JAVA PROGRAMMING

COURSE NAME: MCA – 5TH SEMESTER

COURSE CODE: MCA18501CR

Teacher Incharge: *Dr. Shifaa Basharat*
Contact: *fazilishifaa@gmail.com*

EVENT HANDLING

Event handling is fundamental to Java programming because it is integral to the creation of applets and other types of GUI-based programs. Any program that uses a graphical user interface, such as a Java application written for Windows and applets are event driven. Events are supported by a number of packages, including **java.util**, **java.awt**, and **java.awt.event**. Most events to which a program will respond are generated when the user interacts with a GUI-based program. There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.

The Delegation Event Model

The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code.
In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Events Object

In the delegation model, an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.

Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. The general form is:

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such a method is :

public void add*Type*Listener(*Type*Listener *el*)
throws java.util.TooManyListenersException

Here, *Type* is the name of the event, and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is :

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**. The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners
A *listener* is an object that is notified when an event occurs. It has two major requirements.
First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.
The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Classes

The classes that represent events are at the core of Java’s event handling mechanism. At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its one constructor is shown here:

EventObject(Object *src*)

Here, *src* is the object that generates this event.
EventObject contains two methods: **getSource( )** and **toString( )**. The **getSource( )** method returns the source of the event. Its general form is shown here:

Object getSource( )

As expected, **toString( )** returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

int getID( )

Thus, we can say that
• **EventObject** is a superclass of all events.
• **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

The package **java.awt.event** defines many types of events that are generated by various user interface elements. They are mentioned in the following table:

Event Class	**Description**
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Read Details of each event class from your text book (Java: The Complete Reference (7th Edition) by Herbert Schildt).

Sources of Events:

Event Source	**Description**
Button	Generates action events when the button is pressed.
Check Box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll Bar	Generates adjustment events when the scroll bar is manipulated.
Text Components	Generates text events when the user enters a character.

Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

In addition to these graphical user interface elements, any class derived from Component, such as Applet, can generate events. For example, you can receive key and mouse events from an applet. (You may also build your own components that generate events.)

Event Listener Interfaces

Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.The commonly used listener interfaces and a brief description of the methods that they define are discussed below.

The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)
void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

The ItemListener Interface
This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface
This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered. For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released. The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

The MouseListener Interface
This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )**method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively. The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface
This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface
This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

The TextListener Interface

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

The WindowFocusListener Interface
This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

The WindowListener Interface
This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified( )** method is called. When a window is opened or closed, the **windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent *we*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

Using the Delegation Event Model

To use the delegation event model the following two steps need to be followed:
1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

To see how the delegation model works in practice, we will look at examples that handle two commonly used event generators: the mouse and keyboard.

Handling Mouse Events
To handle mouse events, we must implement the **MouseListener** and the **MouseMotionListener** interfaces. The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.
As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse

pointer as it is dragged. The two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint( )** to display output at the point of these occurrences.

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener {
String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse
public void init() {
addMouseListener(this);
addMouseMotionListener(this);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// show status
showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}
// Display msg in applet window at current X,Y location.
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}
}
```

In the above example **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Here the applet is both the source and the listener for these events. This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets. Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown below:

void addMouseListener(MouseListener *ml*)
void addMouseMotionListener(MouseMotionListener *mml*)

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both. The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

Handling Keyboard Events:

We can handle keyboard events, by using the same general architecture as that shown in the mouse event example in the preceding section. The difference is that we will have to implement the KeyListener interface.

When a key is pressed, a KEY_PRESSED event is generated. This results in a call to the keyPressed( ) event handler. When the key is released, a KEY_RELEASED event is generated and the keyReleased( ) handler is executed. If a character is generated by the keystroke, then a KEY_TYPED event is sent and the keyTyped( ) handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the keyPressed( ) handler.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
```

```
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
}
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
}
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}
// Display keystrokes.
public void paint(Graphics g) {
g.drawString(msg, X, Y);
}
}
```

If we want to handle the special keys, such as the arrow or function keys, we need to respond to them within the keyPressed( ) handler. They are not available through keyTyped( ). To identify the keys, we use their virtual key codes. For example, the following program applet outputs the name of a few of the special keys:

```
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/
public class KeyEvents extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
}
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
int key = ke.getKeyCode();
switch(key) {
case KeyEvent.VK_F1:
msg += "<F1>";
break;
case KeyEvent.VK_F2:
msg += "<F2>";
break;
case KeyEvent.VK_F3:
msg += "<F3>";
break;
case KeyEvent.VK_PAGE_DOWN:
msg += "<PgDn>";
break;
case KeyEvent.VK_PAGE_UP:
msg += "<PgUp>";
break;
case KeyEvent.VK_LEFT:
msg += "<Left Arrow>";
break;
case KeyEvent.VK_RIGHT:
msg += "<Right Arrow>";
break;
}
repaint();
}
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}
// Display keystrokes.
public void paint(Graphics g) {
g.drawString(msg, X, Y);
}
}
```

LOW LEVEL EVENT VS SEMANTIC EVENTS

Events can be divided into two groups: *low-level* events and *semantic* events. Low-level events represent window-system occurrences or low-level input. **Low level events** represent direct interaction with the user. Examples of low-level events include mouse and key events both of which result directly from user input.

Semantic events are dependent events i.e. they depend on low level events. Examples of semantic events include action and item events. A semantic event might be triggered by user input; for example, a button customarily fires an action event when the user clicks it, and a text field fires an action event when the user presses *Enter*. However, some semantic events are not triggered by low-level events, at all. For example, a table-model event might be fired when a table model receives new data from a database.

Whenever possible, you should listen for semantic events rather than low-level events. That way, you can make your code as robust and portable as possible. For example, listening for action events on buttons, rather than mouse events, means that the button will react appropriately when the user tries to activate the button using a keyboard alternative or a look-and-feel-specific gesture. When dealing with a compound component such as a combo box, it is imperative that you stick to semantic events, since you have no reliable way of registering listeners on all the look-and-feel-specific components that might be used to form the compound component.

Low Level Events Examples

Event Name	**Description**
ComponentEvent	A low-level event which indicates that a component moved, changed size, or changed visibility (also, the root class for the other component-level events).
ContainerEvent	A low-level event which indicates that a container's contents changed

	because a component was added or removed.
FocusEvent	A low-level event which indicates that a Component has gained or lost the input focus.
WindowEvent	A low-level event that indicates that a window has changed its status.

Semantic Events Examples

Event Name	**Event Descritpiton**
ActionEvent	A semantic event which indicates that a component-defined action occurred.
ItemEvent	A semantic event which indicates that an item was selected or deselected.
TextEvent	A semantic event which indicates that an object's text changed.

Adapter Classes

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. We can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which we are interested.
For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface. If we are interested in only mouse drag events, then we could simply extend **MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events for us. Following table lists the commonly used adapter classes in **java.awt.event** and the interface that each implements.

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentLIstener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. **AdapterDemo** extends **Applet**. Its **init( )** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument. **MyMouseAdapter** extends **MouseAdapter** and overrides the **mouseClicked( )** method.The other mouse events are silently ignored by code inherited from the **MouseAdapter** class. **MyMouseMotionAdapter** extends **MouseMotionAdapter** and overrides the **mouseDragged( )** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
addMouseMotionListener(new MyMouseMotionAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
AdapterDemo adapterDemo;
public MyMouseAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
adapterDemo.showStatus("Mouse clicked");
}
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
AdapterDemo adapterDemo;
public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
adapterDemo.showStatus("Mouse dragged");
}
}
```

Inner Classes

An *inner class* is a class defined within another class, or even within an expression. Inner classes can be used to simplify the code when using event adapter classes. To understand the benefit provided by inner classes, consider the applet shown in the following listing. It *does not* use an inner class. Its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed. There are two top-level classes in this program. **MousePressedDemo** extends **Applet**, and **MyMouseAdapter** extends **MouseAdapter**. The **init( )** method of **MousePressedDemo** instantiates **MyMouseAdapter** and provides this object as an argument to the **addMouseListener( )** method. A reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor. This reference is stored in an instance variable for later use by the **mousePressed( )** method. When the mouse is pressed, it invokes the **showStatus( )** method of the applet through the stored applet reference. In other words, **showStatus( )** is invoked relative to the applet reference stored by **MyMouseAdapter**.

```
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/
public class MousePressedDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
MousePressedDemo mousePressedDemo;
public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
this.mousePressedDemo = mousePressedDemo;
}
public void mousePressed(MouseEvent me) {
mousePressedDemo.showStatus("Mouse Pressed.");
}
}
```

The following listing shows how the preceding program can be improved by using an inner class. Here, **InnerClassDemo** is a top-level class that extends **Applet**. **MyMouseAdapter** is an inner class that extends **MouseAdapter**. Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed( )** method can call the **showStatus( )** method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass **MyMouseAdapter( )** a reference to the invoking object.

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/
public class InnerClassDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter());
}
class MyMouseAdapter extends MouseAdapter {
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}
}
}
```

Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name. An anonymous inner class can facilitate the writing of event handlers as discussed in this section. Consider the applet shown in the following listing.Its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed.

```
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/
public class AnonymousInnerClassDemo extends Applet {
public void init() {
```

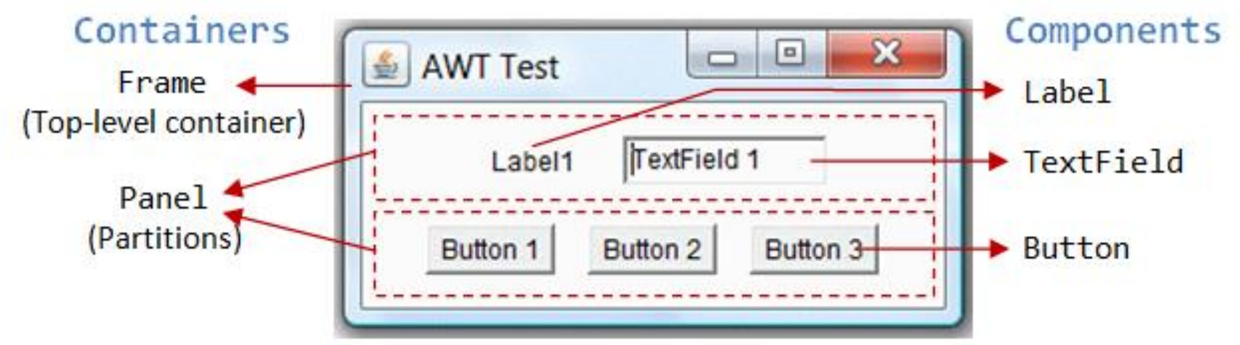
```
addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}
});
}
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**. The **init( )** method calls the **addMouseListener( )** method. Its argument is an expression that defines and instantiates an anonymous inner class. The syntax **new MouseAdapter( ) { ... }** indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends **MouseAdapter**. This new class is not named, but it is automatically instantiated when this expression is executed. Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **showStatus( )** method directly.

ADDING GUI ELEMENTS TO APPLETS

There are two types of GUI elements:

1. *Component*: Components are elementary GUI entities, such as Button, Label, and TextField.
2. *Container*: Containers, such as Frame and Panel, are used to *hold components in a specific layout* (such as FlowLayout or GridLayout). A container can also hold sub-containers.



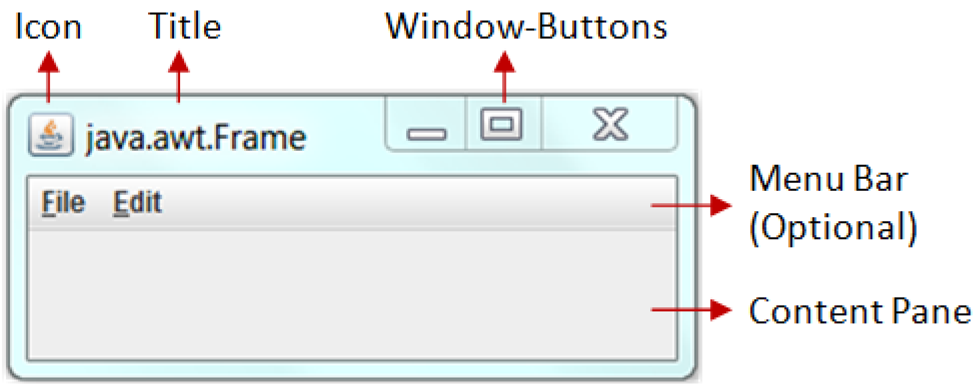
In the above figure, there are three containers: a `Frame` and two `Panels`. A `Frame` is the *top-level container* of an AWT program. A `Frame` has a title bar (containing an icon, a title, and the minimize/maximize/close buttons), an optional menu bar and the content display area. A `Panel` is a *rectangular area* used to group related GUI components in a certain layout. In the above figure, the top-level `Frame` contains two `Panels`. There are five components: a `Label` (providing description), a `TextField` (for users to enter text), and three `Buttons` (for user to trigger certain programmed actions).

In a GUI program, a component must be kept in a container. You need to identify a container to hold the components. Every container has a method called `add(Component c)`. A `container` (say `c`) can invoke `c.add(aComponent)` to add `aComponent` into itself.

AWT Container Classes

Each GUI program has a *top-level container*. The commonly-used top-level containers in AWT are Frame, Dialog and Applet:

- A Frame provides the "main window" for your GUI application. It has a title bar (containing an icon, a title, the minimize, maximize/restore-down and close buttons), an optional menu bar, and the content display area.



- An AWT Dialog is a *"pop-up window"* used for interacting with the users. A Dialog has a title-bar (containing an icon, a title and a close button) and a content display area.
- An AWT Applet (in package java.applet) is the top-level container for an applet, which is a Java program running inside a browser.

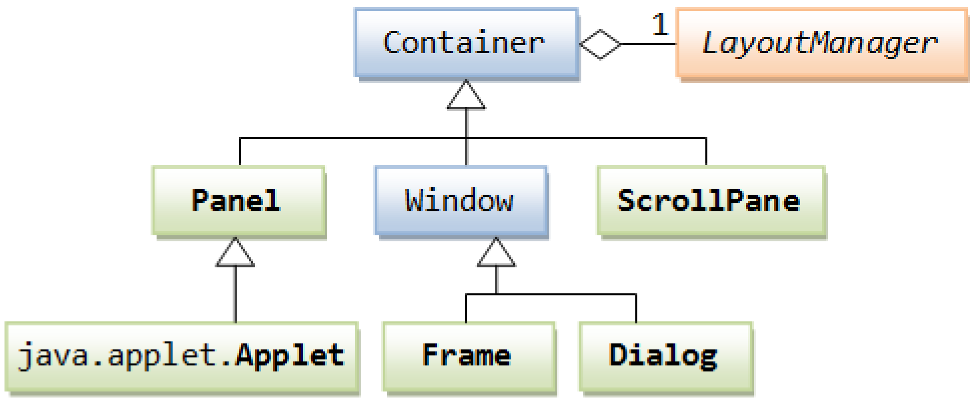
Secondary Containers: Panel and ScrollPane

Secondary containers are placed inside a top-level container or another secondary container. AWT provides these secondary containers:

- Panel: a rectangular box used to *layout* a set of related GUI components in pattern such as grid or flow.
- ScrollPane: provides automatic horizontal and/or vertical scrolling for a single child component.

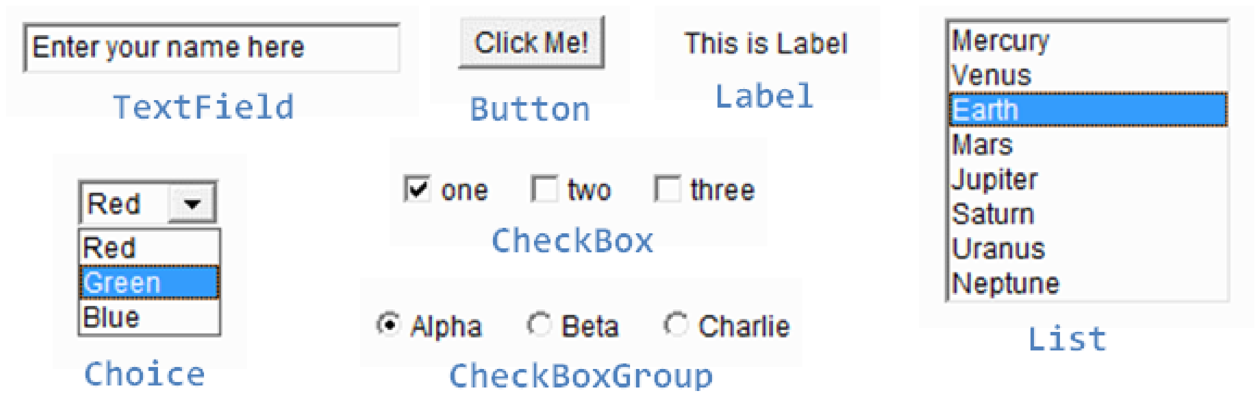
Hierarchy of the AWT Container Classes

The hierarchy of the AWT Container classes is as follows:



AWT Component Classes

AWT provides many ready-made and reusable GUI components in package java.awt. The frequently-used are: Button, TextField, Label, Checkbox, CheckboxGroup (radio buttons), List, and Choice, as illustrated below.



Constructing a Component and Adding the Component into a Container

Three steps are necessary to create and place a GUI component:

1. Declare the component with an *identifier* (*name*);
2. Construct the component by invoking an appropriate constructor via the new operator;
3. Identify the container (such as Frame or Panel) designed to hold this component. The container can then add this component onto itself via *aContainer*.add(*aComponent*) method. Every container has add(Component) method.

<u>Example</u>

Label lblInput; // Declare an Label instance called lblInput
lblInput = new Label("Enter ID"); // Construct by invoking a constructor via the new operator

```
add(lblInput);                    // this.add(lblInput) - "this" is typically a subclass of Frame
lblInput.setText("Enter password"); // Modify the Label's text string
lblInput.getText();               // Retrieve the Label's text string
```

Example:
Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

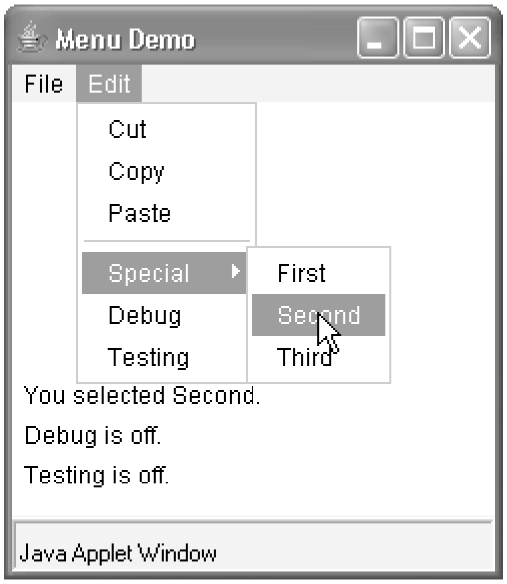
```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*

<applet code="MenuDemo" width=250 height=250>
</applet>
*/
// Create a subclass of Frame.
class MenuFrame extends Frame {
String msg = "";
CheckboxMenuItem debug, test;
MenuFrame(String title) {
super(title);
// create menu bar and add it to frame
MenuBar mbar = new MenuBar();
setMenuBar(mbar);
// create the menu items
Menu file = new Menu("File");
MenuItem item1, item2, item3, item4, item5;
file.add(item1 = new MenuItem("New..."));
file.add(item2 = new MenuItem("Open..."));
file.add(item3 = new MenuItem("Close"));
file.add(item4 = new MenuItem("-"));
file.add(item5 = new MenuItem("Quit..."));
mbar.add(file);
Menu edit = new Menu("Edit");
MenuItem item6, item7, item8, item9;
edit.add(item6 = new MenuItem("Cut"));
edit.add(item7 = new MenuItem("Copy"));
edit.add(item8 = new MenuItem("Paste"));
edit.add(item9 = new MenuItem("-"));
Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);
// these are checkable menu items
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);
mbar.add(edit);
// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
```

```
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
item11.addActionListener(handler);
item12.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
public void paint(Graphics g) {
g.drawString(msg, 10, 200);
if(debug.getState())
g.drawString("Debug is on.", 10, 220);
else
g.drawString("Debug is off.", 10, 220);
if(test.getState())
g.drawString("Testing is on.", 10, 240);
else
g.drawString("Testing is off.", 10, 240);
}
}
class MyWindowAdapter extends WindowAdapter {
MenuFrame menuFrame;
public MyWindowAdapter(MenuFrame menuFrame) {
this.menuFrame = menuFrame;
}
public void windowClosing(WindowEvent we) {
menuFrame.setVisible(false);
}
}
class MyMenuHandler implements ActionListener, ItemListener {
MenuFrame menuFrame;
public MyMenuHandler(MenuFrame menuFrame) {
this.menuFrame = menuFrame;
}
// Handle action events.
public void actionPerformed(ActionEvent ae) {
String msg = "You selected ";
String arg = ae.getActionCommand();
if(arg.equals("New..."))
msg += "New.";
else if(arg.equals("Open..."))
msg += "Open.";
else if(arg.equals("Close"))
msg += "Close.";
else if(arg.equals("Quit..."))
msg += "Quit.";
else if(arg.equals("Edit"))
msg += "Edit.";
else if(arg.equals("Cut"))
msg += "Cut.";
else if(arg.equals("Copy"))
msg += "Copy.";
else if(arg.equals("Paste"))
msg += "Paste.";
else if(arg.equals("First"))
msg += "First.";
else if(arg.equals("Second"))
msg += "Second.";
else if(arg.equals("Third"))
msg += "Third.";
else if(arg.equals("Debug"))
msg += "Debug.";
```

```
else if(arg.equals("Testing"))
msg += "Testing.";
menuFrame.msg = msg;
menuFrame.repaint();
}
// Handle item events.
public void itemStateChanged(ItemEvent ie) {
menuFrame.repaint();
}
}
// Create frame window.
public class MenuDemo extends Applet {
Frame f;
public void init() {
f = new MenuFrame("Menu Demo");
int width = Integer.parseInt(getParameter("width"));
int height = Integer.parseInt(getParameter("height"));
setSize(new Dimension(width, height));
f.setSize(width, height);
f.setVisible(true);
}
public void start() {
f.setVisible(true);
}
public void stop() {
f.setVisible(false);
}
}
```

Sample Output:



```
You selected Second.
Debug is off.
Testing is off.
```

Read Details of AWT from your text book (Java: The Complete Reference (7th Edition) by Herbert Schildt).