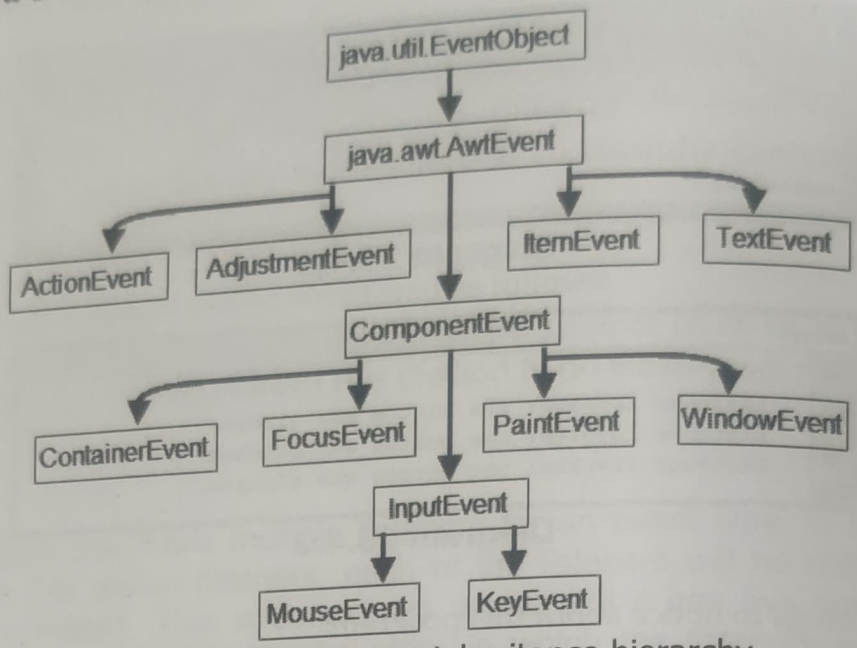

Diagram 25.5: Event inheritance hierarchy

The ancestor of the hierarchy is **java.util.EventObject**, which provides **getSource()**, which returns the component in which the event took place. One level below is **java.awt.AWTEvent**, which provides a method called **getID()**, which returns an **int** that describes the nature of the event. For example, calling **getID()** on an instance of **MouseEvent** results in an **int** whose value might be **MouseEvent.MOUSE_PRESSED**, **MouseEvent.MOUSE_DRAGGED** or one of several other possible values, depending on which specific mouse activity triggered the event.

Event Classes

There are many different kinds of events to which a program may need to respond - from menus, from buttons, from the mouse, from the keyboard and a number of others. In order to have a structured approach to handling events, these are broken down into subsets. At the topmost level, there are two broad categories of events in Java:

- **Low-level Events:** Are events that arise from the keyboard or from the mouse or events associated with operations on a window such as reducing it to an icon or closing it. The meaning of a low-level event is something like 'the mouse was moved', 'this window has been closed' or 'this key was pressed'. There are four kinds of low-level events. They are represented by the following classes in **java.awt.event**. They are:
  - FocusEvent
  - MouseEvent
  - KeyEvent
  - WindowEvent

- **Semantic Events:** Are specific component-related events such as pressing a button by clicking it to cause some program action or adjusting a scrollbar. They originate and are then interpreted, in the context of the GUI created for the program. The meaning of a semantic event is typically something like 'the OK button was pressed' or 'the Save menu item was selected'. Each kind of component, a button or a menu item for example, can generate a particular kind of semantic event. There are three types of semantic events. They are:
  - ActionEvent
  - ItemEvent
  - AdjustmentEvent

These two categories can seem to be a bit confusing as they overlap in a way. If a button is clicked, a **semantic event** is created as well as a **low-level event**. The click produces a low-level event object in the form of 'the mouse was clicked' as well as a semantic event 'the button was pushed'. In fact it produces more than one mouse event. Whether the program handles the low-level events or the semantic event or possibly both kinds of event, depends on what has to be done.

Most of the events relating to the GUI for a program are represented by classes defined in **java.awt.event**. This package also defines the listener interfaces for the various kinds of events that it defines. **javax.swing.event** defines classes for events that are specific to Swing components.

Sensible Event Handling

Events may be handled by the originating component or they may be delegated to the listener. To create components that handle their own events, answers to the following must be known:

- For each event type, which component types can generate the event?
- For each event type, what value should a component pass to enableEvents(), in order to receive notification when the event happens?
- For each event type, which method is called in the component when the event occurs?

If component event handling is been delegated to listeners, answers to the following questions must be known:

- For each event type, which component types can generate the event?
- For each event type, what interface should the listener implement?
- For each event type, which method in the listener is called?

Adjustment Events

Adjustment events are sent by scrollbars.

```
public class AdjustmentEvent extends AWTEvent {
   public static final int BLOCK_DECREMENT;
   public static final int BLOCK_INCREMENT;
   public static final int TRACK;
   public static final int UNIT_DECREMENT;
   public static final int UNIT_INCREMENT;

   public AdjustmentEvent(Adjustable source, int id, int type, int value);
   public Adjustable getAdjustable();
   public int getAdjustmentType();
   public int getValue();
   public String paramString();
}
```

If a component delegates its adjustment events, the delegation must implement AdjustmentListener.

```
public interface AdjustmentListener extends EventListener {
   public void adjustmentValueChanged(AdjustmentEvent AdjEvt);
}
```

Example

Code spec: [ScrollBarTest.java]

```
1  import java.applet.Applet;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class ScrollBarTest extends Applet implements AdjustmentListener {
6     @Override
7     public void init() {
8        setLayout(new BorderLayout());
9
10    /* A plain scrollbar that delegates to the applet */
11       Scrollbar sbar1 = new Scrollbar();
12       sbar1.addAdjustmentListener(this);
13       add(sbar1, "West");
14
15    /* A subclass that handles its own adjustment events */
16       SelfScrollbar sbar2 = new SelfScrollbar();
17       add(sbar2,"East");
18    }
19
20    @Override
21    public void adjustmentValueChanged(AdjustmentEvent AdjEvt) {
22       System.out.println("scrollbar #1:"+ AdjEvt.getValue());
```

```
    }
}
class SelfScrollbar extends Scrollbar {
  public SelfScrollbar() {
    enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);

  }
  @Override
  public void processAdjustmentEvent(AdjustmentEvent AdjEvt) {
    System.out.println("Scrollbar #2:"+ AdjEvt.getValue());
  }
}
```

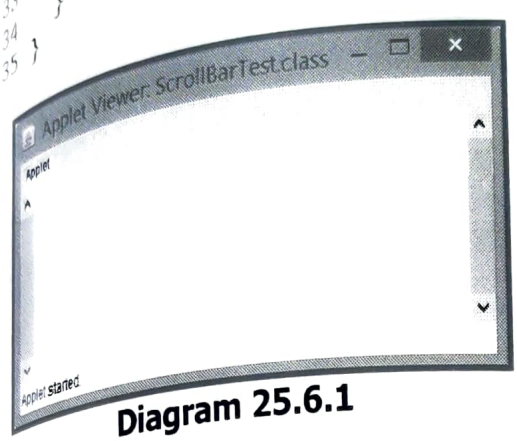

Diagram 25.6.1

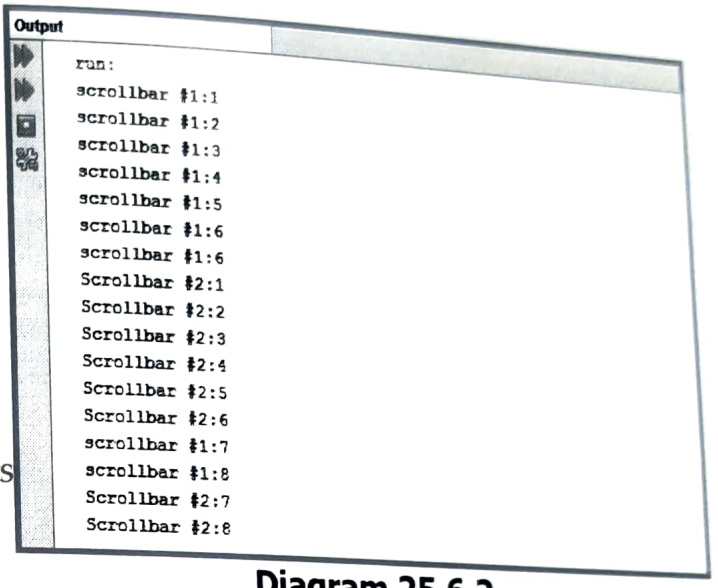

Diagram 25.6.2

The applet code listed above constructs two scrollbars.

The first is an ordinary scrollbar that delegates adjustment events to the applet. When the scrollbar moves, the applet writes a message to the console. The first scrollbar delegates **adjustment events** to the applet, so the applet class must implement **AdjustmentListener**.

The second scrollbar is a subclass called **SelfScrollbar**. This subclass handles its own adjustment events. When the **SelfScrollbar** moves, the scrollbar itself writes a message to the console. The second scrollbar handles its own adjustment events by calling **EnableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK)** and providing **processedAdjustmentEvent()**.

Container Events

Container events occur when a component is added to or removed from a container. Here is the definition of ContainerEvent:

```
public class ContainerEvent extends ComponentEvent {
  public static final int COMPONENT_ADDED;
  public static final int COMPONENT_REMOVED;
```

```
    public ContainerEvent(Component source, int id, Component child);
    public Component getChild();
    public Container getContainer();
    public String paramString();
}
```

A container can process its own events by calling EnableEvents(AWTEvent.CONTAINER_EVENT_MASK) and by providing processContainerEvent().

```
public interface ContainerListener extends EventListener {
    public void componentAdded(ContainerEvent ContEvt);
    public void componentRemoved(ContainerEvent ContEvt);
}
```

REMINDER

The low-level event class, ContainerEvent, defines events relating to a container such as adding or removing components. This event can be ignored as these events are handled automatically.

Focus Events

Focus events are sent when a component gains or loses keyboard input focus.

The following is the definition of **FocusEvent**:

```
public class FocusEvent extends ComponentEvent {
    public static final int FOCUS_GAINED;
    public static final int FOCUS_LOST;
    public FocusEvent(Component source, int id, boolean temporary);
    public FocusEvent(Component source, int id);
    public boolean isTemporary();
    public String paramString();
}
```

A component can process its own focus events by calling enableEvents(AWTEvent.FOCUS_EVENT_MASK) and providing processFocusEvent(). Alternatively, you can delegate focus events to listener that implements the FocusListener interface:

```
public interface FocusListener extends EventListener {
    public void focusGained(FocusEvent FocEvt);
    public void focusLost(FocusEvent FocEvt);
}
```

Example

The following applet uses a BorderLayout manager to put a text field at "north" and a text area at "center". The text field delegates its focus events to the applet. The text area is a subclass that handles its own focus events. Depending upon whether focus gained or lost, both the event handlers print a message to the console.

Code spec: **[FocusEventTest.java]**

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class FocusEventTest extends Applet implements FocusListener {
   @Override
   public void init() {
      setLayout(new BorderLayout());
   /* A text field that delegates to the applet */
      TextField txt = new TextField();
      txt.addFocusListener(this);
      add(txt,"North");
   /* A subclass that handles its own focus events */
      SelfTextArea slfta = new SelfTextArea();
      add(slfta,"Center");

   }

   @Override
   public void focusGained(FocusEvent FocEvt) {
      System.out.println("Text field gained focus");

   }

   @Override
   public void focusLost(FocusEvent FocEvt) {
      System.out.println("Text field lost focus");
   }
}

class SelfTextArea extends TextArea {
   public SelfTextArea() {
      enableEvents(AWTEvent.FOCUS_EVENT_MASK);
   }

   @Override
   public void processFocusEvent(FocusEvent FocEvt) {
      if(FocEvt.getID() == FocusEvent.FOCUS_GAINED)
         System.out.println("Text area gained focus");
      else
         System.out.println("Text area lost focus");
   }
}
```

Diagram 25.7.1

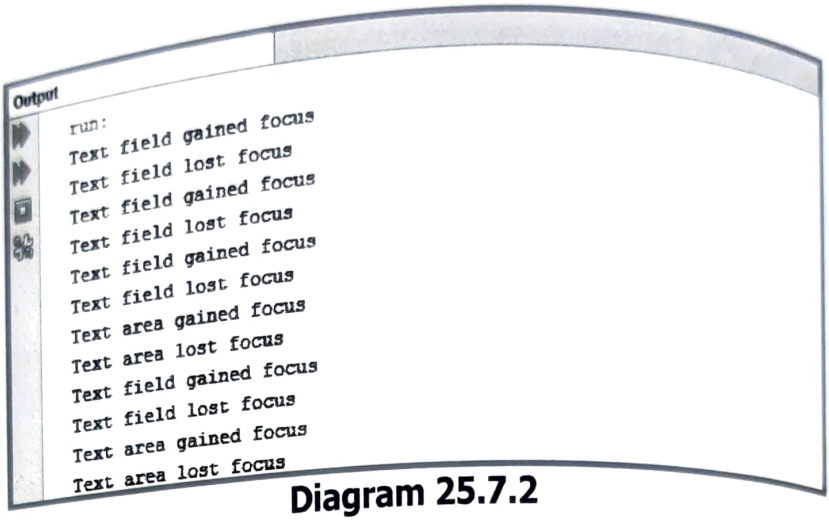

Diagram 25.7.2

Item Events

Item events are generated by components that present users with items to choose from components that generate these events are Choice, List, Checkbox and CheckboxMenuIte The following is the definition of ItemEvent:

```
public class ItemEvent extends AWTEvent {
   public static final int DESELECTED;
   public static final int ITEM_STATE_CHANGED;
   public static final int SELECTED;
   public ItemEvent(ItemSelectable source,int id,Object item,int stateChanged);
   public Object getItem();
   public ItemSelectable getItemSelectable();
   public int getStateChange();
   public String paramString();
}
```

A component can process its own item events by calling enableEvents(AWTEvent.ITEM_EVENT_MASK) and providing processItemEvent Alternatively, a component can delegate item events to a listener that implements ItemListener:

```
public interface ItemListener extends EventListener {
   void itemStateChanged(ItemEvent ItmEvt);
}
```

Example

The applet listed below consists of a list component and a choice component. The List delegates its item events to the applet. The choice is a subclass that handles its own item events. Both event handlers print a message to the console.

Code spec: **[ItemEventTest.java]**

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ItemEventTest extends Applet implements ItemListener {
   List lt;
   SelfChoice slfch;

   @Override
   public void init() {
   /* A list that delegates to the applet. */
      lt = new List(5,false);
      lt.add("Chocolate");
      lt.add("Vanilla");
      lt.add("Strawberry");
      lt.add("Mocha");
      lt.add("Peppermint Swirl");
      lt.add("Blackberry Ripple");
      lt.add("Butterscotch");
      lt.add("Almond");
      lt.addItemListener(this);
      add(lt);

   /* A choice subclass that handles its own item events */
      slfch = new SelfChoice();
      slfch.addItem("Ice Cream");
      slfch.addItem("Frozen Yogurt");
      slfch.addItem("Sorbet");
      add(slfch);
   }

   @Override
   public void itemStateChanged(ItemEvent ItmEvt) {
      System.out.println("New item from list:" + lt.getSelectedItem());
   }
}

class SelfChoice extends Choice {
   public SelfChoice() {
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
   }

   @Override
```

```
44      public void processItemEvent(ItemEvent ItmEvt) {
45          System.out.println("New item from choice: "+ getSelectedItem());
46      }
47  }
```

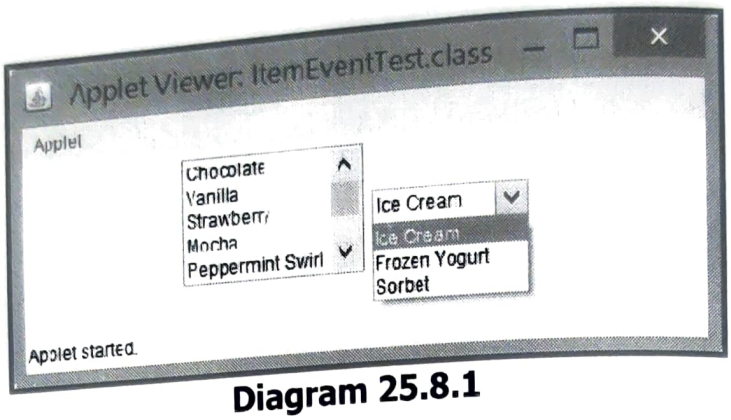

Diagram 25.8.1

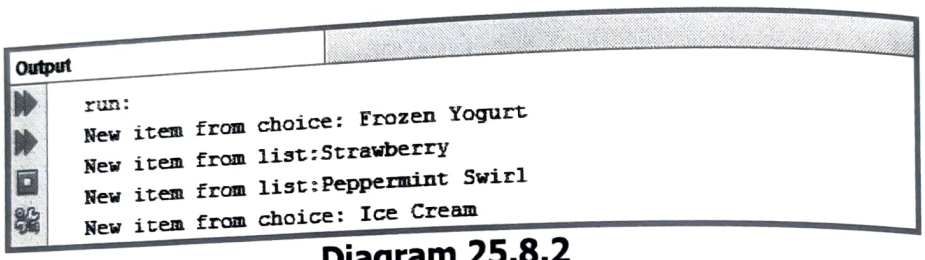

Diagram 25.8.2

REMINDER

Item events are the only kind of events created by a list component. If the list element is double-clicked, the list will generate an **ActionEvent**.

Key Events

Key events are generated when a user presses or releases a key on the keyboard. It is extensive because there are a large number of constants. The first three constants appear in the event's id field and describe the key event:

- KEY_PRESSED indicates that a key was pushed down
- KEY_RELEASED indicates that a key was released
- KEY_TYPED denotes a key press followed by a key release

The remaining constants all have names that begin with VK, which stands for virtual key such as VK_0...VK_9, VK_NUMPAD0...VK_NUMPAD9, VK_F1...VK_F12, VK_A...VK_Z, VK_HELP, VK_HOME, VK_UP and so on.

A component can process its own key events by calling enableEvents(AWTEvent.KEY_EVENT_MASK) and providing processKeyEvent().

Alternatively, a component can delegate key events to a listener that implements KeyListener:

```
public interface KeyListener extends EventListener {
   public void keyPressed(KeyEvent KyEvt);
   public void keyReleased(KeyEvent KyEvt);
   public void keyTyped(KeyEvent KyEvt);
}
```

Example

The following applet uses a BorderLayout manager to put a text field at "North" and a text area at "Center". The text field delegates all its key events to the applet. The textarea is a subclass that handles its own key events. Both the event handlers print a message to the console. The events, which are trapped, are key Typed, key Released and key Pressed.

Code spec: [KeyEventTest.java]

```
1  import java.applet.Applet;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class KeyEventTest extends Applet implements KeyListener {
6     @Override
7     public void init() {
8        setLayout(new BorderLayout());
9     /* A text field that delegates to the applet */
10       TextField txt = new TextField();
11       txt.addKeyListener(this);
12       add(txt,"North");
13    /* A text area subclass that handles its own item events */
14       SelfKeyTextArea slfTa = new SelfKeyTextArea();
15       add(slfTa,"Center");
16    }
17
18    @Override
19    public void keyTyped(KeyEvent KyEvt) {
20       System.out.println("Key typed in text field: "+ KyEvt.getKeyChar());
21    }
22
23    @Override
24    public void keyPressed(KeyEvent KyEvt) {}
25    @Override
26    public void keyReleased(KeyEvent KyEvt) {}
27 }
28
29 class SelfKeyTextArea extends TextArea {
30    public SelfKeyTextArea() {
31       enableEvents(AWTEvent.KEY_EVENT_MASK);
32    }
33
34    @Override
```

```
390     public void processKeyEvent(KeyEvent KyEvt) {
35        if(KyEvt.getID() == KeyEvent.KEY_TYPED)
36          System.out.println("Key typed in text area: " + KyEvt.getKeyChar());
37      }
38  }
```

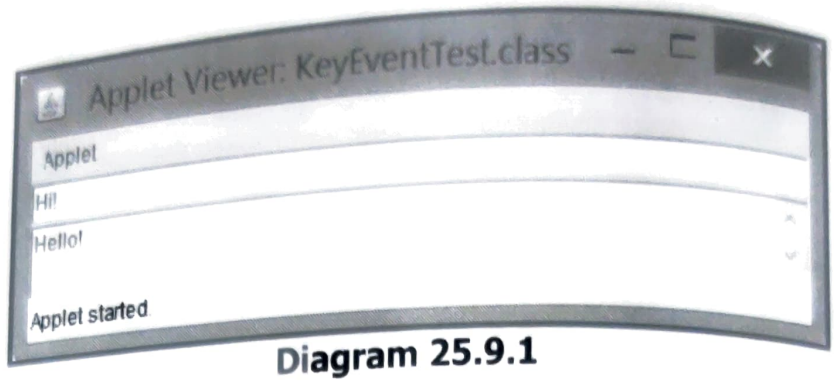

Diagram 25.9.1

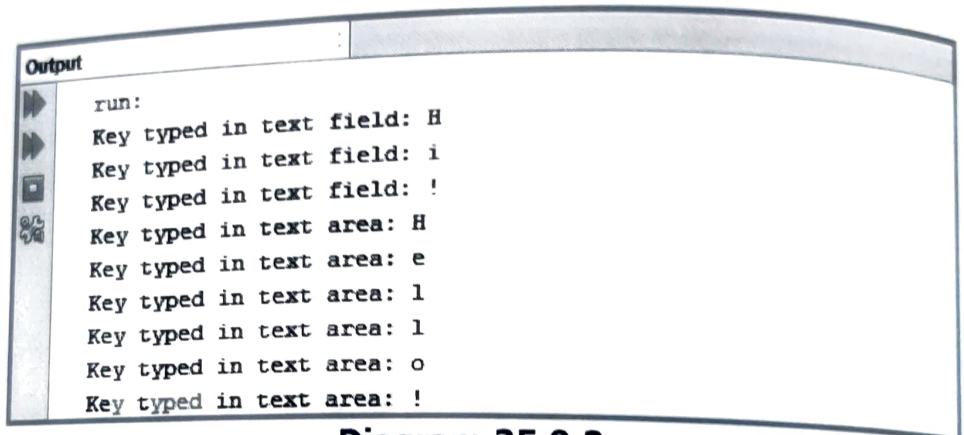

Diagram 25.9.2

Mouse Events

Mouse events are generated when the user clicks a mouse button or moves the mouse. There are six mouse event types, represented by the constants in **MouseEvent**. These constants are the possible values for a mouse events id field. Here is the definition for **MouseEvent**:

```
public class MouseEvent extends InputEvent {
   public static final int MOUSE_DRAGGED;
   public static final int MOUSE_ENTERED;
   public static final int MOUSE_EXITED;
   public static final int MOUSE_MOVED;
   public static final int MOUSE_PRESSED;
   public static final int MOUSE_RELEASED;

   public MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int
clickCount, boolean popupTrigger);
   public int getClickCount();
   public Point getPoint();
   public int getX();
   public int getY();
```

```
    public boolean isPopupTrigger();
    public String paramString();
    public synchronized void translatePoint(int x, int y);
}
```

Generally speaking, programs treat mouse-moved and mouse-dragged events very differently from the way they treat the other four event types. Java provides two mouse events and two mouse event masks, so that programs can deal separately with ordinary mouse events [pressed, released, entered and exited] and **mouse-motion** events [moved and dragged].

A component can process its own ordinary mouse events by calling enableEvents(AWTEvent.MOUSE_EVENT_MASK) and providing processMouseEvent(). Alternatively, a component can delegate ordinary mouse events to a listener that implements MouseListener.

```
public interface MouseListener extends EventListener {
    public void mouseClicked(MouseEvent MosEvt);
    public void mouseEntered(MouseEvent MosEvt);
    public void mouseExited(MouseEvent MosEvt);
    public void mousePressed(MouseEvent MosEvt);
    public void mouseReleased(MouseEvent MosEvt);
}
```

A mouse can process its own mouse-motion events by calling enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK) and providing processMouseMotionEvent(). Alternatively, a component can delegate mouse-motion events to a listener that implements MouseMotionListener:

```
public interface MouseMotionListener extends EventListener {
    public void mouseDragged(MouseEvent MosEvt);
    public void mouseMoved(MouseEvent MosEvt);
}
```

Example

The following applet creates two canvases. The upper canvas is yellow. It delegates all its mouse and mouse-motion events to the applet. The lower canvas is green. It is a subclass that handles its own mouse and mouse-motion events. All four event handlers (two in the applet, two in the canvas subclass) write a message to the console. The only events of interest to this applet are mouse pressed, mouse released and mouse entered, all other mouse event types are ignored.

Code spec: **[MouseEventTest.java]**

```
1 import java.applet.Applet;
2 import java.awt.*;
```

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

public class MouseEventTest extends Applet implements MouseListener,
MouseMotionListener {
    @Override
    public void init() {
        setLayout(new GridLayout(2,1));

    /* A canvas that delegates to the applet */
        Canvas can = new Canvas();
        can.setBackground(Color.yellow);
        can.addMouseListener(this);
        can.addMouseMotionListener(this);
        add(can);

    /* A canvas subclass that handles its own item events */
        SelfMouseCanvas slfCan = new SelfMouseCanvas();
        add(slfCan);
    }

    @Override
    public void mousePressed(MouseEvent MosEvt) {
        System.out.println("UPPER: mouse pressed at "+ MosEvt.getX() + "," +
        MosEvt.getY());
    }

    @Override
    public void mouseReleased(MouseEvent MosEvt) {
        System.out.println("UPPER: mouse released at "+ MosEvt.getX() + ","
        +MosEvt.getY());
    }

    @Override
    public void mouseEntered(MouseEvent MosEvt) {
        System.out.println("UPPER: mouse entered");
    }

    @Override
    public void mouseExited(MouseEvent MosEvt) {}
    @Override
    public void mouseClicked(MouseEvent MosEvt) {}
    @Override
    public void mouseMoved(MouseEvent MosEvt) {}
    @Override
    public void mouseDragged(MouseEvent MosEvt) {}
}

class SelfMouseCanvas extends Canvas {
    public SelfMouseCanvas() {
        setBackground(Color.green);
        enableEvents(AWTEvent.MOUSE_EVENT_MASK |
        AWTEvent.MOUSE_MOTION_EVENT_MASK);
    }
```

```
@Override
public void processMouseEvent(MouseEvent MosEvt) {
  if(MosEvt.getID() == MouseEvent.MOUSE_PRESSED)
    System.out.println("LOWER: mouse pressed at " + MosEvt.getX() + " " +
    MosEvt.getY());
  else if (MosEvt.getID() == MouseEvent.MOUSE_RELEASED)
    System.out.println("LOWER: mouse released at " + MosEvt.getX() + " " +
    MosEvt.getY());
  else if (MosEvt.getID() == MouseEvent.MOUSE_ENTERED)
    System.out.println("LOWER: mouse entered");
}
}
```

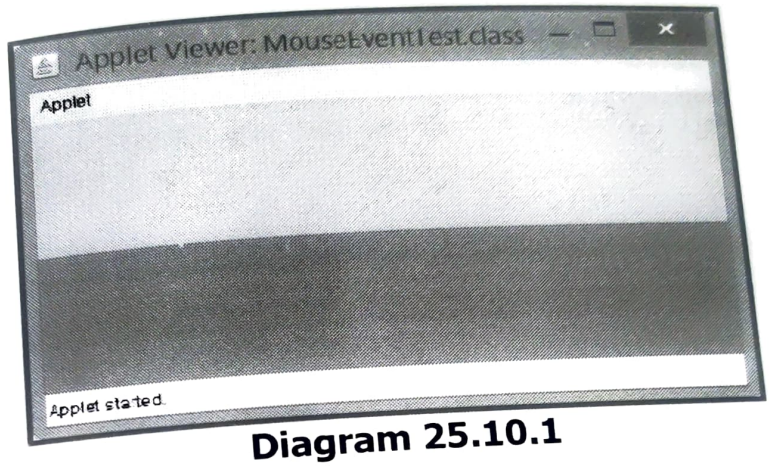

Diagram 25.10.1

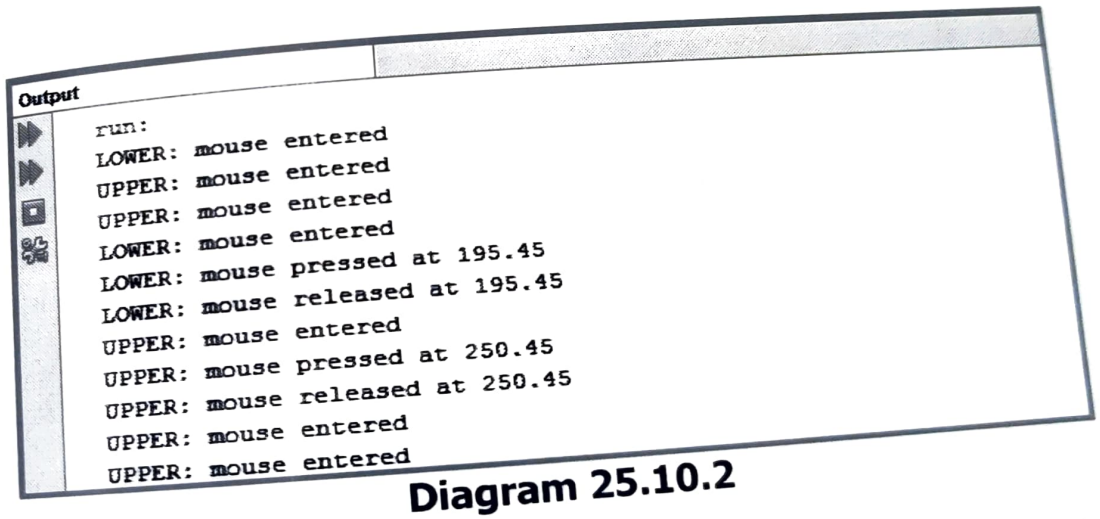

Diagram 25.10.2

Notice that the applet subclass declares that it implements not one but **two** interfaces. There is nothing wrong with this, Java's single inheritance concerns extending classes and has nothing to do with implementing interfaces.

Notice also that the canvas subclass enables two event types in a single instruction:

```
enableEvents(AWTEvent.MOUSE_EVENT_MASK |
AWTEvent.MOUSE_MOTION_EVENT_MASK);
```

The two event masks are ORed together using the | operator, which performs a bit operation on its two arguments, which are of type **long**.

Text Events

Text events are sent to text components [text field and text areas] when a change occurs in the text they contain. This happens when the user types something or when a program calls a method such as setText().

The following is the definition of **TextEvent**:

```
public class TextEvent extends AWTEvent {
   public static final int TEXT_VALUE_CHANGED;
   public TextEvent(Object source, int id);
   public String paramString();
}
```

A text component can process its own text events by calling enableEvents(AWTEvent.TEXT_EVENT_MASK) and providing processTextEvent(). Alternatively, a text component can delegate text events to a listener that implements TextListener:

```
public interface TextListener extends EventListener {
   public void textValueChanged(TextEvent TxtEvt);
}
```

Example

The following applet contains two text areas: the upper one delegates text events to the applet and the lower one handles its own text events. Both event handlers just print a message to the console.

Code spec: [TextEventTest.java]

```
1  import java.applet.Applet;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class TextEventTest extends Applet implements TextListener {
6     @Override
7     public void init() {
8        setLayout(new GridLayout(2,1));
9     /* A text area that delegates to the applet */
10       TextArea ta = new TextArea();
11       ta.addTextListener(this);
12       add(ta);
13    /* A text area subclass that handles its own item events */
14       SelfTextTA slfta = new SelfTextTA();
```

```
        add(slfta);
    }

    @Override
    public void textValueChanged(TextEvent TxtEvt) {
        System.out.println("UPPER get text event: " + TxtEvt);
    }
}

class SelfTextTA extends TextArea {
    public SelfTextTA() {
        enableEvents(AWTEvent.TEXT_EVENT_MASK);
    }

    @Override
    public void processTextEvent(TextEvent TxtEvt) {
        System.out.println("LOWER get text event: " + TxtEvt);
    }
}
```

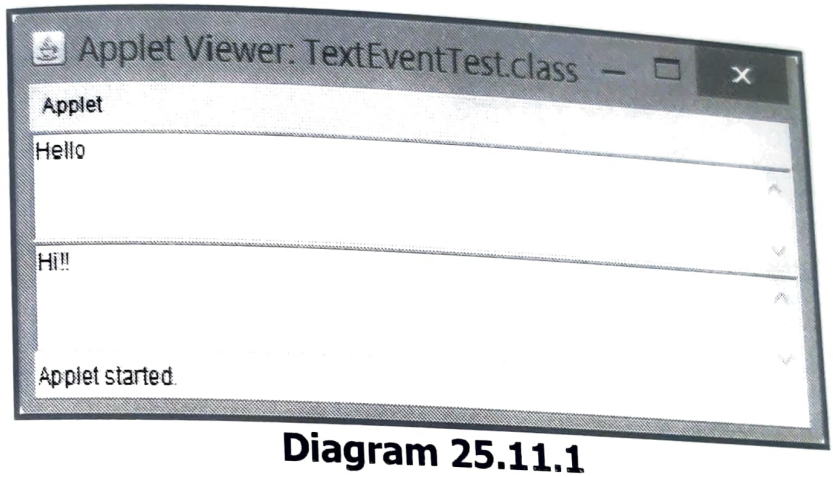

Diagram 25.11.1

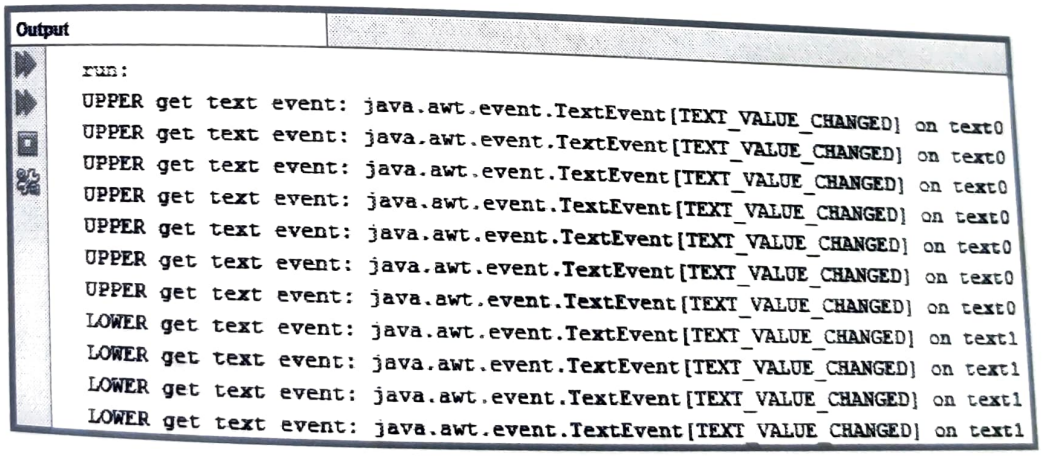

Diagram 25.11.2

Window Events

Window events are sent when a change happens to a window.

Here is the definition of WindowEvent:

```
public class WindowEvent extends ComponentEvent {
   public static final int WINDOW_ACTIVATED;
   public static final int WINDOW_CLOSED;
   public static final int WINDOW_CLOSING;
   public static final int WINDOW_DEACTIVATED;
   public static final int WINDOW_DEICONIFIED
   public static final int WINDOW_ICONIFIED;
   public static final int WINDOW_OPENED;
   public WindowEvent(Window source, int id);
   public Window getWindow();
   public String paramStrings();
}
```

Applets rarely need to be concerned with window events. Applications with GUIs are something concerned with window events.

The most common use for window events has to do with a strange feature of Java's Frame class. The frame's window decoration is drawn and maintained by the underlying window system, not by Java. No matter what the underlying system may be, there is always a mechanism provided for destroying a frame. For example, in Windows, clicking on the little X icon in the upper-right corner, destroys the frame. This window-description mechanism is not very destructive to a Java frame. All that happens is a window event is sent to the frame. The frame must explicitly respond to this event. The default behavior, if no event handler is set, is the same as the default behavior for any other Java event - The event is ignored.

Since users expect to be able to close a window [frame] from the window decoration, it is a good idea to equip all frames with a window event handler that destroys the frame on detection of a window-closing event. The code below does just that. Since there is only one frame, the event handler not only destroys the frame it also exits the application.

Example

The following code produces a very simple applet that demonstrates the use of KillableFrame.java:

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class KillableFrame extends Frame {
5     public static void main(String args[]) {
6        KillableFrame kfrm = new KillableFrame();
7        kfrm.setVisible(true);
8     }
9
10    public KillableFrame() {
11       setSize(250,250);
```

```
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    }
    @Override
    public void paint(Graphics Grph) {
        Grph.drawString("Click on X to Exit.", 20, 100);
    }
    @Override
    public void processWindowEvent(WindowEvent WinEvt) {
        if (WinEvt.getID() == WindowEvent.WINDOW_CLOSING) {
            setVisible(false);
            dispose();
            System.exit(0);
        }
    }
}
```

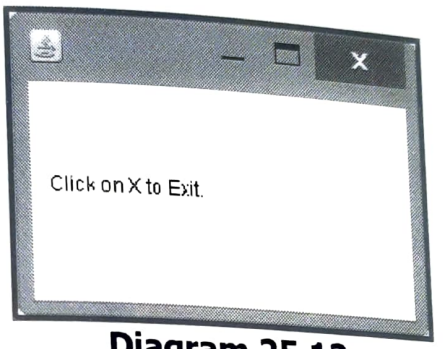

Diagram 25.12

Some of the above examples were executed using the NetBeans IDE, hence an **HTML file** is not required. However, if the examples needs to be executed using the Applet Viewer, then an HTML file is required to see the output.

Code spec: **[index.html]**

```
<html>
    <head>
        <title>Welcome To The World Of Swing</title>
    </head>
    <body>
        <applet code="<Class Name>.class" width="300" height="200"></applet>
    </body>
</html>
```

Just change the class name to execute the application

Most of the code remains the same except for the code attribute's value i.e. the class name. If required, change the width and height of <applet>.

Compile the Java file as follows:

```
<System Prompt:\> javac <Class Name>.java
```

Execute the class file as follows:

<System Prompt:\> appletviewer index.html

The Book CDROM holds the complete application source code for the following appli...

- Codespecs \ Section_05 \ Chapter25_CDS \ **MyButton.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **MyButtonTest.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **ButtonDelegateTest.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **ScrollBarTest.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **FocusEventTest.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **ItemEventTest.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **KeyEventTest.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **MouseEventTest.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **TextEventTest.java**
- Codespecs \ Section_05 \ Chapter25_CDS \ **KillableFrame.java**

The application can be directly used by compiling and executing it.

Chapter

23

SECTION IV: GRAPHIC USER INTERFACE

Abstract Window Toolkit [AWT]

Java applets are small .class files that run within a Java enabled Web Browser. Most Web browsers today have a built in JRE that is either active by default or can be set as active. A Web Browser with its JRE set to active is a Java enabled Web Browser. This is what makes an applet different from a standalone Java program. A Web Browser and JDK's Applet Viewer can be used to execute Java Applet .class file.

REMINDER

Both a Web Browser and the JDK's 'Applet Viewer' are GUI's. Before a Java Applet can be run in 'Applet Viewer' its .class file must be embedded in an HTML program between the <applet></applet> tags. Then the HTML program must be passed to 'Applet Viewer' for the Java Applet to run.

In order to display data within a GUI, the applet's .class file must use Java classes specifically designed for displaying information in a GUI environment. The classes are contained within java.awt package.

The AWT package provides:

- A complete set of UI components including windows, menus, buttons, check boxes, text fields, scroll bars and scrolling lists required for use in a UI environment
- Support for UI containers, which in turn can hold embedded containers or UI widgets
- An event system for managing system and user events
- Techniques and code spec for laying out UI components in such a way that permits platform independent UI design

The basis behind Java's AWT package is that a GUI is a set of nested components, starting from the outermost window all the way down to the smallest UI component held within. Components can include visual elements such as windows, menu bars, buttons and text fields or other containers, which in turn hold other components.

Window Fundamentals

Java's AWT defines windows according to a class hierarchy. The most commonly used windows are derived from **Panel**, which is used by applets and those derived from **Frame**, which is used to create a standard window. These windows have much of their functionality derived from their parent classes. Hence, to understand these two types of window objects it's necessary to have an understanding of their class hierarchies. Diagram 23.1 shows the class hierarchy for **Panel** and **Frame**.

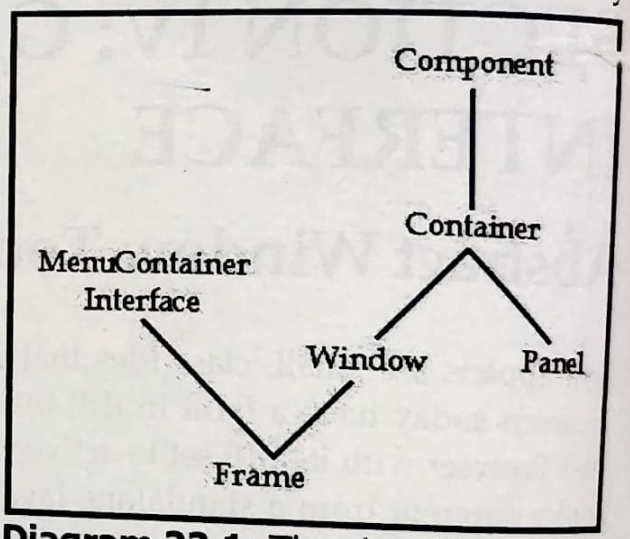

Diagram 23.1: The class hierarchy for Panel and Frame

Component

The **Component** class, which is right at the top of the AWT hierarchy, is an abstract class that encapsulates all of the attributes of a visual component.

All user interface elements displayed on the screen and which interact with a user are subclasses of **Component**. Component defines over a hundred public methods that are responsible for managing events such as positioning, sizing and repainting the window, handling mouse and keyboard input and so on. The **Component** object also registers and uses the current foreground color, background color and specific font used to display text.

Container

Container subclasses **Component**, additionally it has methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside a Container. This makes for a multileveled containment system. A **container** is responsible for positioning all the components it contains. It does this through the use of multiple types of layout managers.

Panel

Panel subclasses **Container**, it does not add any new methods to Container. It simply implements **Container**. A **Panel** can be conceptualized as a recursively nestable, screen component. **Panel** is the superclass of **Applet**. When screen output is required in an Applet, it is drawn on the surface of a Panel object bound to that Applet. A **Panel** is a window that does not contain a title bar, menu bar or border. This is why these objects are invisible when an applet runs within a browser. When an applet is run within JDK's applet viewer, the applet viewer provides the title and border.

Multiple components can be added to a **Panel** object. Once the components have been added, they can be positioned and resized manually using methods defined within Component.

Window

The **Window** class creates a top-level window. A **top-level window** is not contained within any other object. It sits directly on the desktop. Normally a subclass of **Window** called **Frame** is used to craft a Window. Window objects are not normally created directly.

Frame

Frame encapsulates **Window**. **Frame** is a subclass of **Window** and has a title bar, menu bar, borders and resizing corners. If a Frame object is created within an applet, it will display a warning message, **Warning: Applet Window**, to let a user know that they are working with an applet window. This message informs users that the window they are working with is an applet running within a Browser [or Applet viewer] and not by some software running on their computer. [An applet could masquerade as a host-based application and be used to capture Login ID's and passwords and other sensitive information without the user's knowledge and return this to a Web server for fraudulent use]. When a **Frame** window is created by any program other than a Java applet, a normal window is created.

Canvas

Canvas is not part of the hierarchy for applet or frame windows.

Canvas encapsulates a blank window, which can be drawn upon.

The following are major components in AWT:

- **Container:** Containers are generic AWT components that can contain other components [including other containers]. The most common form of container is Panel, which represents a container that can be displayed on the VDU. Applets are a form of panel [i.e. the Applet class is a sub class of the Panel class]
- **Canvas:** A canvas is a simple, blank, drawing surface. A canvas is excellent for painting images or performing other graphics operations
- **UI components:** These include buttons, lists, simple pop-up menus, check boxes, text boxes, text areas and other such elements used within a graphical user interface [GUI]
- **Window construction components:** These include windows, frames, menu bars and dialog boxes

Classes within java.awt are written and organized to mirror the abstract structure of containers, components and individual UI components. The root of AWT components is the Component class, which provides basic display and event handling features. The Container, Canvas, TextComponent classes and many other UI components are inherited from Component. The Panel and Window classes that can contain other AWT components are also inherited from Container. A partial AWT class hierarchy is shown in the diagram 23.2.

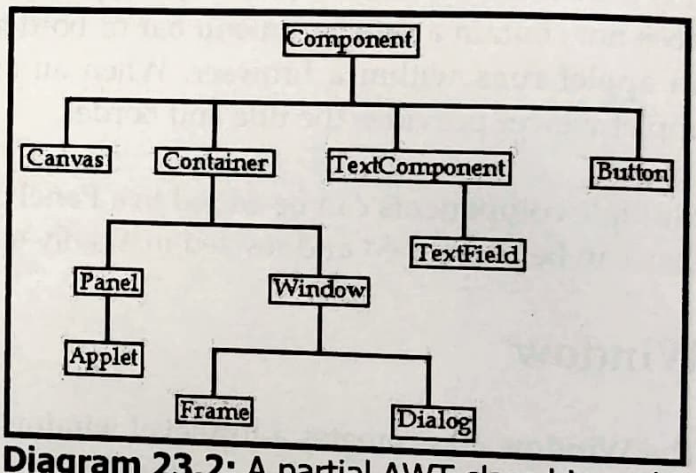

Diagram 23.2: A partial AWT class hierarchy

Basic User Interface Components

Some User Interface [UI] components within AWT are:

- Labels
- Buttons
- Check boxes
- Radio buttons
- Choice menus or Choice list
- Text fields
- Text Area
- Scrolling lists
- Scrollbars

For displaying any of the above UI components on a form [i.e. a UI], first instantiate the component and then add it to a panel or applet. Once done it is displayed on the VDU. To add a component to a panel such as an applet, add() of Container is used.

Solution:

```
public void init() {
   Button btnOk = new Button("OK");
   add(btnOk);
}
```

add() references the current applet, this means add this element to the current applet.

Labels

Label is the simplest form of UI component. Label is a string that is used to label other UI components. A **label** is defined as an uneditable string that acts as a description for other AWT components.

Use one of the following constructors to create a label:

- **Label():** Creates an empty label
- **Label(String):** Creates a label with the string, aligned left
- **Label(String, int):** Creates a label with the defined string and an integer that specifies alignment. The alignment numbers, stored in the form of integer constants in the Label class, are: **Label.RIGHT**, **Label.LEFT** and **Label.CENTER**. All these alignment numbers are recognized by the JRE as constants of the type int

REMINDER

Change the label's font using setFont() of the label class.

Syntax:

```
Label lblName = new Label("Label Name");
add(lblName);
```

Here,

- The first statement instantiates an object of the Label class and passes the string to be displayed as the label, as a parameter to the constructor of the class
- The second statement adds the Label object lblName to an applet using add() so that the label will be visible when the applet runs

Example

The following example displays the Label text at the left, right and at the center of an applet using the Label properties.

Solution:[LabelTest.java]

```
1  import java.applet.Applet;
2  import java.awt.Font;
3  import java.awt.Label;
4
5  public class LabelTest extends Applet {
6      @Override
7      public void init() {
8          setFont(new Font("Verdana", Font.ITALIC, 18));
9          Label lblLeft = new Label("Label left aligned", Label.LEFT);
10         add(lblLeft);
11         Label lblCenter = new Label("Label center aligned", Label.CENTER);
12         add(lblCenter);
13         Label lblRight = new Label("Label right aligned", Label.RIGHT);
14         add(lblRight);
15     }
16 }
```

Explanation:

A class is defined, which extends **java.applet.Applet**. **Applet** is the super class of any applet. The applet is embedded in a web page. **Applet** provides a standard interface between applets and the environment in which they execute. init() is called by the Browser that spawns the applet in the Browser window.

HINT

Subclass Applet and override its init() if the applet has initialization jobs to perform. For example: An applet with threads should use init() to create the threads and destroy() to kill the threads created.

Sets the font details such as:

- The name of the font i.e. **Verdana**
- The style of the font i.e. **Italic**
- The size of the font i.e. **18**

Three objects of the Label class are created. The text to be displayed as the label is passed as a parameter to the constructor of the class. add() adds the Label object to an applet to make the text visible in the browser.

Being an applet, this class file needs to be embedded into the HTML file, i.e. index.html.

The contents of HTML:[index.html]

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Label Test</title>
5      </head>
```