as follows.

```
0203     private void dispenseCan( String drinkRequested) {
0204
0205     String drinkCanMessage = null;
0206     String refundedCoins   = null;
0207
0208        if ( ! coinHopper.canGiveChange(
0209                         enteredValue - CAN_COST)) {
0210           refundedCoins = getRefundString( enteredValue);
```

```
0211           JOptionPane.showMessageDialog( this,
0212                      NO_CHANGE_MESSAGE + refundedCoins,
0213                              NOCHANGE_DIALOG_TITLE,
0214                              JOptionPane.PLAIN_MESSAGE,
0215                              null);
0216           this.setResetState();
```

The argument to the method call will be a String, obtained from the actionCommand of the ActionEvent dispatched from the JButton, containing the English language identity of the *drinkRequested*. The first part of this method determines whether change can be obtained from the *coinHopper* and posts the *nochange dialog*, as shown on the STD in Figure 11.9, if change cannot be made.

The **if** decision on lines 0208 is a call of the *coinHopper canGiveChange()* method, passing as an argument the *enteredValue* minus the *CAN_COST*. So, for example, if the user has simulated the entry of a £2 coin and requested a drink the *coinHopper* will be asked whether it can provide £1.50 (200–50) in change. If change cannot be provided then line 0210 obtains a String representation of the *enteredValue*, and this is passed as an argument to the JOptionPane showMessageDialog() method call on lines 0211 to 0215. The effect of this call is to post the *nochange dialog*, as illustrated in the bottom left of Figure 11.10, and block until it is dismissed, whereupon line 0216 will effect the transition to the *reset state*, as required by the STD in Figure 11.9.

The second part of the *dispenseCan()* method will only be called if it is possible for the machine to provide the change required, and is implemented as follows.

```
0217        } else {
0218           if ( drinkRequested.equals( "orange")) {
0219              try {
0220                 orangeHopper.dispense();
0221              } catch (HopperException exception ){
0222                 // Do Nothing!
0223              } // End if.
0224              drinkCanMessage = ORANGE_MESSAGE;
0225           } else {
0226              try {
0227                 colaHopper.dispense();
0228              } catch (HopperException exception ){
0229                 // Do Nothing!
0230              } // End if.
0231              drinkCanMessage = COLA_MESSAGE;
0232           } // End if.
0233
0234           refundedCoins = getRefundString(
0235                              enteredValue - CAN_COST);
0236
0237           JOptionPane.showMessageDialog( this,
0238                        drinkCanMessage + refundedCoins,
0239                                  DISPENSE_DIALOG_TITLE,
0240                              JOptionPane.PLAIN_MESSAGE,
0241                                          null);
0242           if ( orangeHopper.isEmpty() &&
0243                colaHopper.isEmpty()   ){
```

```
0244              this.setTerminalState();
0245           } else {
0246              this.setResetState();
0247           } // End if.
0248        } // End if.
0249     } // End dispenseCan.
```

The first step in this part of the method is to decide whether the user requested a can of *orange* or a can of *cola* and to attempt to dispense a can from the appropriate hopper. Thus line 0218 determines that the user requested a can of *orange* and leads to a call of the *orangeHopper dispense()* method on line 0220. This call may throw a HopperException, which cannot be ignored but is not attended to, on line 0222, in the **catch** part of the **try/catch** structure. The call of *dispense()* will remove one can from the hopper and may leave it empty; if it is not already empty no exception will be thrown and the local variable *drinkCanMessage* is given the manifest value *ORANGE_MESSAGE*, whose declaration is as follows.

```
0318  private static final String ORANGE_MESSAGE =
0319                          "The machine has just dispensed \n" +
0320                          "A can of orange.";
0321  private static final String COLA_MESSAGE =
0322                          "The machine has just dispensed \n" +
0323                          "A can of cola.";
```

Lines 0226 to 0232 are essentially identical and dispense a can from the *colaHopper*, placing the COLA_MESSAGE into *drinkCanMessage*. The method continues with lines 0234 and 0235 obtaining a String representation of the change due to the user. If the *enteredValue* is equal to *CAN_COST* then this will be a call of *getRefundString()* with the argument 0, which will result in an empty String being returned.

The next step in the *dispenseCan()* method, on lines 0237 to 0241, is to post the *dispense dialog* to the desktop, again using a blocking call of JOptionPane showMessageDialog() method. The second argument to this call, on line 0238, is the *drinkCanMessage* and the *refundMessage* which, if the *refundMessage* is not empty, appears to the user as shown in the lower middle illustration in Figure 11.10. If the user has provided the exact cost of a can then the *refundMessage* will be empty and only the *drinkCanMessage* will appear on the dialog.

The final step in the *dispenseCan()* method is to effect the transition back to the *reset state* or to the *terminal state*, depending upon the state of the drink hoppers. The **if** condition on lines 0242 and 0243 will be **true** if both drink hoppers are empty and the consequence is to call the *setTerminalState()* method. Otherwise at least one of the drink hoppers must still have cans available, and line 0246 will transit to the *reset state*. When the transition to the dispense state is subsequently effected any empty drink hoppers will cause their corresponding drink button to be disabled, as shown in the *setDispenseState()* action above.

The only remaining method in the *ColaMachine* implementation is *setTerminalState()* and is implemented, without comment, as follows.

```
0152     private void setTerminalState(){
0153        dynaMessage.setMessage(   TERMINAL_MESSAGE);
0154        dynaMessage.setShowStyle( DynaLabel.CONTINUAL);
0155        refundButton.setEnabled( false);
```

```
         orangeButton.setEnabled( false);
         colaButton.setEnabled(   false);
         theState = TERMINAL_STATE;
      } // End setTerminalState.
```

This completes the first implementation of the *ColaMachine* and provides a mechanism by which the two hopper classes and the *CoinHopperUI* class can be almost completely demonstrated. For example, the *restock()* method of the *CanHopper* class cannot be demonstrated in this implementation. The motivation to develop the *ColaMachine* was not only to provide an example of the implementation of a more complex STD, but also to provide a realistic example of internationalization. The process for this will be introduced in the next section.

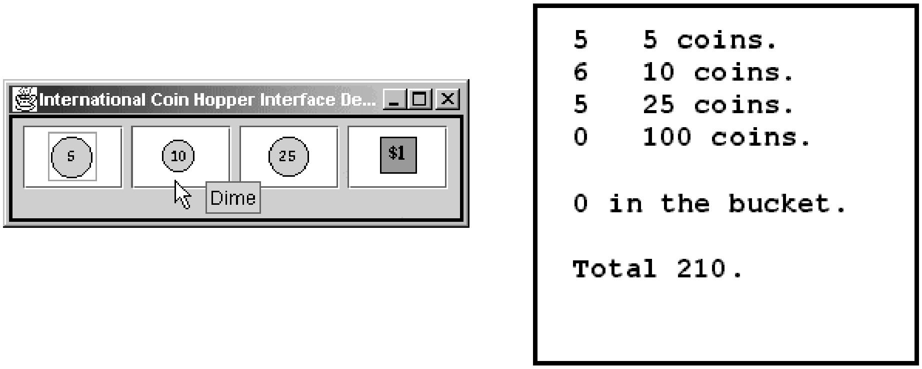
11.6 The *InternationalCoinHopperUIDemo* artifact

Rather than introduce the full range of internationalization options on an artifact as complex as the *ColaMachine* the *CoinHopperUIDemo* example will be redeveloped as the *InternationalCoinHopperUIDemo* and configured so as to use USA denomination coins and $1 bank notes. The appearance of the revised implementation in this configuration is shown in Figure 11.12.

The interface shows the three coins, 5¢, 10¢, 25¢ and a representation of a green one dollar bill. The tool tip for the 10¢ coin is shown as “Dime” and the output produced by pressing this button, immediately after the interface became visible to the user, is shown on the right. It shows that there are now six 10¢ coins in the coin hopper and the total value is 210 (cents).

The strategy for preparing an artifact for international use is to identify all the resources which will change between linguistic and political locales and extract them into a separate resource file which is consulted during applet initialization. For example, the UK resources for the *InternationalCoinHopperUIDemo* artifact were placed in a file called *CoinUIResources*, which commences as follows.

```
package hoppers;

import java.util.ListResourceBundle;
import javax.swing.*;


```



```
5   5 coins.
6   10 coins.
5   25 coins.
0   100 coins.

0 in the bucket.

Total 210.
```

Figure 11.12 *InternationalCoinHopperdemoInterface*: USA configuration.

```
0016 public class CoinUIResources extends ListResourceBundle {
0017
0018 private final static ImageIcon ukIcons[] =
0019    { new ImageIcon( "hoppers/fivep.gif",     "Five P"),
0020      new ImageIcon( "hoppers/tenp.gif",      "Ten P"),
0021      new ImageIcon( "hoppers/twentyp.gif",   "Twenty P"),
0022      new ImageIcon( "hoppers/fiftyp.gif",    "Fifty P"),
0023      new ImageIcon( "hoppers/onepound.gif",  "One £"),
0024      new ImageIcon( "hoppers/twopounds.gif", "Two £") };
0025
0026 private final static String ukNames[] =
0027                       { "Five P",  "Ten P", "Twenty P",
0028                         "Fifty P", "One £", "Two £"};
0029
0030 private final static int ukValues[] =
0031                               { 5, 10, 20, 50, 100, 200};
```

The *CoinUIResources* class is a member of the *hoppers* package of classes and extends the java.util.ListResourceBundle, for reasons which will be explained shortly. The three arrays declared on lines 0018 to 0031 are identical to the declaration of the three arrays containing the UK resources, as previously explained.

The only *ListResourceBundle* method which needs to be overridden in the *CoinUIResources* class is called getContents() and must return a two-dimensional array of Object. The first dimension of the array will be used as an index to obtain the corresponding element of the second dimension of the array when the resource bundle is opened by a client. The implementation of this method in *CoinUIResources*, and the declaration of the two-dimensional array of Object which it returns is as follows.

```
0034 private static final Object[][] contents = {
0035       { "coinIcons",   ukIcons },
0036       { "coinNames",   ukNames },
0037       { "coinValues",  ukValues },
0038    }; // End contents.
0039
0040    public Object[][] getContents() {
0041       return contents;
0042    } // End getContents.
0043 } // End class CoinUIResources.
```

The declaration of the *contents* array, between lines 0034 and 0038, contains in the second dimension the identities of the three arrays containing the resources. The corresponding elements in the first dimension of the array are occupied by Strings which identify the resource. For example the index String in the second location of the array is "*coinNames*" and the corresponding object reference is *ukNames*. The reasons for this organization will become clearer when the *InternationalCoinHopperUIDemo* class is described below. The *getContents()* method is overridden on lines 0040 to 0042 and is implemented as a single Java statement which returns the identity of the *contents* array.

This extended resource bundle is used by the *InternationalCoinHopperUIDemo* class whose implementation as far as the start of its *init()* method, as follows.

```
0010 package hoppers;
0011
```

```
0012  import javax.swing.*;
0013
0014  import java.awt.*;
0015  import java.awt.event.*;
0016  import java.util.*;
0017
0018  public class InternationalCoinHopperUIDemo
0019                                    extends JApplet
0020                            implements ItemListener {
0021
0022  private ImageIcon coinIcons[];
0023  private String    coinNames[];
0024  private int       coinValues[];
0025
0026  CoinHopperUI demoInterface = null;
0027  CoinHopper       theHopper = null;
0028
```

This implementation differs from the *InternationalCoinHopperUI* implementation by importing the java.util package of classes and by declaring, but not initializing, the three arrays on lines 0022 to 0025. The *init()* method is implemented as follows.

```
0029     public void init() {
0030
0031     ResourceBundle resources = null;
0032
0033        resources = ResourceBundle.getBundle(
0034                               "Hoppers.CoinUIResources");
0035        coinIcons  = (ImageIcon[]) resources.getObject(
0036                                             "coinIcons");
0037        coinNames  = (String[])    resources.getObject(
0038                                             "coinNames");
0039        coinValues = (int[])       resources.getObject(
0040                                            "coinValues");
0041
0042        theHopper     = new CoinHopper( coinValues);
0043        demoInterface = new CoinHopperUI( coinIcons,
0044                                          coinNames);
0045        demoInterface.addItemListener( this);
0046        this.getContentPane().add( demoInterface);
0047     } // End init.
```

This implementation commences with the declaration of an instance of the ResourcesBundle class called *resources* and is used in lines 0033 to 0034 to open the *Hoppers.CoinUIResources* bundle, using the ResourceBundle *getBundle()* method. Having successfully open the *resources* bundle its *getObject()* method can be used to retrieve resources from it. The argument to a call of *getObject()* is expected to be the identity of one of the indexes in the first dimension of the array returned by the bundle's *getContents()* method. If the index Object supplied does match one of the indexes then the associated resource in the second dimension is returned, as an Object, from the array.

For example, on line 0037 and 0038, the resources getObject() method is called with the String "*coinNames*" as an argument. This will match the second index value in the array, so the corresponding resource, the String array *ukNames*, will be returned from the call. As this is returned as an Object it must be cast to its known class, array of String, before being assigned to *coinNames*. The other two arrays from the resource bundle are retrieved in a similar manner, cast to their appropriate type and assigned to their referents within this class.

The effect of this first part of the *init()* method is to initialize the three arrays to the same state as that in which they were declared in the *CoinHopperUI* artifact. The remaining part of the *init()* method is essentially identical to the *CoinHopperUI init()* method, so the artifact will appear to the user exactly as shown in Figure 11.8.

Having removed the resources from the *CoinHopperUI* class and placed them into a default resource bundle called *CoinUIResources* it is possible for alternative versions of this class to be produced which, using a systematic naming convention which will be explained shortly, may be used in preference to the default bundle. For example, the systematic name of the *CoinUIResources* bundle class which should contain the resources for use in the USA would be *CoinUIResources_en_US* and is implemented as follows.

```
package hoppers;

import java.util.ListResourceBundle;
import javax.swing.*;

public class CoinUIResources_en_US
                         extends ListResourceBundle {

private final static ImageIcon usIcons[] =
   { new ImageIcon( "Hoppers/fivecents.gif",
                                         "Five cent"),
     new ImageIcon( "Hoppers/tencents.gif",
                                          "Ten cent"),
     new ImageIcon( "Hoppers/twentyfivecents.gif",
                                  "Twenty Five cent"),
     new ImageIcon( "Hoppers/onedollar.gif",
                                             "One $") };

private final static String usNames[] =
                      { "Nickle",  "Dime",
                        "Quarter", "Dollar Bill"};

private final static int usValues[] = { 5, 10, 25, 100};

private static final Object[][] contents = {
      { "coinIcons",   usIcons },
      { "coinNames",   usNames },
      { "coinValues",  usValues },
   }; // End contents.

   public Object[][] getContents() {
      return contents;
   } // End getContents.
} // End class CoinUIResources_en_US.
```

The overall structure of this class is identical to that of the *CoinUIResources* class declaring three equal-sized arrays, one of ImageIcons, one of Strings and one of **int**s. These identify the US version of the resources; for example, the second elements of the arrays will contain the `tencents.gif` image as the icon with the phrase "*Ten cents*" associated with it, the colloquial name for the coin, "*Dime*" and the value in cents, 10[1]. The *contents* array contains the same Strings as indexes in the first dimension but contains references to the US resources in the second dimension.

If the *InternationalCoinHopperUI* artifact is executed on a machine which is configured for use in the USA then the resources in *CoinUIResources_en_US* will be loaded in preference to the default resources in *CoinUIResources* when the resource bundle is opened, and the artifact will appear as in Figure 11.12. However, the alternative resource bundle can be demonstrated by using an alternative version of the ResourceBundle getBundle() method which takes a second argument identifying the locale which should be used. For example, to ensure that the US resource bundle is always opened (if available) the revised call of getBundle() would be as follows.

```
       resources = ResourceBundle.getBundle(
                        "Hoppers.CoinUIResources",
                                        Locale.US);
```

The second argument is a manifest constant from the java.util.Locale class called US which identifies the USA. The first argument remains, as before, the name of the default resource bundle.

A Locale identifies both a language and a country by means of standard two letter codes. The Locale.US contains the code *en* for English language and *US* for the USA. The getBundle() method will first attempt to obtain a resource bundle by adding both the language code and the country code to the end of the default bundle name. In this example this would be *CoinUIResources_en_US*. If this class cannot be opened then the country code will be removed and a second attempt made, in this example an attempt to open *CoinUIResources_en*. Only if both these attempts fail will an attempt be made to open the default bundle, *CoinUIResources*.

Hence if the US resources were required to be the default for this artifact then they should be placed in the *CoinUIResources* class and the UK resources placed in a class called *CoinUIResources_en_UK*. Likewise, if it were required to support the two major linguistic communities in Canada the two files required would be *CoinUIResources_en_CA* and *CoinUIResources_fr_CA*, the difference between the two classes being the language used in the strings.

The two-letter codes were not chosen arbitrarily but are contained in ISO specifications 639:1998 for the language codes and 3166 for the country codes. Examples of the commonest codes are given in Tables 11.1 and 11.2. Details of how to obtain the complete list of codes are contained in the Appendix.

The same technique can be used in the *ColaMachine* artifact, involving the removal of the explicit resource declarations from the end of the `ColaMachine.java` file and placing them into a resource bundle file, which will be called *ColaMachineResources*.

[1] This clarifies the relationship between the name associated with the Icon and the name used in the ToolTip. British English used to have a large number of colloquial names for coins, such as "tanner" or "half-bob", but these were lost when the currency was decimalized in 1971, and colloquial names for the new coins do not seem to have evolved (yet?).

Table 11.1 Common ISO 639:1998 language codes.

Arabic	ar	Chinese	zh	Danish	da
Dutch	nl	English	en	Finnish	fi
French	fr	German	de	Greek	el
Hebrew	iw	Irish	ga	Italian	it
Japanese	ja	Korean	ko	Norwegian	no
Portugese	pt	Russian	ru	Spanish	es

Table 11.2 Common ISO 3166 region codes.

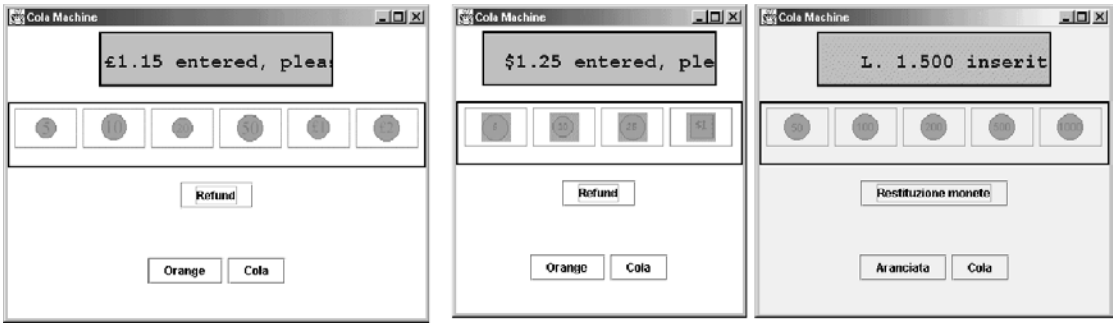
Australia	AU	Brazil	BR	Canada	CA
China	CN	Denmark	DK	Finland	FI
France	FR	Germany	DE	Greece	GR
India	IN	Ireland	IE	Israel	IL
Italy	IT	Japan	JP	Korea (south)	KR
Mexico	MX	Netherlands	NL	New Zealand	NZ
Norway	NO	Portugal	PR	Russian Federation	RU
Singapore	SG	South Africa	ZA	Spain	ES
Sweden	SE	United Kingdom	UK	United States	US

This resource bundle can then be opened at the start of the *ColaMachine init()* method and the resources which it contains retrieved and used to initialize the resource Objects. For example, the Italian version of the resources file, called *ColaMachineResources_it_IT*, might be as follows.

```
public class ColaMachineResources_it_IT
                                extends ListResourceBundle {


private final static ImageIcon itIcons[] =
 { new ImageIcon( "hoppers/fiftyl.gif",    "Cinquanta Lit."),
 new ImageIcon( "hoppers/hundredl.gif",    "Cento Lit."),
 new ImageIcon( "hoppers/twohundredl.gif", "Duecento Lit."),
 new ImageIcon( "hoppers/fivehundredl.gif","Cinquecento Lit."),
 new ImageIcon( "hoppers/thousandl.gif",   "Mille Lit."),};

private final static String itNames[] =
             { "Cinquanta Lit.", "Cento Lit.", "Duecento Lit.",
               "Cinquecento Lit.", "Mille Lit."};

private final static int itValues[] =
                                  { 50, 100, 200, 500, 1000};


0034  private static final int    CAN_COST             = 1500;
0035  private static final double CURRENCY_DENOMINATOR = 1.0;
0036
0037  private static final String RESET_MESSAGE =
0038                 " Premere un bottone per inserire una moneta. ";

0062  private static final String DISPENSE_PROMPT =
0063         " inserite, si prega selezionare la bibita desiderata.";
0064
0065
0066     static final Object[][] contents = {
0067      { "coinIcons",            itIcons },
0068      { "coinNames",            itNames },
0069      { "coinValues",           itValues },
0070      { "canCost",             new Integer( CAN_COST) } ,
0071      { "currencyDenominator", new Double(CURRENCY_DENOMINATOR)},
0072      { "resetMessage",         RESET_MESSAGE},
0073      { "correctChangeMessage", CORRECT_CHANGE_MESSAGE},
0074      { "terminalMessage",      TERMINAL_MESSAGE},
0075      { "noChangeMessage",      NO_CHANGE_MESSAGE},
0076      { "orangeMessage",        ORANGE_MESSAGE },
0077      { "colaMessage",          COLA_MESSAGE},
0078      { "refundLabel",          REFUND_LABEL},
0079      { "orangeLabel",          ORANGE_LABEL},
0080      { "colaLabel",            COLA_LABEL},
0081      { "coinsLiteral",         COINS_LITERAL},
0082      { "refundDialogTitle",    REFUND_DIALOG_TITLE},
0083      { "noChangeDialogTitle",  NOCHANGE_DIALOG_TITLE},
0084      { "dispenseDialogTitle",  DISPENSE_DIALOG_TITLE},
0085      { "dispensePrompt",       DISPENSE_PROMPT},
0086     }; // End contents.
0087
0088     public Object[][] getContents() {
0089        return contents;
0090     } // End getContents.
0091
0092  } // End class ColaMachineResources_it_IT.
```

The declaration of the *contents* array on lines 0066 to 0086 indicates that there are a total of 19 resources stored in this bundle. These include the images to be used for the coin buttons, declared on lines 0019 to 0024, and the associated values on lines 0030 to 0031. The cost of a can is declared on line 0034 and the use made of the *CURRENCY_DENOMINATOR* resource, declared on line 0035, will be described below. The remaining declarations are all Strings, many of which have been omitted from this listing.

Figure 11.13 shows the *InternationalColaMachine* artifact in UK, US and Italian configurations. The most noticeable difference between this implementation of the cola machine and that presented above, apart from the coin buttons, is the formatting of the currency value in the *DynaLabel* message. This has been accomplished by using NumberFormat capabilities.



£1.15 entered, plea

$1.25 entered, ple

L. 1.500 inserit

Figure 11.13 *InternationalColaMachine* in UK, US and Italian configurations.

Object → Format → java.text.NumberFormat

```
public NumberFormat()
```

NumberFormat instances are not normally constructed but obtained using a factory method.

```
public static NumerFormat getNumberInstance()
public static NumerFormat getNumberInstance(   Locale forThisLocale)
public static NumerFormat getCurrencyInstance()
public static NumerFormat getCurrencyInstance( Locale forThisLocale)
public static NumerFormat getPercentInstance()
public static NumerFormat getPercentInstance(  Locale forThisLocale)
```

Class-wide factory methods to obtain NumberFormat instances.

```
public StringBuffer format( long   toFormat)
public StringBuffer format( double toFormat)
```

Formats the numeric values *toFormat* according to the rules of the instance.

The NumberFormat class extends the java.text.Format class and can be used to obtain objects which contain the knowledge of how to present numeric values in a manner appropriate to the locale. For example it is customary in the UK and USA to use a full stop (period) to represent a decimal point and group digits appearing before it in threes, separated by commas (e.g. 1,000,000.00). However, in some European countries, for example France, a comma is used as the decimal point and full stops are used for grouping. The class-wide NumberFormat getNumberInstance() method will return a Format instance whose format() method will take an numeric value as an argument and return a String containing the locale appropriate representation of the value. For example, consider the following fragment.

```
NumberFormat formatter = NumberFormat.getNumberInstance();
System.out.println( formatter.format( 1234567.89));
```

This would produce the output 1,234,567.89 in the UK or US locales but 1.234.567,89 in the FR locale. On the first line a NumberFormat instance called *formatter* is requested and the object that is obtained will be capable of formatting numeric values in a manner appropriate to the locale that the artifact is operating within. On the second line the format() method of the *formatter* instance is used to prepare a String containing a formatted representation of the value supplied. In the UK or USA this would be the first example shown at the start of this paragraph, but in France it would be as in the second example.

The *InternationalColaMachine* class contains a NumberFormat instance called *currencyFormatter* which is initialized in the class's *addNotify()* method as follows.

```
public void addNotify() {
  currencyFormatter = NumberFormat.getCurrencyInstance()
  super.addNotify();
} // End addNotify.
```

Any locale specific-resources must be obtained in an addNotify() method, rather than in an init() method or a constructor, as it is only when the artifact has been initialized that it knows which locale it is operating within. The class-wide NumberFormat getCurrencyInstance() method will return a NumberFormat object which will contain the knowledge of how the value supplied is to be rendered as a currency value. These considerations include the symbol that is to be used to indicate currency, the position of the symbol before or after the value, and the number of decimal points to use.

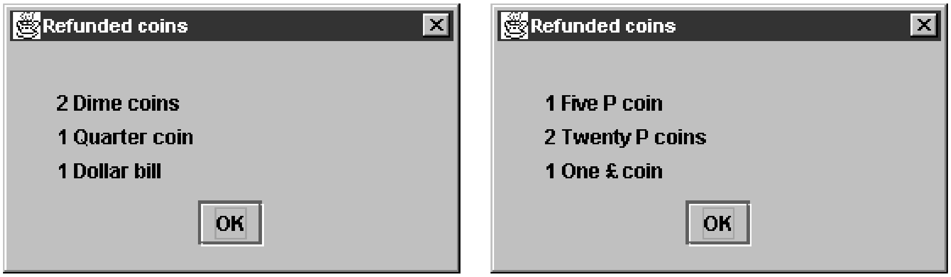
Figure 11.13 shows that the correct symbol (£, $ or L.) has been obtained and also shows that the pound and dollar values have two decimal places while the lira value has none. The *ColaMachine* operates in integral (penny, cent or lira) values and the *currencyDenominator* value, supplied by the resource bundle, is used to convert the value immediately prior to being formatted. The fragment of code responsible for producing the *DynaLabel* messages shown in Figure 11.13 is as follows.

```
0213       dynaMessage.setMessage(
0214                    currencyFormatter.format(
0215                            (enteredValue / currencyDenominator)) +
0216                                                        dispensePrompt);
0217       dynaMessage.setShowStyle( DynaLabel.CONTINUAL);
```

On lines 0214 to 0215 the *currencyFormatter format()* method is used to obtain the currency value which precedes the *dispensePrompt*. The argument to the *format()* method is the *enteredValue*, an **int**eger, divided by the *currencyDenominator*, a **double**. In the Italian locale the *currencyDenominator* value would be 1.0, resulting in the unchanged value being passed to the *currencyFormatter*. In the UK or USA the *currencyDenominator* value would be 100.0, resulting in the value in pennies or cents being converted into a value in pounds or dollars before being passed to the *currencyFormatter*.

The NumberFormat class has a sibling class called DateFormat which can be used for formatting dates and times. It also has two child classes called DecimalFormat and ChoiceFormat. The DecimalFormat class can be used for more precise formatting of floating-point values. The ChoiceFormat class can be used to ensure that the natural language syntax for numeric agreement is used in messages to the user.

For example, Figure 11.14 shows a US *Refunded Coins* message which might have been produced by the *ColaMachine* class and one produced by the *InternationalColaMachine* class when upgraded to make use of ChoiceFormat capabilities. In the original version the last term of each line of the message was a clumsy and inelegant "*coin(s)*.", which was



2 Dime coins
1 Quarter coin
1 Dollar bill

1 Five P coin
2 Twenty P coins
1 One £ coin

Figure 11.14 Refund dialogs using *ChoiceFormatters*.

provided even if the thing being returned was a bank note (bill). The revised version has correctly formatted the final phrase as either "*coin*." or "*coins*" (or "*bill*" or "*bills*.") depending upon the number of things being returned.

Object → Format → NumberFormat→ java.text.ChoiceFormat

```
public ChoiceFormat( double[] limits, String[] phrases)
```

Constructor which prepares an instance which will associate *phrases* with the most appropriate *limit*.

```
public StringBuffer format( long   toFormat)
public StringBuffer format( double toFormat)
```

Returns the most appropriate *phrase* given the value *toFormat*.

A *ChoiceFormat* instance is constructed by supplying it with an array of **double** values and a corresponding array of Strings. Its format() method will take a numeric value as an argument and return the most appropriate String. For example, a *ChoiceFormat* instance called *coinNamer* whose format() method will output "*coin*" or "*coins*" is constructed in the US and UK resource files as follows.

```
private static final double[] coinTermLimits = {0.0, 2.0};
private static final String[] coinTerms = {"coin", "coins"};
private static final ChoiceFormat coinNamer =
                        new ChoiceFormat( coinTermLimits,
                                                coinTerms);
```

The *coinTermLimits* indicate that for all values less than 2.0 the String *'coin'* should be returned from the format() method and for all values greater than or equal to 2.0 should return *'coins'*. This is a general purpose mechanism which allows for linguistic environments that may have different phrases associated with negative, zero, one, two, many and lots of things.

The *coinNamer* object is accompanied by a *billNamer* instance in the US version of the resources file and then a *coinNamers* array of *ChoiceFormat* instances is prepared, as follows.

```
private static final double[] billTermLimits = {0.0, 2.0};
private static final String[] billTerms = {"bill", "bills"};
private static final ChoiceFormat billNamer =
                        new ChoiceFormat( billTermLimits,
                                               billTerms);

private static final ChoiceFormat coinNamers[] =
          { coinNamer, coinNamer, coinNamer, billNamer };
```

The *billNamer* object is very similar to the *coinNamer* instance allowing for the production "*bill*" or "*bills*" as appropriate. The *coinNamers* array should contain a ChoiceFormat instance for each coin denomination. The *coinNamers* array is placed into the *contents* array, with the associated key *coinNamers* as follows.

```
      { "coinNamers",           coinNamers},
   }; // End contents.
```

In the *InternationalColaMachine init()* method the array is retrieved from the ResourceBundle instance called *resources* and referenced by a ChoiceFormat array called *coinNamers* as follows.

```
   coinNamers  = (ChoiceFormat[]) resources.getObject(
                                               "coinNamers");
```

This array is then used within the *getRefundString()*, which converts a value supplied as an argument into a String representation, as follows.

```
private String getRefundString( int forThisAmount) {

int[]  coinsToReturn = null;
String refundString  = "";
int    numberOfCoins = 0;
String lastPhrase    = null;

   try {
      coinsToReturn = coinHopper.giveChange(
                                     forThisAmount);
   } catch ( HopperException exception) {
      // Do Nothing!
   } // End try catch.
   for ( int thisCoin = 0;
             thisCoin < coinsToReturn.length;
             thisCoin++) {
      if ( coinsToReturn[ thisCoin] > 0 ) {
         numberOfCoins = coinsToReturn[ thisCoin];
         lastPhrase    = coinNamers[ thisCoin].
                                 format( numberOfCoins);

         refundString = refundString.concat( "\n " +
                                  numberOfCoins + " "  +
                                  coinNames[ thisCoin] +
                                 " " + lastPhrase + ".");
      } // End if.
   } // End for.
   return refundString;
} // End getRefundString.
```

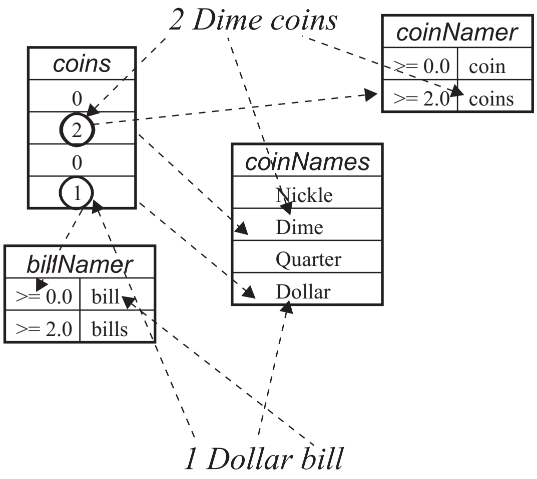


Figure 11.15 *InternationalColaMachine*: choice formatter details.

The key part of this method is lines 0362 to 0364, where a single line of the message is assembled. The message line starts, on code line 0362, with the number of coins or bills that are to be returned to which is appended, on line 0363, the name of the coin or bill. The message line is concluded, on line 0364, with the *lastPhrase*, which has been obtained from one of the ChoiceFormatters on lines 0358 and 0359. The first part of this expression *coinNamers*[*thisCoin*], selects one of the ChoiceFormatters from the array, depending upon the *thisCoin* loop index. Referring to the declaration of the *coinNamers* array this will be either the *coinNamer* of *billNamer* instance, as appropriate.

Having selected the appropriate ChoiceFormat instance, the second part of the phrase .format(*numberOfCoins*) will select the singular or plural, depending upon the value of *numberOfCoins*. Referring to the right-hand image in Figure 11.14 the message line "*2 Dime coins.*" would have been produced when the value of *thisCoin* was 1 and numberOfCoins was 2. The value of *thisCoin* would have selected a *coinNamer* instance whose format() method would return "*coins*" when called with the value 2. The second line of the message "*1 Dollar bill*" would have been produced when the value of of *thisCoin* was 3 and numberOfCoins was 1. The value of *thisCoin* would have selected the *billNamer* instance whose format() method would return "*bill*" when called with the value 1.

Figure 11.15 attempts to further explain this process. The *coins* array on the left indicates that there are 0 5¢ coins, 2 10¢ coins, 0 25¢ coins and 1 $1 dollar bill to be described. The 2 10¢ coins produce the phrase shown at the top of the diagram using the *Dime* name from the *coinNames* array and the *coinNamer* ChoiceFormat instance from the *coinNamer* array. As the number of coins to be described is plural the *coinNamer* format() method will return *coins* rather than *coin*. The 1 $1 dollar bill produces the phrase shown at the bottom of the diagram using the *Dollar* name from the *coinNames* array and the *billNamer* ChoiceFormat instance from the *coinNamer* array. As the number of bills to be described is singular the *billNamer* format() method will return *bill* rather than *bills*.

11.7 I18n and L12n

The considerations which have been described in this chapter are referred to as *Internationalization* and *Localization*, which are commonly abbreviated to *I18n* and *L12n* (there are 18 letters between the initial *I* and terminal *n* in Internationalization and likewise 12 between *L* and *n* in Localization). The positioning of the I18n and L12n considerations and facilities so late in this book does not imply that they are unimportant. Indeed, to give then their due prominence they should have been placed in the first part of the first chapter and continually reinforced in all the examples. However, although not particularly complicated, the techniques are somewhat complex and would have occluded many other points made in the majority of the chapters.

Using the design principle that details matter, the care and attention required for the production of an artifact that can adapt to its locale will bring benefit to the organization that is providing the resource. This is of particular importance when Java artifacts are being deployed via the World Wide Web. In such circumstances the most acceptable default locale would be US English, but customized alternative resources could be easily produced if the artifact is being shown to be extensively used in a particular region or in support of having it adopted in a new region.

Summary

- An artifact, particularly one intended for use on the Web, should be designed from the outset with internationalization (I18n) and localization (L12n) in mind.
- National and linguistic two-letter codes are defined by an ISO standard and are used by Java's I18n and L12n facilities.
- The ListResourceBundle class, which extends the ResourceBundle class, is a convenient class to extend to produce a collection of I18n and L12n resources.
- When the ResourceBundle getResourceBundle() method is used it will conduct a search, using the host machine's I18n and L12n codes, for the most suitable available bundle. It will open the named default bundle only if no more suitable one can be located.
- A resource bundle can contain any class of Object.
- The java.text package contains a number of format packages which can be used for locale-specific formatting of numbers, currency and dates, as well as more complex facilities for number/verb agreement and the sequencing of phrases within prompts.

Exercises

11.1 Provide additional *ColaMachineResource* bundles for other locales.

11.2 Investigate the protocol of the Locale class which contains a number of manifest locales, e.g. Italy or Japan. Design and implement an artifact which allows the user to select a supported locale and then illustrates number, date and currency formatting for that locale.

11.3 Revisit any of the artifacts introduced in any of the chapters or end of chapter exercises, and re-engineer it to be locale-aware.

11.4 Reimplement the *InternationalColaMachine* so that it contains a main menu bar affording the user the opportunity to change the locale.

11.5 Design and implement a simulation of a bank cash dispenser machine with localization considered from the outset of the process.

12

Trees and tables

12.1 Introduction

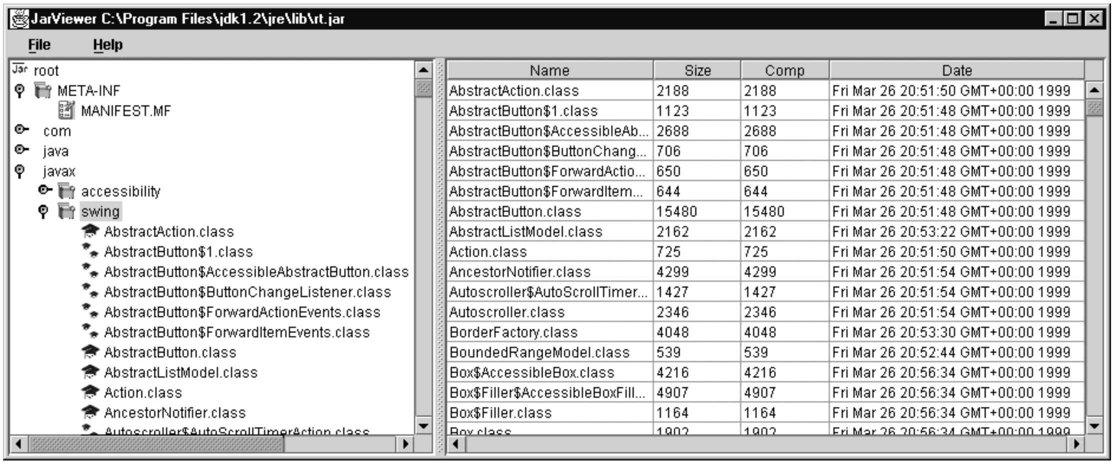
This last chapter will introduce the JFC classes which supply tree view and table capability. A tree view is very suitable for investigating a hierarchical structure such as a file system, using utilities that have become known as *explorers*. Conventionally in explorers a directory is presented with a graphic icon to identify it, usually a representation of a folder. Double-clicking upon the icon will 'open the folder' with its contents shown connected to it. The contents of a folder can contain other folders, which can be opened to show other folders, etc. This capability will be illustrated by the development of an artifact known as *JarViewer* which supplies a tree view of the contents of a Java Archive File (`*.jar`).

When a folder, or jar entry, is selected in an explorer tool an area alongside the tree view can be used to supply additional information. In the *JarViewer* utility various different views will be provided depending upon the entry selected. For example, any text entries will be shown within a text area, HTML entries will be shown within a JEditorPane and icon resources will be imaged. Most significantly when a package entry is selected a table view will be used to show its contents. A table is a composite component that contains an array of cells, rather like a spreadsheet, and may also contain horizontal and vertical header rows.

12.2 The *JarViewer* artifact – overview

The general appearance of the *JarViewer* artifact is illustrated in Figure 12.1. It contains a main menu above a horizontal split pane. The left-hand area of the split pane contains a scroll-pane with a viewport onto a JTree instance. The contents of the right-hand scroll-pane depend upon the currently selected node in the tree. When the artifact is first launched both sides of the split pane are empty. The application-level main menu is deliberately simple, consisting of a *File* menu which contains *Open...* and *Exit...* items and a *Help...* menu which contains a *Version...* item. The *Open...* item posts a standard file open dialog, as described in Chapter 9, containing a file filter which masks all but `*.jar` files. When the user selects a jar file a JTree instance containing a view of its contents is constructed and placed in the left-hand split pane.

The tree in Figure 12.1 is showing a view of the *r*un-*t*ime jar (`rt.jar`) file supplied as a part of the JDK 1.2 Java Runtime Environment Utility (rte). It is this file which contains the standard Java class libraries which are used by the JDK environment. In the illustration the javax.swing package has been *expanded*, as shown by the small downward-facing handle alongside its list contents shown below and indented from it. In contrast, the



```
root
  META-INF
    MANIFEST.MF
  com
  java
  javax
    accessibility
    swing
      AbstractAction.class
      AbstractButton$1.class
      AbstractButton$AccessibleAbstractButton.class
      AbstractButton$ButtonChangeListener.class
      AbstractButton$ForwardActionEvents.class
      AbstractButton$ForwardItemEvents.class
      AbstractButton.class
      AbstractListModel.class
      Action.class
      AncestorNotifier.class
      Autoscroller$AutoScrollTimerAction.class
```

Name	Size	Comp	Date
AbstractAction.class	2188	2188	Fri Mar 26 20:51:50 GMT+00:00 1999
AbstractButton$1.class	1123	1123	Fri Mar 26 20:51:48 GMT+00:00 1999
AbstractButton$AccessibleAb...	2688	2688	Fri Mar 26 20:51:48 GMT+00:00 1999
AbstractButton$ButtonChang...	706	706	Fri Mar 26 20:51:48 GMT+00:00 1999
AbstractButton$ForwardActio...	650	650	Fri Mar 26 20:51:48 GMT+00:00 1999
AbstractButton$ForwardItem...	644	644	Fri Mar 26 20:51:48 GMT+00:00 1999
AbstractButton.class	15480	15480	Fri Mar 26 20:51:48 GMT+00:00 1999
AbstractListModel.class	2162	2162	Fri Mar 26 20:53:22 GMT+00:00 1999
Action.class	725	725	Fri Mar 26 20:51:50 GMT+00:00 1999
AncestorNotifier.class	4299	4299	Fri Mar 26 20:51:54 GMT+00:00 1999
Autoscroller$AutoScrollTimer...	1427	1427	Fri Mar 26 20:51:54 GMT+00:00 1999
Autoscroller.class	2346	2346	Fri Mar 26 20:51:54 GMT+00:00 1999
BorderFactory.class	4048	4048	Fri Mar 26 20:53:30 GMT+00:00 1999
BoundedRangeModel.class	539	539	Fri Mar 26 20:52:44 GMT+00:00 1999
Box$AccessibleBox.class	4216	4216	Fri Mar 26 20:56:34 GMT+00:00 1999
Box$Filler$AccessibleBoxFill...	4907	4907	Fri Mar 26 20:56:34 GMT+00:00 1999
Box$Filler.class	1164	1164	Fri Mar 26 20:56:34 GMT+00:00 1999
Box.class	1902	1902	Fri Mar 26 20:56:34 GMT+00:00 1999

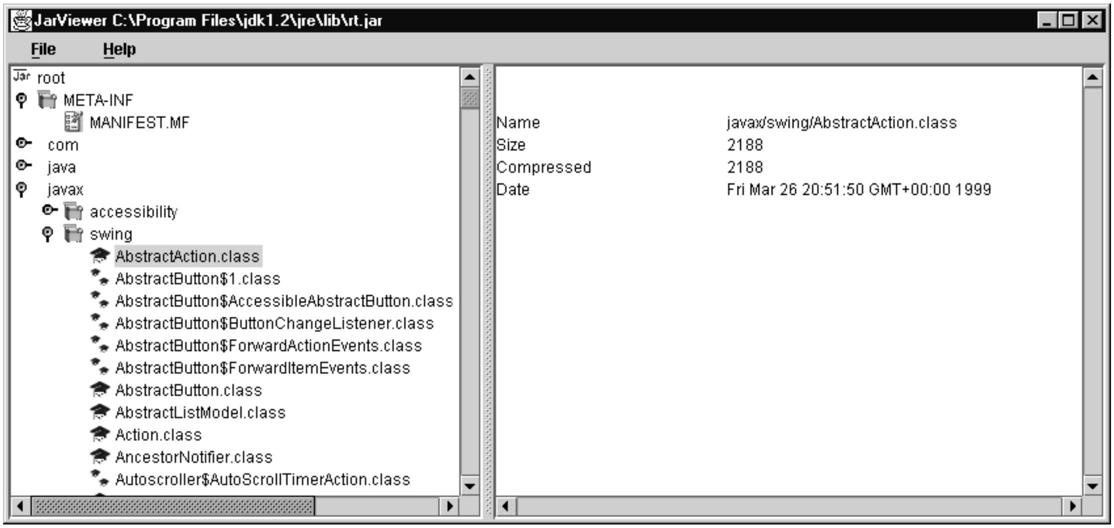
Figure 12.1 The *JarViewer* artifact showing a package view.

javax.accessibility package, immediately above javax.swing, has not been expanded, as shown by the right-facing handle alongside it. The javax.swing package has also been *selected*, as shown by the highlight, and the right-hand pane contains a JTable instance, each row of which lists four attributes of each of the entries contained in the package. The attributes are the name of the entry, its size before compression, its size after compression and the date when it was created or last modified. Class files do not compress significantly and are stored in an uncompressed form, as can be inferred from the table.

The box icon alongside the javax.swing and javax.accessibility entries is indicating that they are packages; that is, they contain `*.class` and/or other resources. The first item in the list of javax.swing contents, AbstractAction.class, is accompanied by a teacher's mortarboard icon to indicate that it is a class file. The second item in the list, AbstractButton$1.class, is an anonymous inner class contained within the AbstractButton.class and the double mortarboard icon is indicating that it is a contained class. The next four items on the list are non-anonymous contained classes of the AbstractButton.class, before the AbstractButton.class itself is shown. Contained classes are classes that are totally encapsulated within other classes and can, as indicated above, be anonymous. They have not been used in this book and will not be used in this chapter as, although they can be very valuable for experienced developers, they can obscure an artifact's architecture.

Figure 12.2 shows the *JarViewer* artifact when a `*.class` file has been selected in the tree view. The right-hand split pane is showing the information that was displayed in the class's row in the package view. For some, but not all classes, additional information concerning its security encoding or certification is available, but as the Swing classes are intended for very general distribution this is not appropriate in this case.

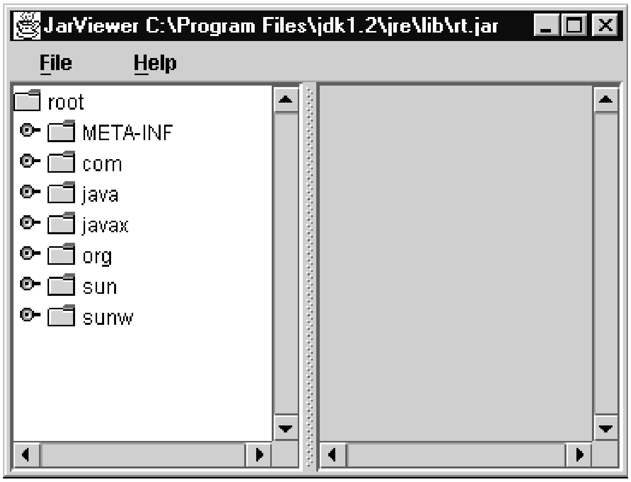
When the artifact is first presented to the user, as shown in Figure 12.3, only the root of the tree is expanded with only its top-level elements visible. The first element of every jar file is the meta information (`META_INF`), which always contains a manifest file called `MANIFEST.MNF`. This is a text file and, like any other text files which are contained within the archive such as `*.text` or `*.java` files, when it is selected its contents are shown within a JTextArea instance. This is illustrated in Figure 12.4, where the start of the `rt.jar` manifest file is shown.



root
META-INF
MANIFEST.MF
com
java
javax
accessibility
swing
AbstractAction.class
AbstractButton$1.class
AbstractButton$AccessibleAbstractButton.class
AbstractButton$ButtonChangeListener.class
AbstractButton$ForwardActionEvents.class
AbstractButton$ForwardItemEvents.class
AbstractButton.class
AbstractListModel.class
Action.class
AncestorNotifier.class
Autoscroller$AutoScrollTimerAction.class

Name	javax/swing/AbstractAction.class
Size	2188
Compressed	2188
Date	Fri Mar 26 20:51:50 GMT+00:00 1999

Figure 12.2 The *JarViewer* artifact showing a class view.

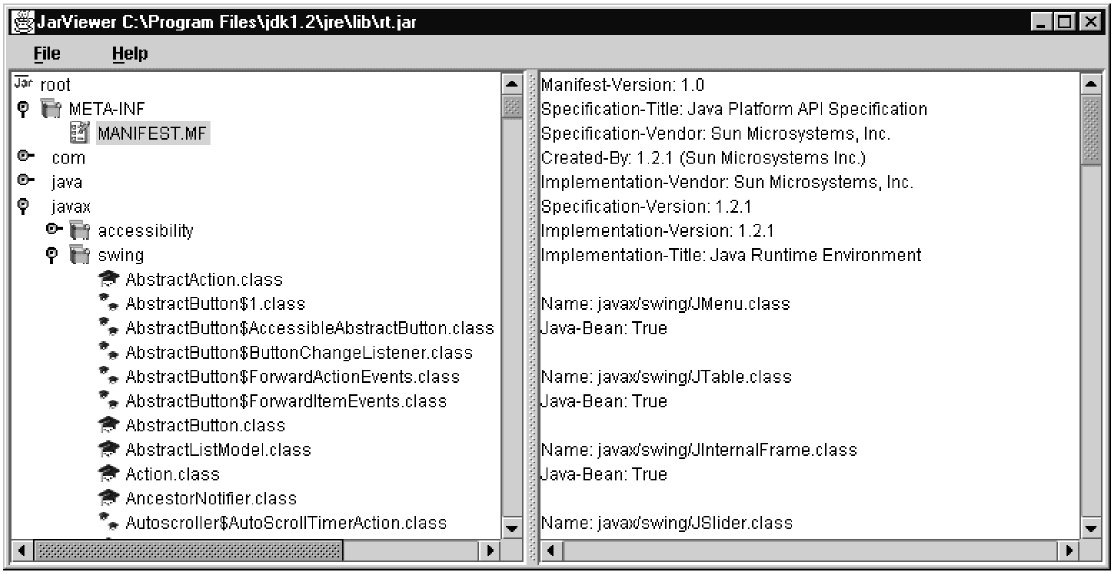


root
META-INF
com
java
javax
org
sun
sunw

Figure 12.3 The *JarViewer* artifact showing an initial tree view.

An image view is supplied for `*.jpeg` or `*.gif` image files; Figure 12.5 shows the *JarViewer* artifact with the `Question.gif` file, containing the question mark image used on the JOptionPane question dialog, selected and displayed in the right-hand split pane.

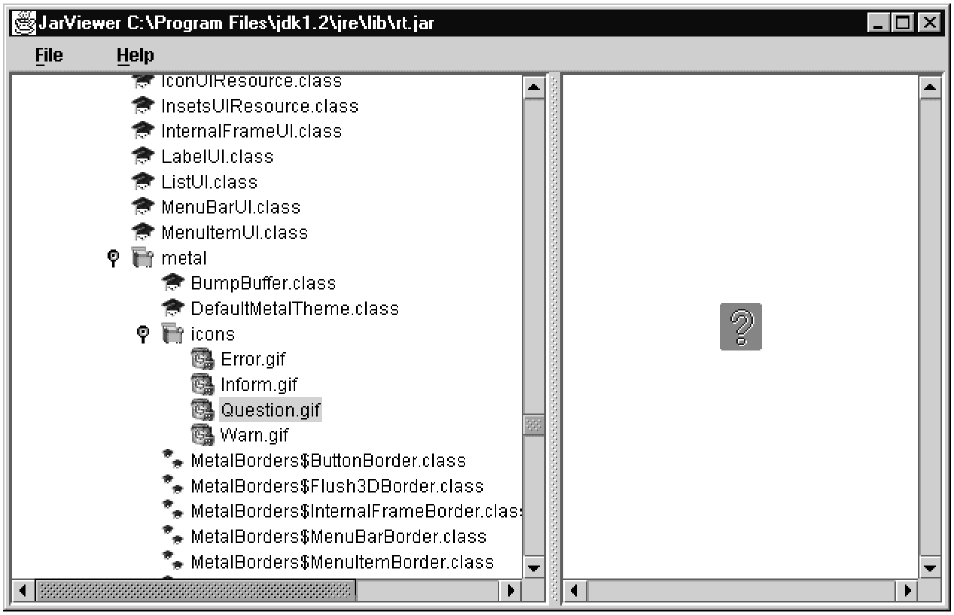
Any entries which are not associated with one of the views, as described above, apart from the HTML view, are labeled with an unknown icon and the right-hand area of the split pane is cleared when they are selected. Figure 12.6 illustrates the tree view of an artificially prepared jar archive which shows all possible icons. It also illustrates the selection of an HTML file with its contents rendered within a suitably configured JEditorPane instance in the right-hand split pane.



```
root
  META-INF
    MANIFEST.MF
  com
  java
  javax
    accessibility
    swing
      AbstractAction.class
      AbstractButton$1.class
      AbstractButton$AccessibleAbstractButton.class
      AbstractButton$ButtonChangeListener.class
      AbstractButton$ForwardActionEvents.class
      AbstractButton$ForwardItemEvents.class
      AbstractButton.class
      AbstractListModel.class
      Action.class
      AncestorNotifier.class
      Autoscroller$AutoScrollTimerAction.class
```

```
Manifest-Version: 1.0
Specification-Title: Java Platform API Specification
Specification-Vendor: Sun Microsystems, Inc.
Created-By: 1.2.1 (Sun Microsystems Inc.)
Implementation-Vendor: Sun Microsystems, Inc.
Specification-Version: 1.2.1
Implementation-Version: 1.2.1
Implementation-Title: Java Runtime Environment

Name: javax/swing/JMenu.class
Java-Bean: True

Name: javax/swing/JTable.class
Java-Bean: True

Name: javax/swing/JInternalFrame.class
Java-Bean: True

Name: javax/swing/JSlider.class
```

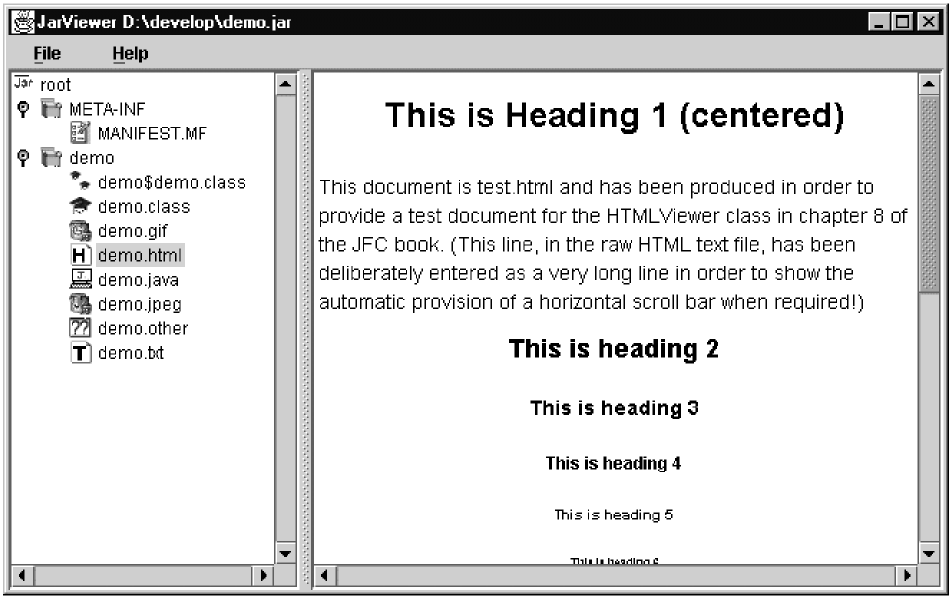
Figure 12.4 The *JarViewer* artifact showing a text view.



```
      IconUIResource.class
      InsetsUIResource.class
      InternalFrameUI.class
      LabelUI.class
      ListUI.class
      MenuBarUI.class
      MenuItemUI.class
    metal
      BumpBuffer.class
      DefaultMetalTheme.class
      icons
        Error.gif
        Inform.gif
        Question.gif
        Warn.gif
      MetalBorders$ButtonBorder.class
      MetalBorders$Flush3DBorder.class
      MetalBorders$InternalFrameBorder.class
      MetalBorders$MenuBarBorder.class
      MetalBorders$MenuItemBorder.class
```

Figure 12.5 The *JarViewer* artifact showing an image view.

A JTree instance is a container whose extent fills the entire left-hand split pane in the illustrations presented above. It contains a collection of *nodes*, commencing with the *root node*, which can be either *expanded* or *collapsed* and, for the *JarViewer*, one and at most one node in the tree can be selected. All nodes are connected to the root node by a *path* of



This is Heading 1 (centered)

This document is test.html and has been produced in order to provide a test document for the HTMLViewer class in chapter 8 of the JFC book. (This line, in the raw HTML text file, has been deliberately entered as a very long line in order to show the automatic provision of a horizontal scroll bar when required!)

This is heading 2

This is heading 3

This is heading 4

This is heading 5

This is heading 6

Figure 12.6 The *JarViewer* artifact, showing all icons and an HTML view.

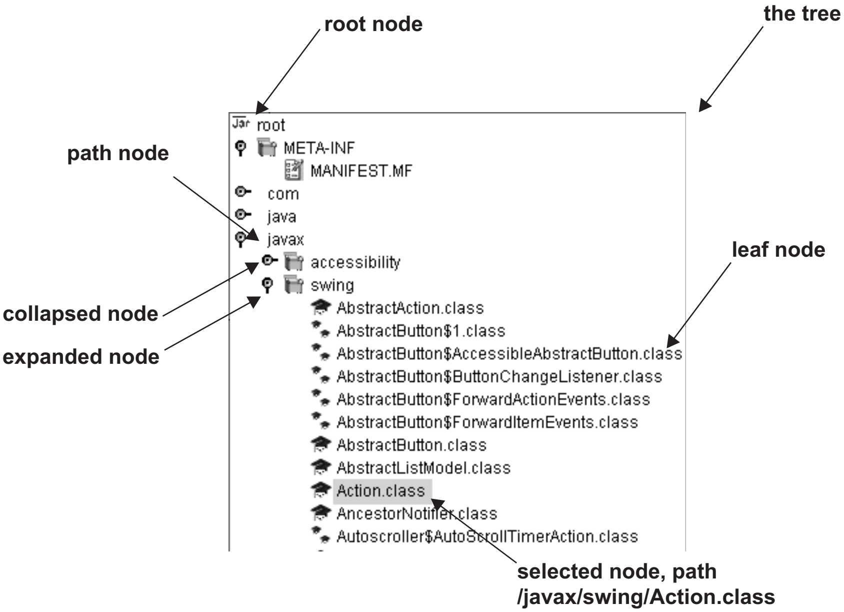

root node
the tree
path node
leaf node
collapsed node
expanded node
selected node, path /javax/swing/Action.class
root
META-INF
MANIFEST.MF
com
java
javax
accessibility
swing
AbstractAction.class
AbstractButton$1.class
AbstractButton$AccessibleAbstractButton.class
AbstractButton$ButtonChangeListener.class
AbstractButton$ForwardActionEvents.class
AbstractButton$ForwardItemEvents.class
AbstractButton.class
AbstractListModel.class
Action.class
AncestorNotifier.class
Autoscroller$AutoScrollTimerAction.class

Figure 12.7 JTree: general terminology.

nodes with those at the end of the path, which have no *child nodes*, being known as *leaf nodes*. All nodes, apart from the root node, have a single *parent node* and nodes that only have non-leaf child nodes are known as *path nodes*. This terminology is illustrated in Figure 12.7, which also indicates that the path to a particular node can be represented by catenating the names of all the nodes between the node being identified and the root node, which is represented by a slash (/).

Figure 12.8 contains a context diagram for the *JarViewer* artifact. It shows that the applet is an instance of the *JarViewer* class and makes use of an instance of the *JarViewArchive* class called *theArchive*, which it uses to obtain details about the entries in a jar file. The *JarViewArchive* class is an extension of the JarFile class, which itself is an extension of the ZipFile class. This indicates that jar files use zip algorithms and protocols to compress and organize their contents; the two pre-supplied Java classes provide the methods to decompress and extract the contents.

The *jarViewer* instance also contains a *JarTree* instance called *theTree* which is an extended JScrollPane. The scroll pane's viewport contains an instance of the JTree class, called *theTree*, which contains a *rootNode*, of the *JarViewNode* class, attached to which are other anonymous instances of the same class. The *JarViewNode* class is an extension of the DefaultMutableTreeNode class which supplies the essential resources and methods for a tree node that can change (mutate) from a leaf to a non-leaf node. The JTree instance also makes use of an instance of the *JarViewNodeRenderer* class, which is responsible for the particular appearance of trees contained within the *jarViewer* artifact. Figure 12.9 contains the same tree as shown in Figure 12.1, but without a renderer, and shows that, by default, leaf nodes are shown with document icons and non-leaf nodes with folder icons.

The context diagram also shows that when the *jarViewer* is showing a table view it makes use of a JTable instance called *theTable*. The JTextAreas, JPanels and JEditorPanes used for the other views have been omitted from the diagram in the interests of clarity.

12.3 The *JarViewArchive* class

The class diagram for the *JarViewArchive* class is given in Figure 12.10. It shows that the *JarViewArchive* class extends the JarFile class and is contained within the *jarview* package of classes. The single constructor requires a File as an argument and will attempt to open it, throwing an exception if the file does not exist or if it is not a `*.jar` file.

The *getContentsList()* method will return an Enumeration, each element of which is a String which identifies the path of one of the entries in the file. For example if the `rt.jar` file were opened the parts of the Enumeration obtained from its *getContentsList()* method which correspond to the view shown in Figure 12.1 would be as follows.

```
META-INF/
META-INF/MANIFEST.MF
com/


java/

javax/
javax/accessibility/

javax/swing/
javax/swing/AbstractAction.class
javax/swing/AbstractButton$1.class
javax/swing/AbstractButton$AccessibleAbstractButton.class
javax/swing/AbstractButton$ButtonChangeListener.class
javax/swing/AbstractButton$ForwardActionEvents.class
javax/swing/AbstractButton$ForwardItemEvents.class
javax/swing/AbstractButton.class
```

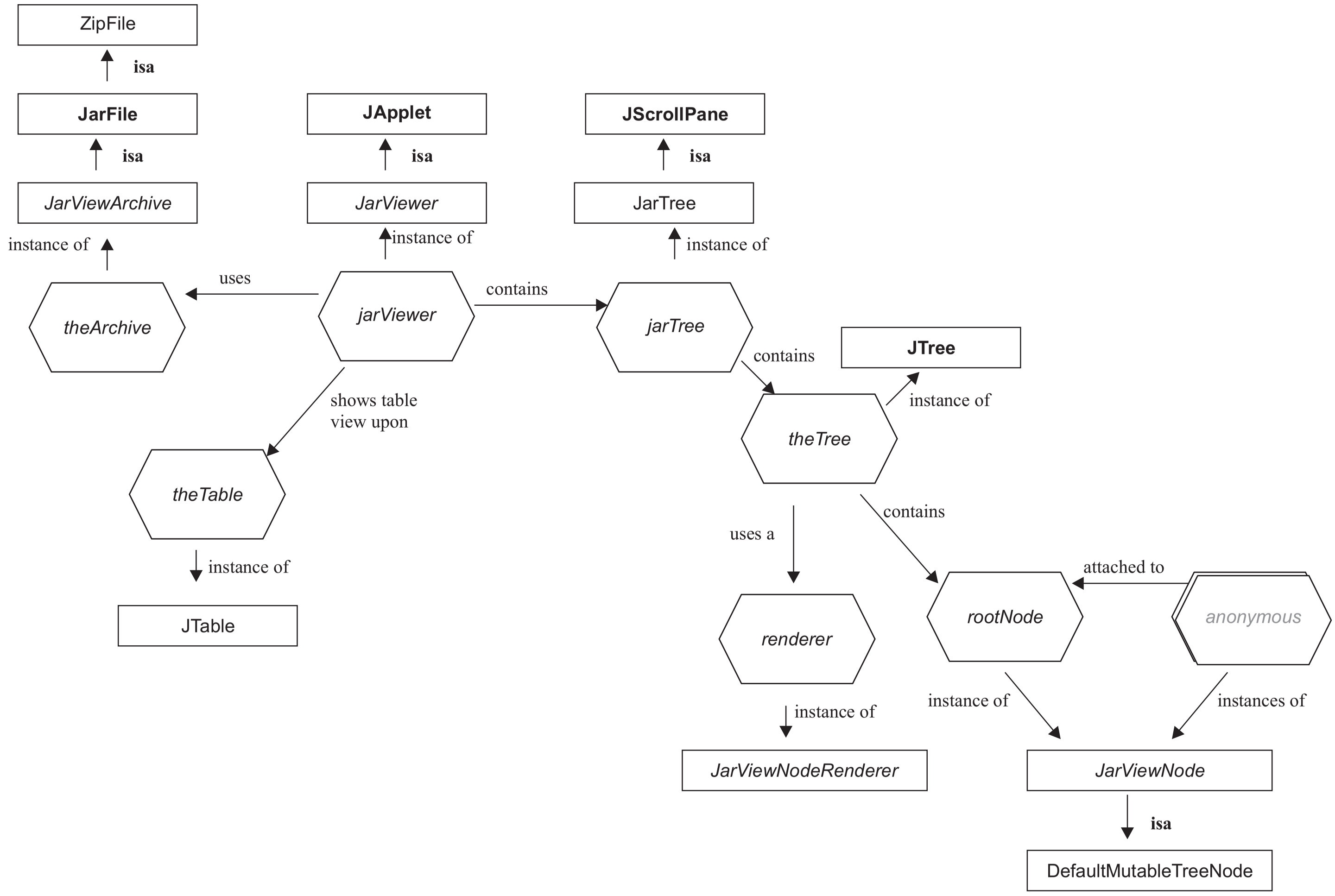
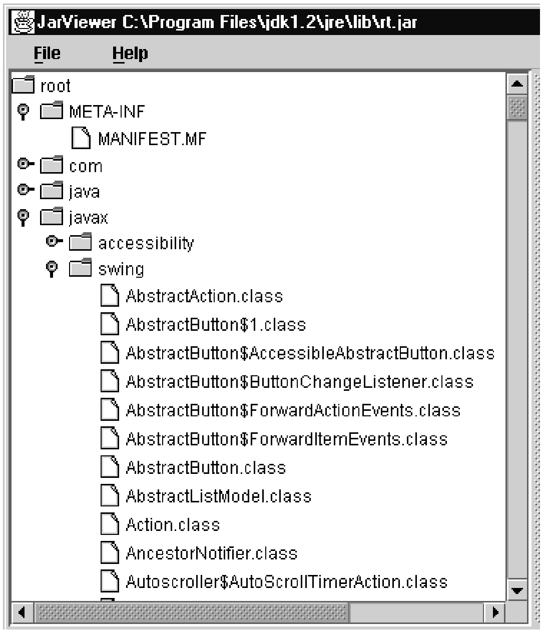


Figure 12.8 JarViewer: context diagram.



```
root
  META-INF
      MANIFEST.MF
  com
  java
  javax
      accessibility
      swing
          AbstractAction.class
          AbstractButton$1.class
          AbstractButton$AccessibleAbstractButton.class
          AbstractButton$ButtonChangeListener.class
          AbstractButton$ForwardActionEvents.class
          AbstractButton$ForwardItemEvents.class
          AbstractButton.class
          AbstractListModel.class
          Action.class
          AncestorNotifier.class
          Autoscroller$AutoScrollTimerAction.class
```

Figure 12.9 *JarViewer* without a specialized renderer.

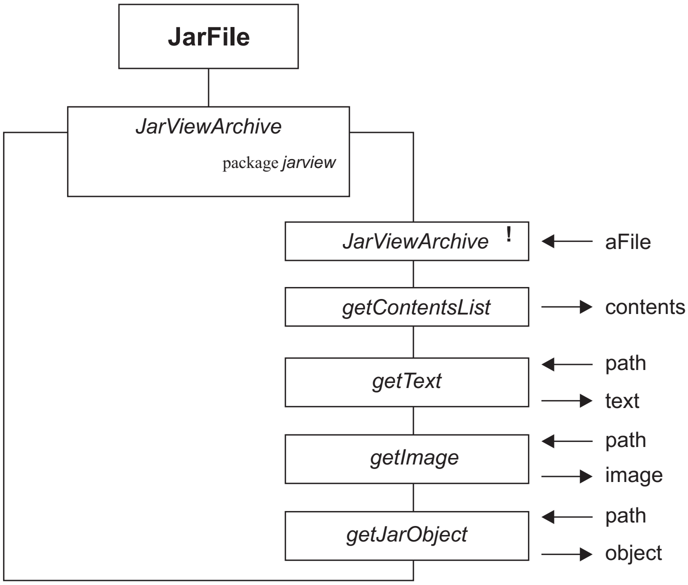


Figure 12.10 *JarViewArchive* class diagram.

Each element is identifying an entry in the archive and the *JarViewer* utility will obtain a contents list and use it to construct the tree to be explored. The *getText()* method takes as an argument a String *path*, assumed to be one of the entries from the *contentsList* and will return the contents of the entry as a String. The *getImage()* method will likewise require a *path*, assumed to identify a `*.gif` or a `*.jpeg` entry in the archive, and will return an ImageIcon instance containing the image. The last method, *getJarObject()*, also requires a path as an argument and will return an instance of the JarEntry class which contains the information shown in the class and tree views.

The implementation of the *JarViewArchive* class, as far as the end of the constructor, is as follows.

```
package jarview;

import javax.swing.*;
import java.io.*;
import java.util.jar.*;
import java.util.*;
import java.net.*;

public class JarViewArchive extends JarFile {

private boolean     contentsListObtained = false;
private Enumeration contentsList         = null;
private String      jarFileName          = null;


   public JarViewArchive( File jarFile)
                          throws IOException {
      super( jarFile);
      jarFileName = jarFile.getAbsolutePath();
   } // End JarViewArchive.
```

The *contentsList* attribute is cached within the class in case it is ever asked for more than once and the *contentsListObtained* flag will indicate whether it has already been obtained. The remaining attribute, *jarFileName* records the absolute filepath of the `*.jar` file.

The constructor commences by calling its **super**, JarFile, constructor. It is this step which ensures that the file exists on the file store, that it can be opened, that it is a jar file, that the structure of the file is undamaged and, if it is signed or certified, that the signatures and certificates are valid. If any of these conditions are not met an IOException will be thrown. The remaining step in the constructor is to use the File getAbsolutePath() method to obtain a String containing the filepath that identifies the file. The *getContentsList()* method is implemented as follows.

```
   public Enumeration getContentsList() {

   Vector      theEntries = null;
   Enumeration enum       = null;

      if ( !contentsListObtained) {
         theEntries = new Vector();
         enum       = this.entries();

         while ( enum.hasMoreElements()) {
            theEntries.add( enum.nextElement().toString());
         } // End while.
         contentsList = theEntries.elements();
```

```
0045          } // End if.
0046          return contentsList;
0047       } // End getContentsList.
```

If the *contentsListObtained* flag indicates that the list has not yet been asked for it is constructed on lines 0038 to 0045, which create the Enumeration, before it is returned on line 0046. To create the Enumeration of Strings in *contentsList* the JarFile entries() method is used, on line 0039, to obtain an Enumeration of JarEntry elements called *enum*. This Enumeration is then unpacked, in the body of the **while** loop, obtaining a String representation of each entry and storing them in a Vector called *theEntries*. Having obtained a Vector of Strings containing the paths of the entries in the JarFile, the Vector elements() method is used to supply this information as an Enumeration which is referenced by the instance attribute *contentsList*. The format of the contents of this list is as illustrated above. The implementation of the *getText()* method is as follows.

Object → java.util.zip.ZipFile

```
public ZipFile( File   file) throws IOException
public ZipFile( String name) throws IOException
```

Constructs a ZipFile instance to the File provided or named.

```
public void close()
public Enumeration entries()
public ZipEntry    getEntry( String path)
public InputStream getInputStream( ZipEntry entry)
```

Methods to obtain an Enumeration of all the entries or details of a particular entry identified by its *path*, or an InputStream to read the contents of the *entry*.

Object → java.util.zip.ZipEntry

```
public ZipEntry( String path)
```

Constructs an empty ZipEntry instance with the *path* named.

```
public long   size()
public long   getCompressedSize()
public String name()
public long   getTime()
```

Methods to obtain the four most significant attributes of the entry.

```
0052       public String getText( String path) {
0053
0054       InputStream    theStream = null;
0055       BufferedReader readFrom  = null;
0056       String         aLine     = null;
0057       StringBuffer   theText   = null;
0058       boolean        endOfFile = false;
0059
0060          theText = new StringBuffer( "");
0061          try {
```

```
0062           theStream = this.getInputStream(
0063                                      this.getEntry( path));
0064           readFrom = new BufferedReader(
0065                          new InputStreamReader( theStream));
0066
0067           endOfFile = false;
0068           while ( !endOfFile) {
0069              aLine = readFrom.readLine();
0070              if ( aLine == null) {
0071                 endOfFile = true;
0072              } else {
0073                 theText.append( aLine + "\n");
0074              } // End if.
0075           } // End while.
0076           theStream.close();
0077        } catch ( IOException exception) {
0078          // Do nothing!
0079        } // End try/catch
0080        return theText.toString();
0081     } // End getText.
```

The method commences, on lines 0062 to 0065, after initializing a StringBuffer to an empty string, by constructing a BufferedReader stream which is connected to the entry in the JarFile identified by the *path* passed as an argument. This is accomplished in two steps, first obtaining an InputStream to the entry and then constructing a BufferedReader from it. To construct the InputStream the *path* is first used to obtain the JarEntry object associated with it. In a sense this is the reverse of toString() used on line 0042 of the *getContentsList()* method. The toString() method converted a JarEntry object into a String, whereas the getEntry() method converts a String into a JarEntry. Having obtained the entry for the *path* its getInputStream() method is used to obtain a stream which can be used to read the contents of the entry.

Object → ZipFile →java.util.jar.JarFile

```
public JarFile( File   file) throws IOException
public JarFile( String name) throws IOException
```

Constructors overriding the corresponding ZipFile constructors.

```
public Manifest getManifest()
public JarEntry getJarEntry( String path)
```

Every JarFile must contain a manifest which is represented by the java.util.jar.Manifest class. The entries in a JarFile are instances of the JarEntry class which extends the ZipEntry class.

Object → ZipEntry → java.util.jar.JarEntry

```
public JarEntry( String path)
```

Constructs an empty JarEntry instance with the *path* named.

```
public Certificates[] getCertificates()
public Attributes[]   getAttributes
```

Methods to obtain the two additional attributes of a JarEntry.

Having obtained a BufferedReader connected to the contents of the entry the remaining parts of the method are fairly standard, transferring the contents of the stream, line by line, to the StringBuffer. If an exception is thrown during this process the StringBuffer is left unchanged and the method will return an empty string, the partial contents of the entry or the entire contents of the entry.

The implementation of the *getImage()* method is as follows.

```
0085     public ImageIcon getImage( String path) {
0086
0087     String    jarURL   = null;
0088     URL       location = null;
0089     ImageIcon theIcon  = null;
0090
0091        jarURL = new String( "jar:file:" + jarFileName +
0092                             "!/" + path);
0093        try {
0094           location = new URL( jarURL);
0095        } catch ( MalformedURLException e) {
0096           // do nothing.
0097        } // End try/catch.
0098        theIcon = new ImageIcon( location, path);
0099        return theIcon;
0100     } // End getImage.
```

The most obvious technique for obtaining an image from a JarFile would be to open a Stream connection to the *gif* or *jpeg* entry and read the contents, constructing and returning an Image. However, there is no simple capability for this to be done within the standard Java classes. Instead, an ImageIcon is constructed from a URL subclass called JarURL and returned. A JarUrl starts with `jar:` is followed by a standard URL to identify the jar file and is followed by an exclamation mark (**!**)and the path identifying the entry within the jar archive. For example, the *JarURL* for the `Question.gif` image shown in Figure 12.5 would be the single string.

```
jar:file:c:/Program Files/jdk1.2/jre/lib/rt.jar!
javax/swing/plaf/metal/icons/Question.gif
```

Within the *getImage()* method the *jarUrl* String is constructed on lines 0091 to 0092 and a JarURL instance called *location* constructed from it on line 0094. When the URL is available an ImageIcon, called *theIcon*, can be constructed from it and returned on line 0099. The construction of an ImageIcon is a synchronous method which will not return until the image has, or has not, been loaded and the ImageIcon containing the image constructed. This means that media tracking is not required and if the image cannot be obtained the second argument to the constructor, the *path* argument, is rendered instead of the image. The implementation of the final method, *getJarObject()*, is as follows.

```
   public JarEntry getJarObject( String path) {
      return this.getJarEntry( path);
   } // End getJarObject

} // End JarViewArchive.
```

This is a wrapper method onto the JarFile getJarEntry() method. The nature of a JarEntry instance and the use made of it will be described, when the call of this method is used in the description of the *JarViewer* class below.

12.4 The *JarTree* class

A *JarTree* instance is used for the entire contents of the left-hand side of the split pane. Visibly it consists of a JScrollPane containing a JTree within its viewport. The class diagram for the *JarTree* class is given in Figure 12.11; the simplicity of its design belies its internal complexity as suggested by the three contained methods.

The constructor creates an empty JScrollPane and registers the identity of the TreeSelectionListener passed to it as an argument. A TreeSelectionEvent will be sent to the listener every time the user changes the selected node of the tree. The tree itself is constructed and installed into the scroll pane whenever the *buildTree()* method is called; the *contents* argument are a String Enumeration of the paths of the entries in the JarFile, as obtained from the *JarViewArchive getContentsList()* method. The three private methods are all used in support of this method. The remaining method, *getNodeForPath()*, will return the identity of the node indicated by its *path* argument.

The implementation of this class, as far as the end of its constructor, is as follows.

```
package jarview;


import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import java.util.*;

```

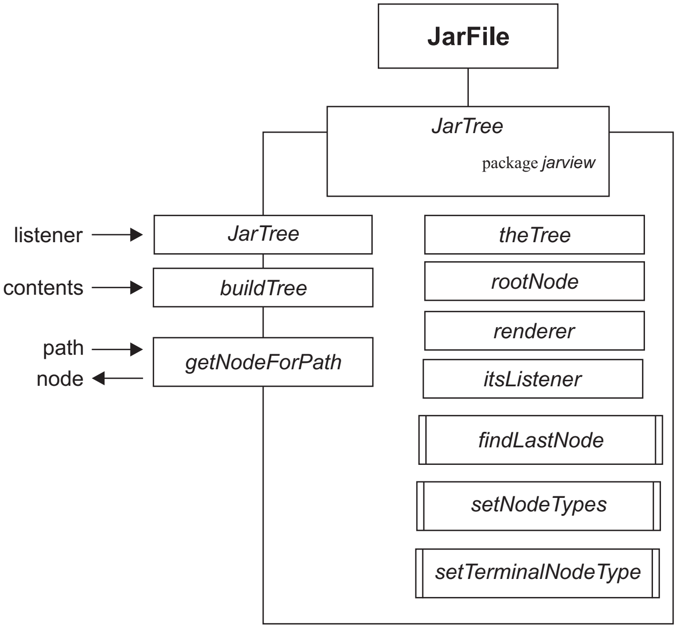


Figure 12.11 *JarTree* class diagram.

```
0018  public class JarTree extends JScrollPane {
0019
0020  private JarViewNode           rootNode    = null;
0021  private JTree                 theTree     = null;
0022  private TreeCellRenderer      renderer    = null;
0023  private TreeSelectionListener itsListener = null;
0024
0025
0026     protected JarTree( TreeSelectionListener listener) {
0027        super( JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
0028               JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
0029        renderer    = new JarViewNodeRenderer();
0030        itsListener = listener;
0031     } // End JarTree constructor.
```

The *rootNode* of *theTree*, declared on lines 0020 and 0021, is an instance of the *JarViewNode* class which inherits from the *DefaultMutableTreeNode* class the resources needed to support child nodes. The *renderer* will be used to paint the nodes in the tree and *itsListener* completes the declaration of the attributes. The constructor commences by calling its **super**, JScrollPane, constructor, indicating that both scroll bars should always be visible. The remaining steps of the constructor create the *renderer* and record the *listener* argument.

JComponent →javax.swing.JTree

```
public JTree( TreeNode rootNode)
```

Constructs a JTree containing the hierarchy of nodes attached to the *rootNode*. (There are many other ways of constructing a tree and supplying its contents.)

```
public void addTreeExpansionListener( TreeExpansionListener listener)
public void addTreeSelectionListener( TreeSelectionListener listener)
```

Methods to register *listener*s for events that are fired when a node is expanded or collapsed and when a node is selected.

```
public void collapsePath( TreePath path)
public void expandPath( TreePath path)
public boolean  isCollapsed( TreePath path)
public boolean  isExpanded(  TreePath path)
public TreePath getSelectionPath()
public boolean  isVisible(   TreePath path)
public void     makeVisible( TreePath path)
public int      getVisibleRowCount()
public void     setVisibleRowCount( int newCount)
public void     setCellRenderer( TreeCellRenderer renderer)
public void     setShowsRootHandles( boolean yesOrNo)
```

Various methods to inquire about or maintain the tree. A TreePath expresses the path between the root node and the node of interest, and are dispatched as a part of tree events. The cellRenderer is used to paint the nodes and the rootHandles are the parts of the visual representation that indicate if a node is expanded or collapsed.

The first part of the implementation of the *buildTree()* method is as follows.

```
protected void buildTree( Enumeration contents) {

String thisNodeName   = null;
String lastTerm       = null;
String thePath        = null;

JarViewNode storeHere = null;
JarViewNode newNode   = null;
int lastSlash = -1;

   rootNode = new JarViewNode( "root");
   rootNode.setNodeType( JarViewNode.ROOT_NODE);
   theTree = new JTree( rootNode);
   theTree.setCellRenderer( renderer);
   theTree.addTreeSelectionListener( itsListener);
```

Object →javax.swing.tree.DefaultMutableTreeNode implements MutableTreeNode

```
public DefaultMutableTreeNode()
public DefaultMutableTreeNode( Object userObject)
```

Constructs a DefaultMutableTreeNode; an arbitary userObject attribute can be associated with each node in a tree. The MutableTreeNode interface extends the TreeNode interface and allows instances to be added to a JTree.

```
public void add( MutableTreeNode newChild)
public int  getChildCount()
public void removeAllChildren()
public void remove( MutableTreeNode thisChild)
public Enumeration children()
```

Methods to maintain the subtree attached to this node by adding, removing and inquiring about its child nodes.

```
public Enumeration breadthFirstEnumeration()
public Enumeration depthFirstEnumeration()
```

Methods to obtain an Enumeration of all the nodes attached, directly or indirectly, to this node.

```
public int getLevel()
public int getDepth()
public boolean  isRoot()
public boolean  isLeaf()
public TreeNode getParent()
public TreeNode getRoot()
```

The level attribute refers to the position from the root node, depth counts the maximum span of nodes to a leaf node.

```
public Object getUserObject()
public void   setUserObject( Object userObject)
```

Getter and setter for the userObject attribute.

This part of the method constructs and configures a new *rootNode*, named *root*, and a new JTree which are associated together, and at the end of the method will be added to **this** JScrollPane so as to become visible to the user. The *rootNode* is an instance of the JarViewNode class and is marked as a *ROOT_NODE* by having its *setNodeType()* method called after construction. The new JTree, *theTree*, has the *renderer* and *itsListener* attributes registered with it. The method continues and concludes as follows.

```
0050         while ( contents.hasMoreElements()) {
0051            thisNodeName = (String) contents.nextElement();
0052 
0053            lastSlash = thisNodeName.lastIndexOf('/');
0054            lastTerm  = thisNodeName.substring( lastSlash+1);
0055            thePath   = thisNodeName.substring( 0, lastSlash);
0056 
0057            if ( lastTerm.length() > 0) {
0058               storeHere = this.findLastNode( thePath);
0059               newNode   = new JarViewNode(   lastTerm);
0060               storeHere.add( newNode);
0061               this.setTerminalNodeType( newNode);
0062            } // End if.
0063         } // End while.
0064 
0065         this.setNodeTypes();
0066         theTree.expandPath( new TreePath( rootNode));
0067         this.getViewport().add( theTree);
0068      } // End buildTree
```

The **while** loop, commencing on line 0050, will retrieve each String from the *contents* argument, storing it in *thisNodeName*, and install them into the tree by adding it, directly or indirectly, to the *rootNode*. The first part of the loop body, on lines 0053 to 0056, splits the path in *thisNodeName* into the part which precedes the last slash (/) and that which follows it, which may be empty. For example the path to the `Question.gif` entry in the `rt.jar` file, as shown in Figure 12.5, is `javax/swing/plaf/metal/icons/Question.gif`, which would be split into `javax/swing/plaf/metal/icons` and `Question.gif`. When this is added to the tree a node named `Question.gif` will have to be added to the node at the end of the path commencing with `javax`, constructing any parts of the path that do not yet exist.

This is accomplished on line 0058, which calls the *findLastNode()* method to obtain the last node in *thePath*, constructing any of its parts which do not already exist. Following this, lines 0059 and 0060 construct the *newNode* and add it to the last node in the path, *storeHere*. Line 0061 configures the *newNode* by setting its *nodeType* attribute so that the *renderer* will be able to decide how to paint it.

When the **while** loop concludes, on line 0063, all of the paths contained in the *contents* argument will have been put through this process, resulting in a network of nodes that reflect the structure of the entries in the JarFile. The final steps in the *buildTree()* method call the *setNodeTypes()* method to set the *nodeType*s of all non-terminal nodes and then expand the *rootNode*, so that its child nodes are visible, before adding the *rootNode* to this JScrollPane's viewport.

The first of the **private** methods called from the *buildTree()* method is *findLastNode()*, which takes as an argument a path *toTrace* and which should return the identity of the node at the end of the path, constructing any parts which do not already exist. Its implementation commences as follows.

```
0072     private JarViewNode findLastNode( String toTrace) {
0073
0074     StringTokenizer tokenizer     = null;
0075     String          nextPart      = null;
0076     String          thisNodesName = null;
0077
0078     JarViewNode thisNode = null;
0079     JarViewNode nextNode = null;
0080
0081     Enumeration thisNodesChildren = null;
0082     boolean     pathExists        = false;
0083
0084        tokenizer = new StringTokenizer( toTrace, "/");
0085        thisNode = rootNode;
0086        while ( tokenizer.hasMoreTokens()) {
```

This part of the method prepares for tracing a path through the nodes commencing with the *rootNode*. The path *toTrace* is used to construct a *tokenizer* which will chop the String on its slashes. Each token, taken in turn within the **while** loop, will identify one step in the path with the local variable *thisNode* indicating the node which must contain a node with the same name as the token, constructing it if required. The method continues as follows.

```
0087           nextPart = tokenizer.nextToken();
0088           thisNodesChildren = thisNode.children();
0089           pathExists        = false;
0090           while( thisNodesChildren.hasMoreElements() &&
0091                  ! pathExists                        ){
0092              nextNode = (JarViewNode)
0093                         thisNodesChildren.nextElement();
0094              thisNodesName = nextNode.getName();
0095              if ( nextPart.equals( thisNodesName)) {
0096                 pathExists = true;
0097              } // End if.
0098           } // End while.
```

The next term in the path, *nextPart*, is obtained on line 0087 and an Enumeration of the children of *thisNode* is obtained on line 0088. The flag *pathExists*, set **false** on line 0089, will be reset **true** if *thisNode* already contains a node with the same name as *nextPart*. The **while** loop, commencing on line 0090, will consider each of the children of the current node terminating when all children have been considered or when a node is located which has the same name as *nextPart*. This is accomplished by retrieving the *nextNode* from the Enumeration, on lines 0092 to 0093, and then testing its *name* attribute on line 0095. At the end of this fragment if *pathExists* is **false** a *newNode* needs to be added to *thisNode*; or if *pathExists* is **true** then *nextNode* indicates the identity of the next step in the path to be traced. The final steps of the method are implemented as follows.

```
0101           if ( ! pathExists) {
0102              nextNode = new JarViewNode( nextPart);
0103              thisNode.add( nextNode);
0104           } // End if.
```

```
0105            thisNode = nextNode;
0106         } // End while.
0107         return thisNode;
0108      } // End findLastNode.
```

If the path does not exist lines 0102 and 0103 construct a new *JarViewNode* with the name *nextPart* and add it to *thisNode*. The effect is that *thisNode* will now definitely contain a suitable child node, and its identity is contained in *nextNode* which, on line 0105, becomes *thisNode* for when the *nextPart* of the path is considered. When all of the parts of the path have been considered *thisNode* will indicate the terminal node of the path and is returned from the method, on line 0107.

The second **private** method is *setTerminalNodeType()*, which is called every time a terminal node is added to the tree and which has the responsibility for setting the *nodeType* attribute of the new node. It is implemented as follows.

```
0110      private void setTerminalNodeType( JarViewNode toType) {
0111
0112      String nodeName  = toType.getName();
0113      String extension = null;
0114      int    lastDot   = -1;
0115
0116         if ( nodeName.lastIndexOf( "$") > -1) {
0117            toType.setNodeType( JarViewNode.CONTAINED_NODE);
0118         } else {
0119            lastDot   = nodeName.lastIndexOf('.');
0120            extension = nodeName.substring(
0121                                          lastDot+1).toLowerCase();
0122            if ( extension.equals( "java") ) {
0123               toType.setNodeType( JarViewNode.JAVA_NODE);
0124            } else if ( extension.equals( "class")) {
0125               toType.setNodeType( JarViewNode.CLASS_NODE);
0126            } else if ( extension.equals( "gif")) {
0127               toType.setNodeType( JarViewNode.GIF_NODE);
0128            } else if ( extension.equals( "jpeg")) {
0129               toType.setNodeType( JarViewNode.JPEG_NODE);
0130            } else if ( extension.equals( "txt")) {
0131               toType.setNodeType( JarViewNode.TEXT_NODE);
0132            } else if ( extension.startsWith( "htm")) {
0133               toType.setNodeType( JarViewNode.HTML_NODE);
0134            } else if ( extension.equals( "mf")) {
0135               toType.setNodeType( JarViewNode.MANIFEST_NODE);
0136            } else {
0137               toType.setNodeType( JarViewNode.OTHER_NODE);
0138            } // End if.
0139         } // End if.
0140      } // End setTerminalNodeType.
```

The method is relatively straightforward, making use of the *JarViewNode setNodeType()* method, using as arguments the manifest values supplied by the *JarViewNode* class. The substantive part of the method is a sequential selection which commences, on lines 0116 to 0117, by looking for a dollar ($) character in the node's *name*, which indicates that it is a contained class and sets the *nodeType* as appropriate. Otherwise the extension part of the entry is obtained and each branch of the selection identifies and sets a specific type with the last, default, branch setting the type to *OTHER_NODE*.

The third **private** method is *setNodeTypes()* which is called when the tree has been completely constructed and has the responsibility of setting the *nodeType* attribute of all untyped nodes. These will be the path nodes and the package nodes, which can be distinguished as a path node which will have no leaf node children. It is implemented as follows.

```
0144     private void setNodeTypes() {
0145
0146     Enumeration allNodes          = null;
0147     JarViewNode thisNode          = null;
0148     Enumeration thisNodesChildren = null;
0149     JarViewNode thisChild         = null;
0150     boolean     leafFound         = false;
0151
0152        allNodes = rootNode.depthFirstEnumeration();
0153        while ( allNodes.hasMoreElements()) {
0154           thisNode = (JarViewNode) allNodes.nextElement();
0155           if ( thisNode.getNodeType() == JarViewNode.UNKNOWN) ){
0156              thisNodesChildren = thisNode.children();
0157              leafFound = false;
0158              while ( thisNodesChildren.hasMoreElements() &&
0159                      !leafFound                          ){
0160                 thisChild = (JarViewNode)
0161                                  thisNodesChildren.nextElement();
0162                 if ( thisChild.isLeaf()) {
0163                    leafFound = true;
0164                 } // End if.
0165              } // End while.
0166              if ( leafFound) {
0167                 thisNode.setNodeType( JarViewNode.PACKAGE_NODE);
0168              } else {
0169                 thisNode.setNodeType( JarViewNode.PATH_NODE);
0170              } // End if.
0171           } // End if.
0172        } // end while.
0173     } // End setNodeTypes.
```

Again the method is relatively straightforward, commencing, on line 0152, by using the *rootNode*'s depthFirstEnumeration() method to obtain an Enumeration that contains the identities of all the nodes in the tree. Each node can then be considered in turn and if its node type is still the default *UNKNOWN* value, it will have to have its type established. This is accomplished on lines 0156 to 0165 where an Enumeration of all the children of *thisNode* is obtained and checked to see if any of its children are leaf nodes. The final step, on lines 0166 to 0171, sets the node type to *PACKAGE_NODE* or *PATH_NODE* as appropriate.

The only remaining method in the JarTree class is *getNodeForPath()*, which takes a TreePath instance as an argument and is a wrapper onto its getLastPathComponent() method. It is implemented as follows.

```
0176     protected JarViewNode getNodeForPath( TreePath thePath ) {
    return (JarViewNode) thePath.getLastPathComponent();
  } // End getNodeForPath.

} // End class JarTree
```

12.5 The *JarViewNode* and *JarViewNodeRenderer* classes

The *JarViewNode* class extends the DefaultMutableTreeNode class and supplies the nodes which form the tree, as described above. Its class diagram is given in Figure 12.12. It adds two attributes, a *name* for the node which must be specified as a part of the constructor and a *nodeType* which is supported by the *NODE_TYPES* manifest values. The implementation of this class is presented, without comment, as follows.

```
package jarview;

import javax.swing.*;
import javax.swing.tree.*;
import java.util.*;

public class JarViewNode extends DefaultMutableTreeNode {

protected static final int UNKNOWN        = 0;
protected static final int ROOT_NODE      = 1;
protected static final int PATH_NODE      = 2;
protected static final int PACKAGE_NODE   = 3;
protected static final int MANIFEST_NODE  = 4;
protected static final int JAVA_NODE      = 5;
protected static final int CLASS_NODE     = 6;
protected static final int HTML_NODE      = 7;
protected static final int GIF_NODE       = 8;
protected static final int JPEG_NODE      = 9;
protected static final int TEXT_NODE      = 10;
```

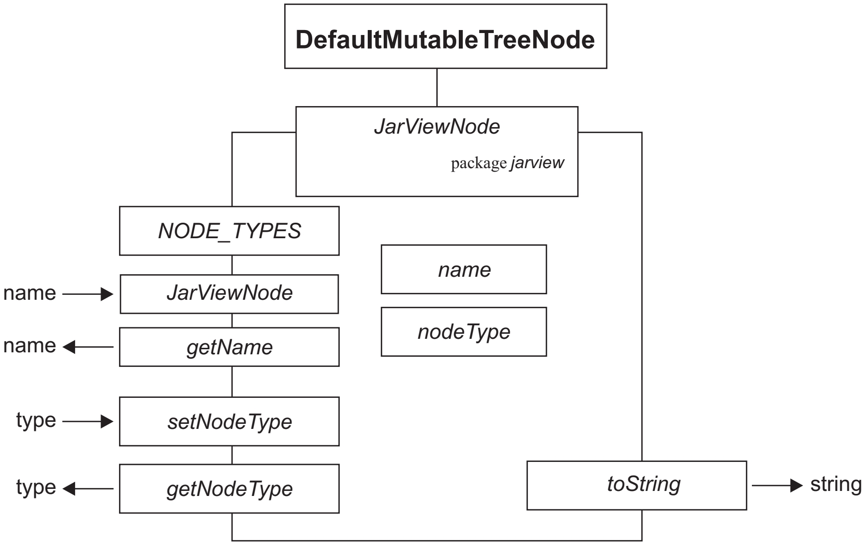


Figure 12.12 *JarViewNode* class diagram.

```
protected static final int OTHER_NODE     = 11;
protected static final int CONTAINED_NODE = 12;
protected static final int NUMBER_OF_NODE_TYPES = 13;

private String itsName  = null;
private int    nodeType = UNKNOWN;

   protected JarViewNode( String name) {
      itsName = new String( name);
   } // End JarViewNode constructor.

   protected String getName() {
      return itsName;
   } // End getName.

   protected void setNodeType( int type) {
      nodeType = type;
   } // End setNodeType.

   protected int getNodeType() {
      return nodeType;
   } // End setNodeType.

   public String toString() {
      return this.getName();
   } // End toString.

} // End JarViewNode.
```

The *JarViewNodeRenderer* class is provided in order to supply a view of the *JarViewNode* instances of the nodes in the JTree, it extends the DefaultTreeCellRenderer class which itself extends the JLabel class. The *JarViewNodeRenderer* class has only to override the inherited getTreeCellRendererComponent() method. This method will be called every time the tree is about to repaint the node and is passed various arguments including the node to be rendered and details of its state. The overridden method should set the icon and text attributes, inherited from JLabel, which will determine how the image of the node will appear.

A substantive part of the implementation of the class is concerned with preparing the ImageIcon instances which will be installed as the icon resource of the JLabel. The first part of the implementation of the class is as follows.

```
package jarview;

import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;
import java.util.*;
import java.net.URL;

public class JarViewNodeRenderer
                  extends DefaultTreeCellRenderer {

```

```
private static Icon[]   nodeIcons  = null;
private static String[] imageFileNames =  {
              "unknown.gif", "root.gif", "path.gif",
          "package.gif", "manifest.gif", "java.gif",
                "class.gif", "html.gif",  "gif.gif",
               "jpeg.gif",  "text.gif", "other.gif",
                                    "contained.gif"};
private static boolean iconsPrepared = false;


    public JarViewNodeRenderer () {
       super();
       if ( ! iconsPrepared) {
          this.prepareIcons();
          iconsPrepared = true;
       } // End if
    } // End JarViewNodeRenderer .
```

The three attributes are all class-wide and consist of an array of ImageIcon, the filenames which contain the images to be used, and a flag that indicates if the icons have already been prepared. The constructor commences by calling the **super**, DefaultTreeCellRenderer, constructor before calling a **private** method called *prepareIcons()* if the *iconsPrepared* flag indicates that they have not already been prepared. The *prepareIcons()* method is implemented as follows.

```
   private void prepareIcons() {

   Image[]      nodeImages = null;
   MediaTracker tracker    = new MediaTracker( this);
   Class        classDescriptor  = this.getClass();
   URL          resourceLocation = null;

      if ( imageFileNames.length <
           JarViewNode.NUMBER_OF_NODE_TYPES) {
         System.err.println( "JarViewNodeRenderer  ... " +
                "not enough icon images ... abending!");
         System.exit( -1);
      } // End if.

      nodeImages = new Image[ imageFileNames.length];
      nodeIcons  = new Icon[  imageFileNames.length];

      for( int index =0;
               index < imageFileNames.length;
               index++) {
         resourceLocation = classDescriptor.getResource(
                        "icons/" + imageFileNames[ index]);
         nodeImages[ index] = Toolkit.getDefaultToolkit().
                                getImage( resourceLocation);
         tracker.addImage( nodeImages[ index], index);
      } // End for.

```

```
0067        try {
0068           tracker.waitForAll();
0069        } catch ( InterruptedException exception) {
0070           // Do nothing.
0071        } // End try/catch.
0072
0073        if ( tracker.isErrorAny()) {
0074           System.err.println( "JarViewNodeRenderer  ... " +
0075                 " icon image not found ... abending!");
0076           System.exit( -1);
0077        } // End if.
0078
0079        for( int index =0;
0080                 index < imageFileNames.length;
0081                 index++) {
0082          nodeIcons[ index] = new ImageIcon( nodeImages[ index]);
0083        } // End for.
0084     } // End prepareIcons.
```

The method is implemented using a *MediaTracker* instance to synchronize the loading of the 13 images that are stored in the *icons* subdirectory of the location where the *JarViewNodeRenderer* class was obtained from. These techniques were fully explained in Chapter 7 when the *TimeInput mevening* icons were loaded as class resources. The method will abend if the icons cannot be prepared; otherwise after it has concluded the *iconsPrepared* flag will be true and the *nodeIcons* array will contain an ImageIcon for each of the *JarViewNode* manifest values.

The implementation of the overriding *getTreeCellRendererComponent()* method is as follows.

```
0087     public Component getTreeCellRendererComponent( JTree   tree,
0088                                                   Object  value,
0089                                                   boolean isSelected,
0090                                                   boolean expanded,
0091                                                   boolean leaf,
0092                                                   int     row,
0093                                                   boolean hasFocus) {
0094     JarViewNode node       = null;
0095     int         nodeType   = -1;
0096
0097        node = (JarViewNode) value;
0098        this.setText( node.getName());
0099        nodeType = node.getNodeType();
0100        this.setIcon( nodeIcons[ nodeType]);
0101        super.selected = isSelected;
0102        return this;
0103     } // End getTreeCellRendererComponent.
0104
0105  } // End JarViewNodeRenderer.
```

The *value* argument contains the identity of the node which is to be presented and the *isSelected* argument indicates the selection state of the node. The remaining arguments are not used by this method. It commences, on line 0097, by casting the *value* argument into its known *JarViewNode* type. Having done this the *name* attribute of the node is retrieved and installed as the *text* resource, following which the *nodeType* is obtained and the appropriate icon from the *nodeIcons* array installed as the icon resource. Before **this** node renderer is returned from the method the *selected* attribute provided by the DefaultTreeCellRenderer class is set directly, as no state setting method is supplied for it, from the *isSelected* argument.

An instance of the *JarViewNodeRenderer* class is constructed by the *JarTree* class and associated with the JTree instance in its *buildTree()* method, as explained above. Its *getTreeCellRendererComponent()* method will be called immediately before each node is rendered and the paint() method of the Component returned by it will be used to actually depict the state of the node. The effects of the method can be seen in Figure 12.1, where the icon and text resources of the JLabel representing each node are clearly visible.

12.6 The *JarViewer* class

The *JarViewer* class extends the JApplet class and implements the ActionListener, MenuListener and TreeSelectionListener interfaces. Its implementation as far as the start of its *init()* method is as follows.

```
package jarview;

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import javax.swing.table.*;
import java.util.*;
import java.util.jar.*;


public class JarViewer extends JApplet
                    implements ActionListener,
                               MenuListener,
                       TreeSelectionListener {

JarViewMenuBar    menuBar     = null;
JarViewArchive    theArchive  = null;
JarTree           jarTree     = null;

JSplitPane  jarPane    = null;
JScrollPane detailPane = null;
JFrame      titleFrame = null;
```

The *JarViewMenuBar* class supplies the application-level main menu, using techniques that were first explained in Chapter 6, and will not be presented in this chapter. The *JarViewer* class must implement the MenuListener interface in order to listen to the events generated when the menus are posted or unposted, and the ActionListener interface for the events fired when the user activates one of the menu items. The instance diagram in Figure 12.8 shows that the *jarViewer* uses an instance of the *JarViewArchive* class called *theArchive*, which is declared on line 0029. It also contains an instance of the *JarTree* class that is declared on line 0030.

The applet contains a JSplitPane, called *jarPane*, in its main work area, the right-hand side of which contains a JScrollPane, called *detailPane*, within which the various views will be shown. The only other attribute declaration is the identity of the JFrame instance that the artifact may be mounted within, declared on line 0034. The *init()* method is implemented as follows.

```
0037     public void init() {
0038
0039        menuBar = new JarViewMenuBar( this, this);
0040
0041        jarPane = new JSplitPane( JSplitPane.HORIZONTAL_SPLIT);
0042
0043        theTree = new JarTree( this);
0044        jarPane.setLeftComponent( theTree);
0045
0046        detailPane = new JScrollPane(
0047                          JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
0048                          JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
0049        jarPane.setRightComponent( detailPane);
0050
0051        jarPane.setDividerLocation( 0.0);
0052
0053        this.setJMenuBar( menuBar);
0054        this.getContentPane().add( jarPane, BorderLayout.CENTER);
0055     } // End init.
```

The *init()* method constructs, configures and assembles the various components in the required manner. However, as at this stage there is no JTree component within *theTree* and nothing within the *detailPane*, when the artifact first becomes visible to the user both of these areas will be empty. The user can only exit from the artifact or activate the *File* menu *Open Jar...* option, which will, via the *actionPerformed()* method, cause the **private** *loadJar()* method, which commences as follows, to be called.

```
0267     private void loadJar() {
0268
0269     JFileChooser        chooser     = null;
0270     ExtensionFileFilter filter      = null;
0271     int                 userAction  = 0;
0272     File                fileChosen  = null;
0273     JarViewArchive      newArchive  = null;
0274     Enumeration         contents    = null;
0275
0276        chooser = new JFileChooser();
0277        filter  = new ExtensionFileFilter();
0278        filter.addExtension(   "jar");
0279        filter.setDescription( "JAR Archives");
0280        chooser.setFileFilter( filter);
0281        userAction = chooser.showOpenDialog( this);
0282
0283        if ( userAction == JFileChooser.APPROVE_OPTION) {
```

```
0284           fileChosen = chooser.getSelectedFile();
```

This part of the method is essentially identical to the first part of the *openFile()* method in Chapter 9. It constructs and configures an instance of the JFileChooser class, in this example restricted to displaying `*.jar` files, and then posts it to the user. If the user selects a file the call of getSelectedFile(), on line 0284, will return the File identity of the selection. The method continues as follows.

```
0286           try {
0287               newArchive = new JarViewArchive( fileChosen);
0288               theArchive = newArchive;
0289               contents = theArchive.getContentsList();
0290               theTree.buildTree( contents);
0291               detailPane.getViewport().add( new JPanel());
0292               detailPane.setColumnHeader( new JViewport());
0293               if( titleFrame != null) {
0294                  titleFrame.setTitle( "JarViewer " + fileChosen);
0295               } // End if.
0296               jarPane.setDividerLocation( 0.5);
0297               this.repaintAll( this.getGraphics());
0298           } catch ( IOException exception) {
0299                JOptionPane.showMessageDialog( this,
0300                                          "The Archive \n" + fileName +
0301                                          "\ncould not be opened!",
0302                                          "File Open Failure",
0303                                         JOptionPane.ERROR_MESSAGE);
0304           } // End try/catch.
0305        } // End if.
0306     } // End loadJar.
```

On line 0287 the File instance, *fileChosen*, is passed as an argument to the *JarViewArchive* constructor. This, as explained above, will attempt to construct an instance from the jar file identified by *fileChosen*, throwing an IOException if this is not possible. If the archive is successfully constructed, on line 0288, the instance attribute *theArchive* is made to reference it.

Having successfully constructed *theArchive*, its *getContentsList()* method is called to obtain details of its entries, and these are immediately passed as an argument to *theTree*'s *buildTree()* method. The effect, as explained above, is to create a network of nodes within *theTree* which reflects the structure of the entries in *theArchive*. On lines 0291 and 0292 any existing contents of the *detailPane* are removed by installing an empty panel into its viewport and an empty viewport into its columnHeader. The columnHeader attribute of a JScrollPane is related to its use when a JTable is installed into its viewport and will be explained in the next section.

The method continues by setting the title attribute of the JFrame *titleFrame* instance, whose identity would have been established by the omitted *addNotify()* method and which will be **null** if the artifact is executing within a browser. The effect of this step can be seen in Figure 12.1. The newly constructed tree is then made visible to the user, on line 0297, by ensuring that all components are repainted with a call of repaintAll(). The final part of the method, on lines 0299 to 0305, will only be executed if the *JarViewArchive* constructor throws an IOException and posts a JOptionPane error dialog to inform the user of this.

EventListener → javax.swing.event.TreeSelectionListener interface

```
public void valueChanged( TreeSelectionEvent event)
```

Method to be called when the user selects a new node in a tree.

Object → EventObject → javax.swing.event.TreeSelectionEvent

```
public TreeSelectionEvent( Object   source,
                           TreePath path,
                           boolean  isNew,
                           TreePath oldLeadSelectionPath,
                           TreePath newLeadSelectionPath)
```

The source attribute is the JTree instance that fired the event and path is the route from its root to the newly selected node. The isNew, oldLeadSelectionPath and newLeadSelectionPath are used for multiple selections.

```
public TreePath getPath()
```

Returns the path of the selected item.

The JarView applet is registered with the *JarTree* as its TreeSelectionListener, in order that it can be informed when the user changes the selected item in the tree. This requires it to implement the TreeSelectionListener interface which mandates a *valueChanged()* method, whose implementation commences as follows.

```
public void valueChanged( TreeSelectionEvent event) {

TreePath    selectionPath   = null;
String      pathName        = null;
JarViewNode selectedNode    = null;
Object      selectedObject  = null;
ImageIcon   imageObject     = null;
String      textObject      = null;
JarEntry    jarObject       = null;
String      jarString       = null;
boolean     objectAvailable = false;
int         nodeType        = -1;

   detailPane.getViewport().add( new JPanel());
   detailPane.setColumnHeader( new JViewport());

   selectionPath   = event.getPath();
   pathName        = this.makePathName( selectionPath);
   selectedNode    = theTree.getNodeForPath( selectionPath);
   nodeType        = selectedNode.getNodeType();
   objectAvailable = selectedNode.getUserObject() != null;

   if ( (nodeType == JarViewNode.MANIFEST_NODE)   ||
        (nodeType == JarViewNode.JAVA_NODE)       ||
        (nodeType == JarViewNode.TEXT_NODE)       ||
0112              (nodeType == JarViewNode.HTML_NODE)      ){
0113           if ( !objectAvailable) {
0114              textObject = theArchive.getText( pathName);
0115              selectedNode.setUserObject( imageObject);
0116           } else {
0117              textObject = (String) selectedNode.getUserObject();
0118           } // End if.
```

The use made of the large number of local variables will be explained as they are used within the method. The first step in the method, on lines 0100 and 0101, is to make sure that any existing contents of the *detailPane* are removed, as explained in the *loadJar()* method above. Lines 0103 to 0107 then prepare several of the local variables; the TreePath *selectionPath* variable is initialized to the *path* attribute of the TreeSelectionEvent *event*. This is a representation of the path from the root node to the newly selected node which is suitable for internal use within the tree but is not useable elsewhere. Accordingly, line 0104 calls the **private** *makePathName*() method to obtain a String representation of the path which will be in the same format as the entries in the *JarViewArchive contentsList*, as described above. The implementation of the *makePathName*() method will be given below.

Line 0105 retrieves the *JarViewNode* from *theTree* and its *nodeType* is then established before the **boolean** *objectAvailable* is set to indicate the existence, or otherwise, of a userObject at the selected node. A userObject is an arbitrary class instance which can be stored at any node and will be used in this artifact to avoid having to re-fetch information from the archive. For example, as will be described below, when an image node (`*.gif` or `*.jpeg`) is visited for the first time the image is retrieved from the archive as an imageIcon. This can be an expensive process, and rather than have to do it again if the node is revisited, the imageIcon is stored as the node's userObject resource so that it can be easily reused.

The substantive part of the method consists of a sequential decision structure, each branch of which deals with certain types of node. The first selection, commencing on line 0109, identifies those node types which have a text resource. Lines 0113 to 01118 either retrieve the *textObject* from *theArchive* and store it as the *userObject* of the *selectedNode*; or obtain the *textObject* directly from the node if it has already been stored there. Node types *MANIFEST*, *JAVA* (source code) and *TEXT* can be displayed within a JTextArea, but the *HTML* text resource must be displayed within a suitably configured JTextField, accomplished as follows.

```
0120           if ( nodeType != JarViewNode.HTML_NODE) {
0121              detailPane.getViewport().add(
0122                                new JTextArea( textObject));
0123           } else {
0124              JEditorPane toAdd =  new JEditorPane();
0125              toAdd.setContentType( "text/html");
0126              toAdd.setEditable( false);
0127              toAdd.setText( textObject);
0128              detailPane.getViewport().add( toAdd);
0129           } // End if.
```

For non-*HTML* nodes the *textObject* can be installed as the text resource of a *JTextArea* instance, which is then added to the viewport of the *detailPane* and which will be made visible as the method finishes. An example of this is given in Figure 12.4, which shows a

MANIFEST node selected and the text contents of the manifest file displayed in the *detailPane*. For an *HTML* node the process is a little more complex, requiring the *textObject* to be installed as the text resource of a JEditorPane instance which is non-editable and has had its contentType attribute set for HTML rendering. Once this has been accomplished, the JEditorPane is then added to the viewport of the *detailPane* as before. An example of the selection of an *HTML* node and the rendering of the contents from the archive is given in Figure 12.6.

The processing of *GIF* and *JPEG* nodes is somewhat similar, with the imageIcon retrieved from the archive or stored at the node being used as the *icon* resource of a JLabel installed within a JPanel, which itself is installed into the *detailPane* viewport, as follows.

```
} else if ( (nodeType == JarViewNode.GIF_NODE)  ||
            (nodeType == JarViewNode.JPEG_NODE) ){
   if ( !objectAvailable) {
      imageObject = theArchive.getGifImage( pathName);
      selectedNode.setUserObject( imageObject);
   } else {
     imageObject = (ImageIcon)selectedNode.getUserObject();
   } // End if.
   detailPane.getViewport().add( new JPanel().add(
                                new JLabel( imageObject)));
```

The next branch is concerned with the processing of all remaining node types, apart from *PACKAGE* nodes and *PATH* nodes. This will include all the leaf nodes which are not explicitly identified as text or image and, most significantly, the *CLASS* nodes. The branch is implemented as follows.

```
} else if ( ( nodeType != JarViewNode.PATH_NODE)    &&
            ( nodeType != JarViewNode.PACKAGE_NODE) ){
  if ( ! objectAvailable) {
      jarObject = theArchive.getJarObject( pathName);
      selectedNode.setUserObject( JarObject);
  } else {
      jarObject = (JarEntry) selectedNode.getUserObject();
  } // End if.
  detailPane.getViewport().add(
                     new JTextArea(
                         this.makeJarString( jarObject)));
```

This fragment installs a string representation of the attributes of an entry in the archive into a JTextArea. The *getJarObject()* method of the *JarViewArchive* class returns an instance of the JarEntry class and the **private** method *makeJarString()* will produce a String representation of its attributes, as follows.

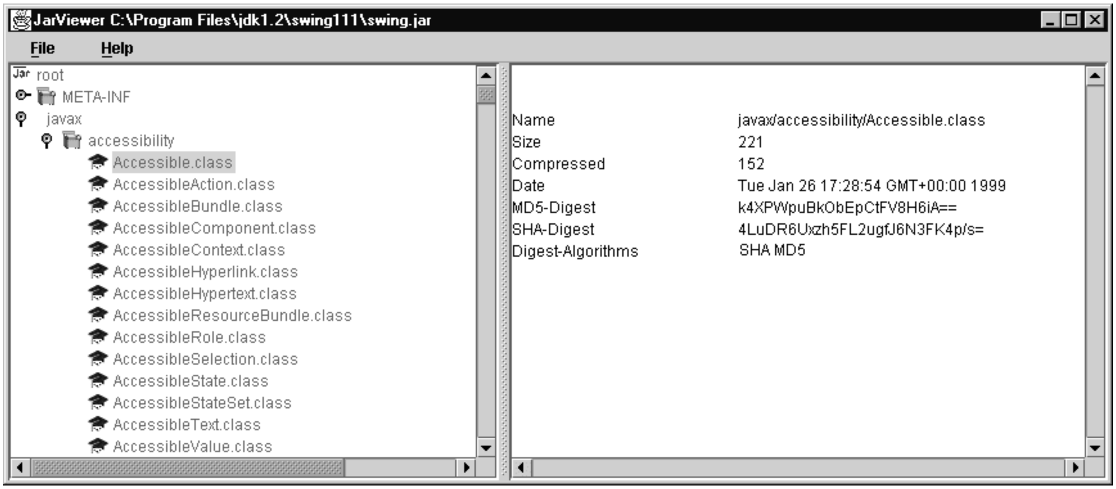
```
private String makeJarString( JarEntry fromThis) {

Attributes atts  = null;
Iterator   iter  = null;
Object     key   = null;
Object     value = null;

StringBuffer theString = new StringBuffer("");

0174        theString.append( "\n\nName\t\t"     + fromThis.getName());
0175        theString.append( "\nSize\t\t"       + fromThis.getSize());
0176        theString.append( "\nCompressed\t\t" +
0177                                 fromThis.getCompressedSize());
0178        theString.append( "\nDate\t\t"       +
0179                                new Date( fromThis.getTime()));
0180
0181        try {
0182           atts = fromThis.getAttributes();
0183           if ( atts != null) {
0184              iter =  atts.keySet().iterator();
0185              while ( iter.hasNext()) {
0186                 key   = iter.next();
0187                 value = atts.get( key);
0188                 theString.append( "\n"+ key +"\t\t" + value);
0189              } // End while.
0190           } // End if.
0191        } catch ( IOException e) {
0192           // Do nothing.
0193        } // End try/catch.
0194
0195        if ( fromThis.getCertificates() != null) {
0196           theString.append( "\nCertified");
0197        } // End if.
0198
0199        return theString.toString();
0200     } // End makeJarString.
```

The first part of this method, on lines 0174 to 0179, is relatively straightforward, retrieving the four major attributes which a JarEntry instance inherits from the ZipEntry class and catenating them, with prompts, into a StringBuffer. For many classes, and other resources contained within jar archives, this will be the only information available, and an example of such a resource can be seen in Figure 12.2.

However, an entry may have additional attributes which the rather complex routine on lines 0181 to 0193 will decode and install into the StringBuffer. The vital part of this fragment is within the **while** loop starting on line 0185, where each attribute *key* is taken in turn and used to obtain the associated attribute *value*. Additionally, a resource may be certified which is detected and added to the buffer on line 0196. Figure 12.13 shows a class view which illustrates additional attributes. The attributes include a Secure Hash Algorithm (SHA) and a Message Digest Version 5 (MD5) checksum which can be used to ensure that the contents have not been corrupted or interfered with during transmission. It also shows that the class files in this archive have been compressed.

The final part of the *valueChanged()* method, as follows, is concerned with *PACKAGE* nodes, and the method used to process them, called *showPackageDetails()*, will be described in the next section.

```
0155        } else if ( nodeType == JarViewNode.PACKAGE_NODE)  {
0156            this.showPackageDetails( selectionPath);
0157
0158        } // End if.
0159        this.paintAll( this.getGraphics());
```



root
META-INF
javax
accessibility
Accessible.class
AccessibleAction.class
AccessibleBundle.class
AccessibleComponent.class
AccessibleContext.class
AccessibleHyperlink.class
AccessibleHypertext.class
AccessibleResourceBundle.class
AccessibleRole.class
AccessibleSelection.class
AccessibleState.class
AccessibleStateSet.class
AccessibleText.class
AccessibleValue.class

Name	javax/accessibility/Accessible.class
Size	221
Compressed	152
Date	Tue Jan 26 17:28:54 GMT+00:00 1999
MD5-Digest	k4XPWpuBkObEpCtFV8H6iA==
SHA-Digest	4LuDR6Uxzh5FL2ugfJ6N3FK4p/s=
Digest-Algorithms	SHA MD5

Figure 12.13 *JarViewer* class view with additional attributes.

```
0160     } // End valueChanged.
```

The final part of the method, on lines 0159 and 0160, ensures that the new contents of the *displayPane*, if any, are made visible to the user by fully painting the applet. The *makePathName()* method was omitted from the description above and converts a TreePath into a String, as follows.

```
0204     private String makePathName( TreePath aPath) {
0205
0206     Object       parts[] = aPath.getPath();
0207     StringBuffer thePath = new StringBuffer("");
0208
0209        for( int index = 1;
0210                 index < parts.length;
0211                 index++ ) {
0212           thePath.append( parts[ index].toString() +"/");
0213        } // End for.
0214        return thePath.substring( 0, thePath.length()-1);
0215     } // End makePathName.
```

The getPath() method will return an array of Objects, each of which, in sequence, describes one node of the path. The substantive part of the method appends a string representation of each part of the path and a slash (/) to an initially empty StringBuffer. When the contents of the buffer are returned from the method as a String, on line 0214, the terminating slash is omitted.

12.7 The *JarViewer showPackageDetails()* method – introducing JTables

The *showPackageDetails()* method is called from the *valueChanged()* listener method when the user selects a *PACKAGE* node and is responsible for constructing and displaying a table view of the contents of the package, as shown in Figure 12.1. When a

JTable instance is placed into a JScrollPane's viewport the scroll pane will automatically supply a columnHeader component that will label each column in the table.

The first part of the method, as follows, is responsible for collecting the information that will be displayed in the table, which is obtained from all non-*PATH*, non-*PACKAGE* children of the node.

```
0219    private void showPackageDetails( TreePath path) {
0220
0221    JarViewNode selectedNode    = theTree.getNodeForPath( path);
0222    String      pathName        = this.makePathName( path);
0223    Enumeration allChildren     = selectedNode.children();
0224    Vector      packageChildren = new Vector();
0225    JarViewNode thisNode        = null;
0226    int         thisNodeType    = -1;
0227    String      childPath       = null;
0228
0229    JTable       theTable       = null;
0230    TableColumn  thisCol        = null;
0231    JarEntry     details        = null;
0232
0233       while ( allChildren.hasMoreElements()) {
0234          thisNode = (JarViewNode) allChildren.nextElement();
0235          thisNodeType = thisNode.getNodeType();
0236          if ( (thisNodeType != JarViewNode.PATH_NODE)    &&
0237               (thisNodeType != JarViewNode.PACKAGE_NODE) ){
0238             packageChildren.add( thisNode);
0239          } // End if.
0240       } // End while.
```

The method commences, on line 0221, by retrieving the selected node from *theTree* using the *path* argument passed to it. Line 0222 obtains a representation of the *path* as a String and an Enumeration of the children of the selected node is obtained on line 0223. The use made of the other local variables will be explained in the description of the body of the method.

JComponent →javax.swing.JTable

```
public JTable( int numRows, int numCols)
public JTable( Object[][] tableData, Object[] columnHeaders)
public JTable( Vector tableData, Vector columnHeaders)
```

Constructs a table with the number of rows and columns as indicated directly or inferred from the data structures provided.

```
public Object  getValueAt( int row, int column)
public void    setValueAt( Object aValue, int row, int column)
public boolean isCellEditable( int row, int column)
```

Setter and getter for the value in an individual cell and an inquiry method for its editability.

```
public TableColumn getColumn( Object identifier)
public int         getColumnCount()
public String      getColumnName( int column)
public void        removeColumn( TableColumn aColumn)
public void        moveColumn( int column, int targetColumn)
public int         getRowCount()
```

Methods to manipulate the table as a sequence of columns.

```
public boolean getShowHorizontalLines()
public void    setShowHorizontalLines( boolean boolean)
public boolean getShowVerticalLines()
public void    setShowVerticalLines( boolean yesOrNo)
public void    setGridColor( Color newColor)

public JTableHeader getTableHeader()
public void setTableHeader( JTableHeader newHeader)
```

Methods concerned with the more decorative aspects of a table.

The **while** loop commencing on line 0233 retrieves each child node in turn and, if it is non-*PATH* and non-*PACKAGE*, adds it to the Vector *packageChildren*. The effect of the loop is to store the identity of all nodes that have to be described in the Vector. The method continues as follows.

```
0242       theTable = new JTable( packageChildren.size(), 4);
0243       theTable.getColumn( "A").setHeaderValue( "Name");
0244       theTable.getColumn( "B").setHeaderValue( "Size");
0245       theTable.getColumn( "C").setHeaderValue( "Comp");
0246       theTable.getColumn( "D").setHeaderValue( "Date");
0247
0248       for ( int row =0;
0249                row < packageChildren.size();
0250                row++) {
0251          thisNode = (JarViewNode)
0252                           packageChildren.elementAt(row);
0253          childPath = pathName + "/" + thisNode.getName();
0254          details   = theArchive.getJarEntry( childPath);
0255
0256          theTable.setValueAt( thisNode.getName(),row, 0);
0257          theTable.setValueAt( new Long(
0258                                  details.getSize()),row, 1);
0259          theTable.setValueAt( new Long(
0260                         details.getCompressedSize()),row, 2);
0261          theTable.setValueAt( new Date(
0262                                  details.getTime()),row, 3);
0263       } // End for.
0264
0265       detailPane.getViewport().add( theTable);
0266    } // End showPackageDetails.
```

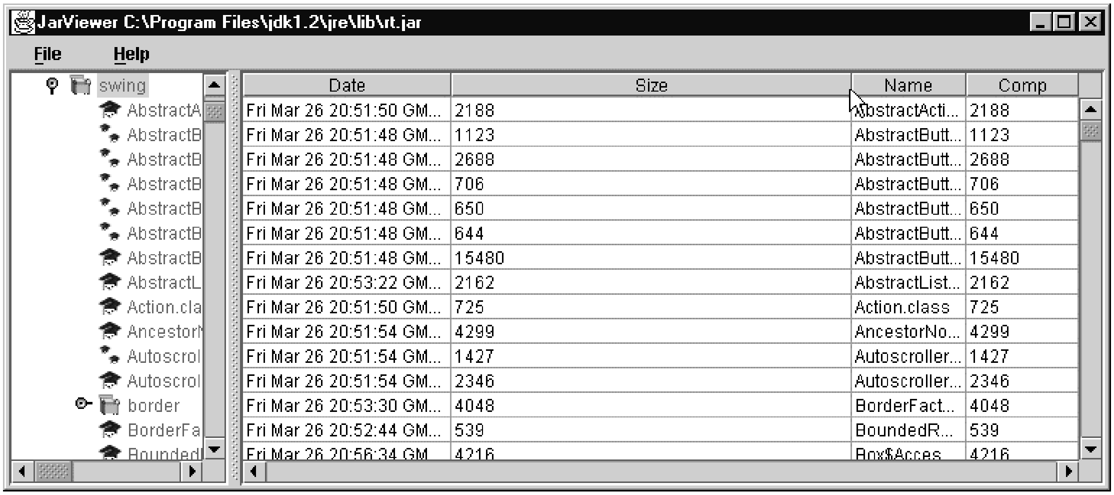
On line 0242 the JTable instance, called *theTable*, is constructed with the arguments to the constructor specifying the number of rows, in this example the number of elements in the *packageChildren* Vector, and the number of columns, in this example 4. Lines 0243 to 0246 then specify the four column titles, by first retrieving the column identified by its default column title (“A”, “B”, “C” or “D”) and then using the TableColumn setHeaderValue() method to install a new title. When a JTree is installed into a JScrollPane’s viewport, the columnHeader of the table is automatically installed as a second viewport above it, so that the header will always remain visible while the table is vertically scrolled beneath it. Hence the need to ensure that the columnHeader viewport of the *detailPane* was cleared, as well as the main viewport, in the implementation of the *JarViewer* class above.

The **for** loop, commencing on line 0248, populates the table by specifying the Object value for each cell in the table, obtaining them from the four principle attributes of the JarEntry class. As a table can only contain Objects in its cells, the primitive **long** values, indicating the compressed and uncompressed size, must be converted into Long objects, and the **long** value indicating its time of creation or modification is used as the argument to a Date constructor.

This technique will produce a simple table view of the attributes of the entries in a package as shown in Figure 12.1. Figure 12.14 shows two additional features of the default table implementation. The order of the columns can be varied by dragging a column header left or right and the width of a column can be varied by dragging on the separator between the columns.

One additional feature of the table is not apparent from its visual appearance, which is that by default the cells are editable. If a cell is double-clicked upon, an editable JTextField allows its contents to be changed. For many, if not most, requirements this is an essential and valuable feature. For example, it allows the development of a spreadsheet-type artifact or a view onto a mutable database. However, for this artifact this is undesirable behavior and the simplest way to prevent the cells in a table from being changed is to extend the JTable class overriding any appropriate constructors and also the *isCellEditable()* method. For example, the following class, called *JarViewTable*, supplies a non-editable table suitable for use in the *JarViewer* artifact.

```
0010  package jarview;
0011
0012  import javax.swing.*;
0013
```



Date	Size	Name	Comp
Fri Mar 26 20:51:50 GM...	2188	AbstractActi...	2188
Fri Mar 26 20:51:48 GM...	1123	AbstractButt...	1123
Fri Mar 26 20:51:48 GM...	2688	AbstractButt...	2688
Fri Mar 26 20:51:48 GM...	706	AbstractButt...	706
Fri Mar 26 20:51:48 GM...	650	AbstractButt...	650
Fri Mar 26 20:51:48 GM...	644	AbstractButt...	644
Fri Mar 26 20:51:48 GM...	15480	AbstractButt...	15480
Fri Mar 26 20:53:22 GM...	2162	AbstractList...	2162
Fri Mar 26 20:51:50 GM...	725	Action.class	725
Fri Mar 26 20:51:54 GM...	4299	AncestorNo...	4299
Fri Mar 26 20:51:54 GM...	1427	Autoscroller...	1427
Fri Mar 26 20:51:54 GM...	2346	Autoscroller...	2346
Fri Mar 26 20:53:30 GM...	4048	BorderFact...	4048
Fri Mar 26 20:52:44 GM...	539	BoundedR...	539
Fri Mar 26 20:56:34 GM	4216	Box$Acces	4216

Figure 12.14 *JarViewer* table view illustrating column rearrangement and resizing.

```
public class JarViewTable extends JTable {

   public JarViewTable( int numRows, int numCols) {
      super( numRows, numCols);
   } // End JarViewTable constructor.

   public boolean isCellEditable( int row, int column) {
      return false;
   } // End isCellEditable.

} // End JarViewiTable.
```

The JTable class, and its support classes in the javax.swing.table package, is probably the most complex component in the JFC. This is a very simple introductory example of a non-editable table, using the default string-based cell rendering, with the data individually supplied for each cell. More realistic uses of the JTable might involve supplying the data from a Vector of Vectors or from a specialized data structure known as a TreeModel, and might also require specialized renderers and editors. However, the techniques for accomplishing this are outside the scope of this book.

Summary

- The JTree component contains a collection of nodes, connected to its root node, which can represent a hierarchical structure such as a file system.
- Nodes within a tree must implement the TreeNode interface and can be conveniently extended from the DefaultMutableTreeNode class.
- A node can be expanded, showing any child nodes associated with it, or collapsed.
- A node is a leaf node if it has no children; otherwise it is non-leaf node.
- A JTree dispatches a TreeExpansionEvent to a TreeExpansionListener treeCollapsed() or treeExpanded() method when it is collapsed or expanded.
- A JTree dispatches a TreeSelectionEvent to a Tree SelectionListener valueChanged() method when it is selected.
- A TreePath instance represents the path from the root node to a node in the tree, or between any two nodes, and is an attribute of TreeExpansionEvents and TreeSelectionEvents.
- By default, a leaf node is presented with a document icon and a non-leaf node with a folder icon. However, a TreeRenderer can be supplied to give a specialized view.
- A JTable contains a sequence of TreeColumns which are, by default, resizable and reorderable, and each of which contains a sequence of rows.
- By default the cells in a JTable are editable and the easiest technique to provide a non-editable table is to extend it overriding the isCellEditable() method to always return **false**.

Exercises

12.1 Extend the *JarViewer* artifact so that it can obtain a jar archive from across the Internet.

12.2 Extend the *JarViewNode* and *JarViewNodeRenderer* classes so that additional types of file can be identified and viewed.

12.3 Extend the *JarViewer* artifact so that it can unpack the selected entry, package or leaf, into the host file system. (There are no facilities for adding entries into jar files, so the reverse operation cannot be accomplished.)

12.4 Extend the table view so that each of the component fields of the Date attribute (year, month, day of month and) time are presented in separate columns.

12.5 Provide the *JarViewer* artifact with a *Sort* menu which contains items corresponding to each of the column titles. When the item is selected the table should be reordered according to the column indicated.

12.6 Investigate the protocol of the File class and implement an artifact which allows the user to explore the file system, supplying suitable views for various file types.

12.7 The Class class supplies methods to obtain details about the methods and other resources contained within a class. Investigate the protocol of the Class class and extend the *JarViewer* so that etch class icon can be expanded to list the names of the methods and other resources. The view associated with a class should be its full signature indicating the type of object returned and the number and types of the arguments.

Appendix

A Web resource containing support for this book, including an archive of source code, and other resources, is available at:

`http://www.sbu.ac.uk/jfl/jfcbook/`

An introductory text on object-oriented design and development in Java is:

Fintan Culwin (1998) *Java, an Object First Approach*. Prentice Hall, London. ISBN 0-13-858457-5

A detailed introduction to the Abstract Windowing Toolkit (AWT) is provided in:

Fintan Culwin (1998) *A Java GUI Programmer's Primer*. Prentice Hall, London. ISBN 0-13-908849-0

The book on statechart-based UI design mentioned in the preface is

Ian Horrocks (1999) *Constructing the User Interface with Statecharts*. Addison Wesley Longman, Harlow. ISBN 0-201-34287-1

The most useful of the Swing tomes mentioned in the preface is:

David M. Geary (1999) *Graphic Java*, Vol. 2, 3rd edn. Prentice Hall, London. ISBN 0-13-079667-0

Detailed information on usability engineering procedures is available in:

Jackob Neilsen (1994) *Usability Engineering*. Academic Press, London. ISBN 0-125-18406-9

A more general reference on the design of user interfaces is available in:

Ben Schneiderman (1997) *Designing the User Interface*, 3rd edn. Addison Wesley Longman, Harlow. ISBN 0-201-69497-2

A Web resource maintained by Ben Schneiderman specializing in usability and the Web is available at:

`http://www.aw.com/DTUI/`

The Bean Development Kit and the JavaBean White Paper are available at:

`http://www.javasoft.com/`

Lists of the ISO 639 region codes and the ISO 3166 language codes are available at:

```
http://www.chemie.fu-berlin.de/diverse/doc/ISO_639.html
http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html
```

Index