MODULE IV
ADVANCED FEATURES OF JAVA

Syllabus

- Event handling - Event Handling Mechanisms, Delegation Event Model, Event Classes, Sources of Events, Event Listener Interfaces, Using the Delegation Model.

EVENT

- When the user interacts with a GUI application, an event is generated. Examples of user events are clicking a button, selecting an item or closing a window. Events are represented as Objects in Java.
- Change in the state of an object is known as event i.e. event describes the change in state of source.
- Events are generated as result of user interaction with the graphical user interface components.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

- The events can be broadly classified into two categories:

Foreground Events

- Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

Background Events

- Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

EVENT HANDLING

- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- This mechanism have the code which is known as event handler that is executed when an event occurs.
- Java Uses the **Delegation Event Model** to handle the events.
- This model defines the standard mechanism to generate and handle the events.
- Let's have a brief introduction to this model.

Delegation Event Model

- There are **three participants** in event delegation model in Java;

  Event Source – the class which broadcasts the events

  Event Listeners – the classes which receive notifications of events

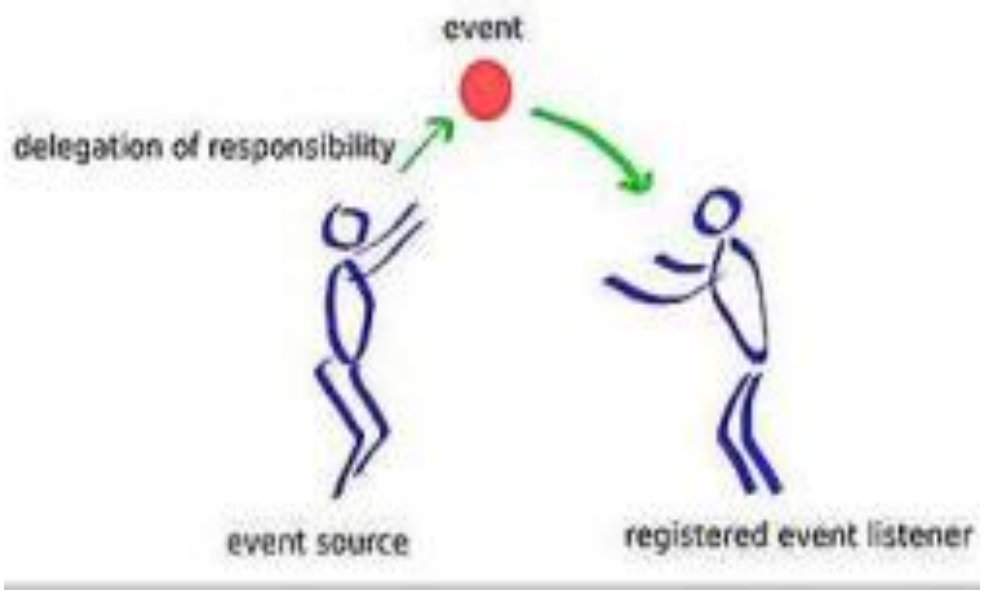
  Event Object – the class object which describes the event.
- Its concept is :
  - A *source* generates an event and sends it to one or more *listeners*.
  - The listener simply waits until it receives an event.
  - Once received, the listener processes the event and then returns.
- The listeners must register with a source in order to receive an event notification.

Advantage :

- The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event.
- The user interface element is able to delegate the processing of an event to the separate piece of code.
- In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification.
- This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

How Events are handled

- A source generates an Event and send it to one or more listeners registered with the source.
- Once event is received by the listener, they process the event and then return.
- Events are supported by a number of Java packages, like java.util, java.awt and java.awt.event



Event Model : Event

- An *event* is an object that describes a state change in a source or it is a type of signal to the program that something has happened.eg. Button pressed, window closed etc.
- It can be generated by
  - Personal interaction
    - Example : Pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse etc.
  - Not directly caused by personal interactions
    - Example : An event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, an operation is completed etc.

Event Model : Event Sources

- A *source* is an object that generates an event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- General form: public void add*Type*Listener(*Type*Listener *el*)

  Type - event name

  el - reference to the event listener.
- Example :

  addKeyListener() : The method that registers a keyboard event listener

  addMouseMotionListener() : The method that registers a mouse motion listener

- Sources may generate more than one type of event.
- **Multicasting:** When an event occurs, all registered listeners are notified and receive a copy of the event object. The notifications are sent only to listeners that register to receive them.
- **Unicasting**: Some sources may allow only one listener to register.

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

  throws java.util.TooManyListenersException

- Unregister a specific type of event

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

  Type - event name

  el - reference to the event listener.

  - Example :

    removeKeyListener( ) : To remove a keyboard listener

Event Model : Event Listener

- A *listener* is an object that is notified when an event occurs.

- It has two major requirements.
  - It must have been registered with one or more sources to receive notifications about specific types of events.

  - It must implement methods to receive and process these notifications. These methods are defined in a set of interfaces found in **java.awt.event**.
    - Example: **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.

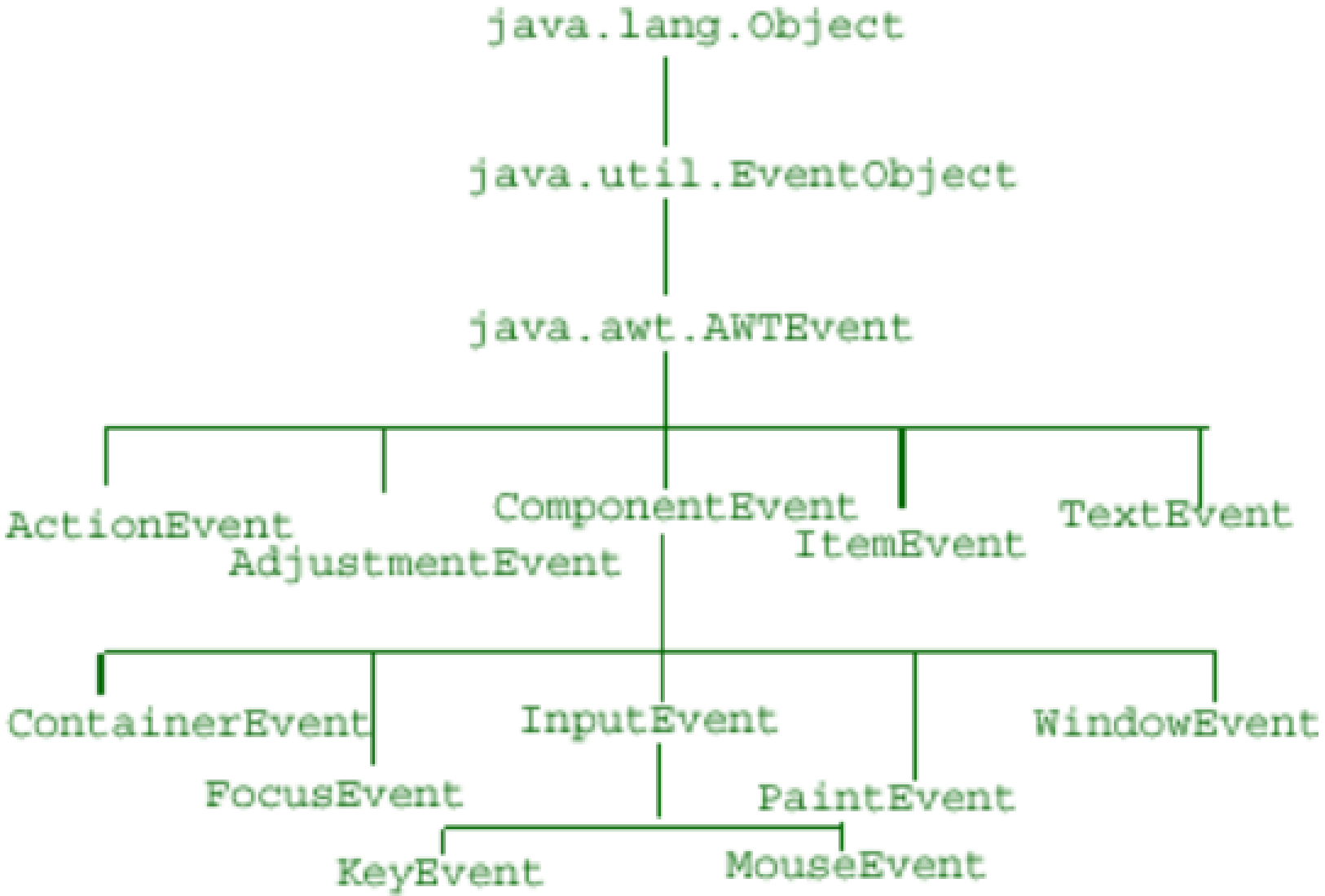
Event Classes

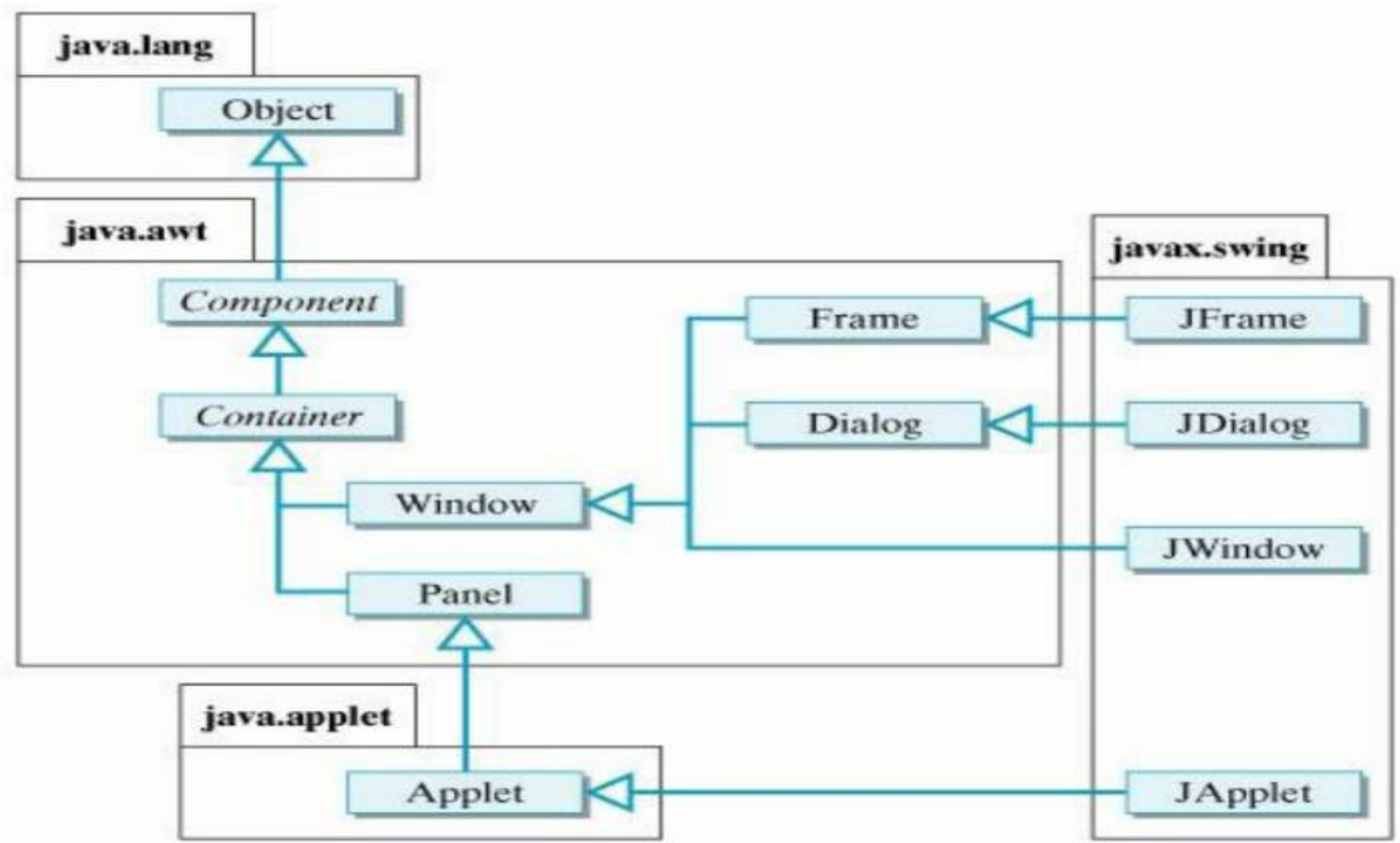
- The classes that represent events are the core of java's event handling. There are many event classes.
- **EventObject**
  - superclass for all events
  - defined inside **java.util** package
  - constructor : EventObject(Object *src*)

    src – object that generates event
  - two methods:
    - **Object getSource( ) :** returns the event source
    - **toString( ) :** returns the string equivalent of the event

- **AWTEvent**
  - subclass of **EventObject**.
  - superclass of all AWT-based events
  - defined within the **java.awt** package
  - **int getID( )** – returns an integer that determine the event type

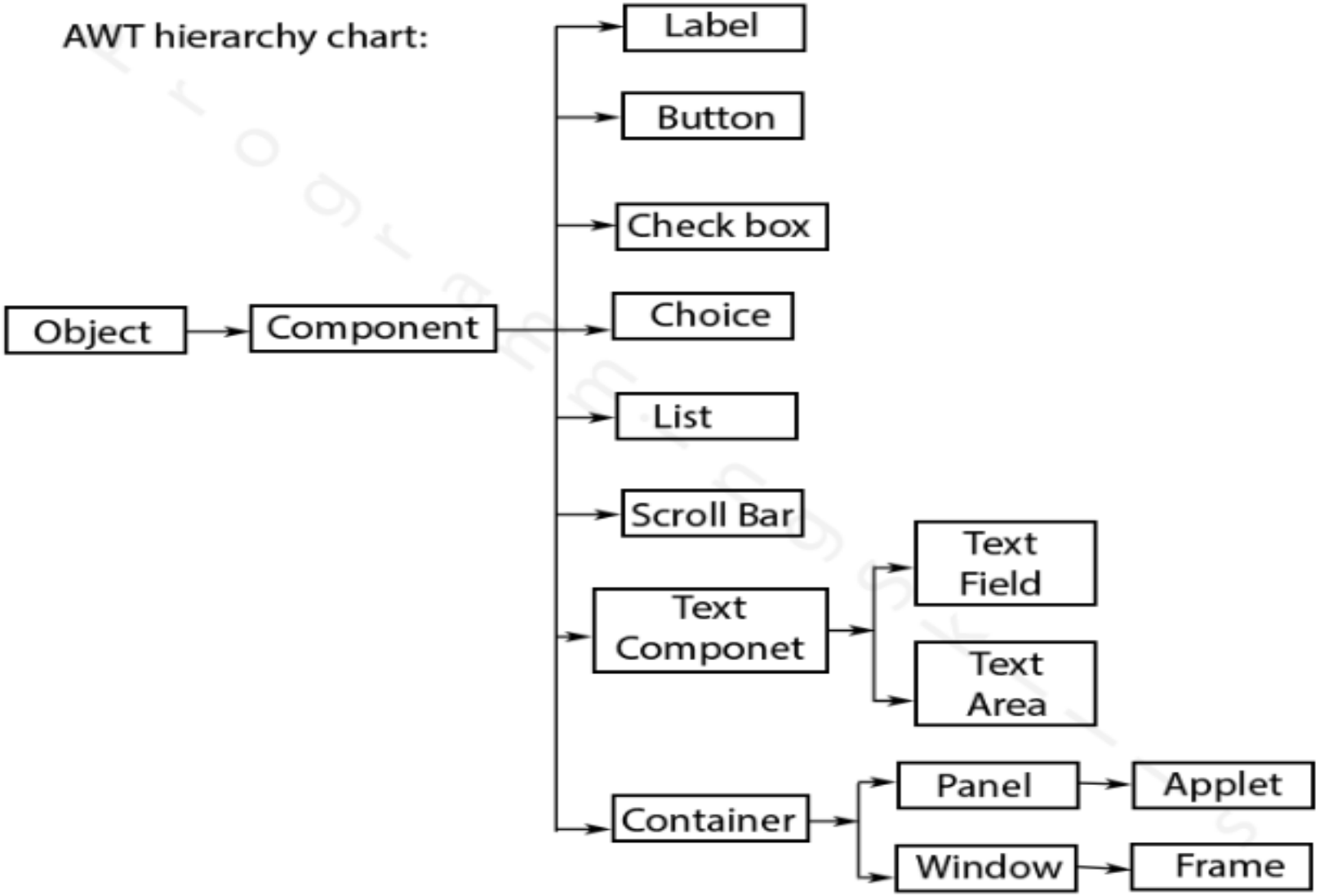
Event class hierarchy





Java AWT, Swing and Applet Components Hierarchy

AWT hierarchy chart:



AWT

- **AWT** stands for **Abstract Window Toolkit**.
- It is a platform dependent API for creating Graphical User Interface (GUI) for java programs.
- **Why AWT is platform dependent?** Java AWT calls native platform (Operating systems) subroutine for creating components such as textbox, checkbox, button etc. In simple, an application build on AWT would look like a windows application when it runs on Windows, but the same application would look like a Mac application when runs on Mac OS.
- AWT is rarely used now days because of its platform dependent and heavy-weight nature.

Components and containers

- All the elements like buttons, text fields, scrollbars etc are known as components.
- In AWT we have classes for each component as shown in the above diagram.
- To have everything placed on a screen to a particular position, we have to add them to a container.
- A container is like a screen wherein we are placing components like buttons, text fields, checkbox etc.
- In short a container contains and controls the layout of components.
- A container itself is a component (shown in the above hierarchy diagram) thus we can add a container inside container.

Types of containers:

As explained above, a container is a place wherein we add components like text field, button, checkbox etc. There are four types of containers available in AWT: Window, Frame, Dialog and Panel. As shown in the hierarchy diagram above, Frame and Dialog are subclasses of Window class.

- **Window:** An instance of the Window class has no border and no title
- **Dialog:** Dialog class has border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.
- **Panel:** Panel does not contain title bar, menu bar or border. It is a generic container for holding components. An instance of the Panel class provides a container to which to add components.
- **Frame:** A frame has title, border and menu bars. It can contain several components like buttons, text fields, scrollbars etc. This is most widely used container while developing an application in AWT.

Creating a Frame

1) By extending Frame class
2) By creating the instance of Frame class

Methods

public void add(Component c)

public void setSize (int width, int height)

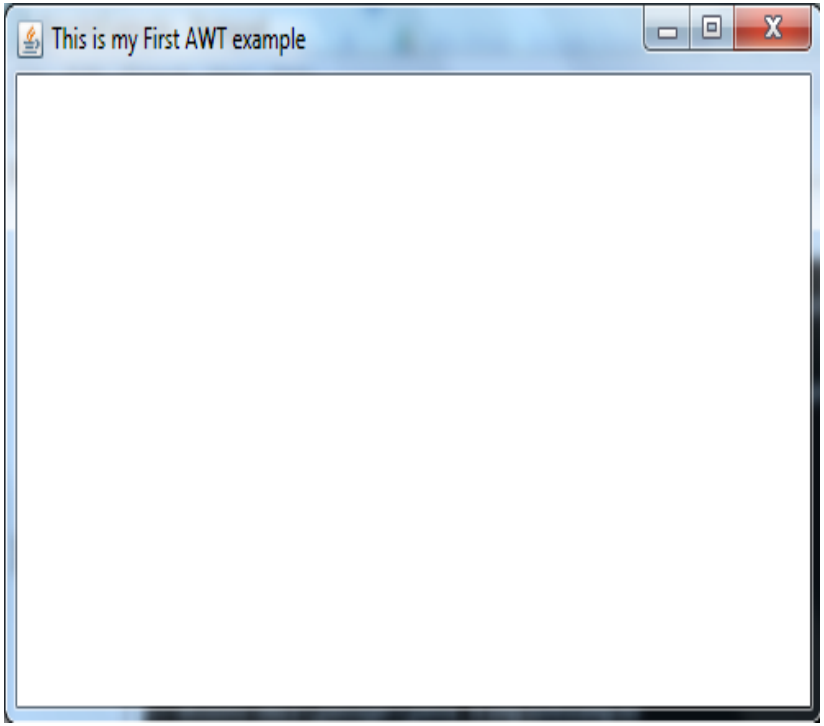
public void setLayout(LayoutManager m)

public void setVisible(boolean status)

Example 1: Creating Frame By extending Frame class

```
import java.awt.*;
public class Frame1 extends Frame{
Frame1(){
setSize(500,300);
setTitle("This is my First AWT example");
 setLayout(null);
 setVisible(true);
    }
 public static void main(String args[]){
        Frame1 f=new Frame1();
    }
}
```

Output

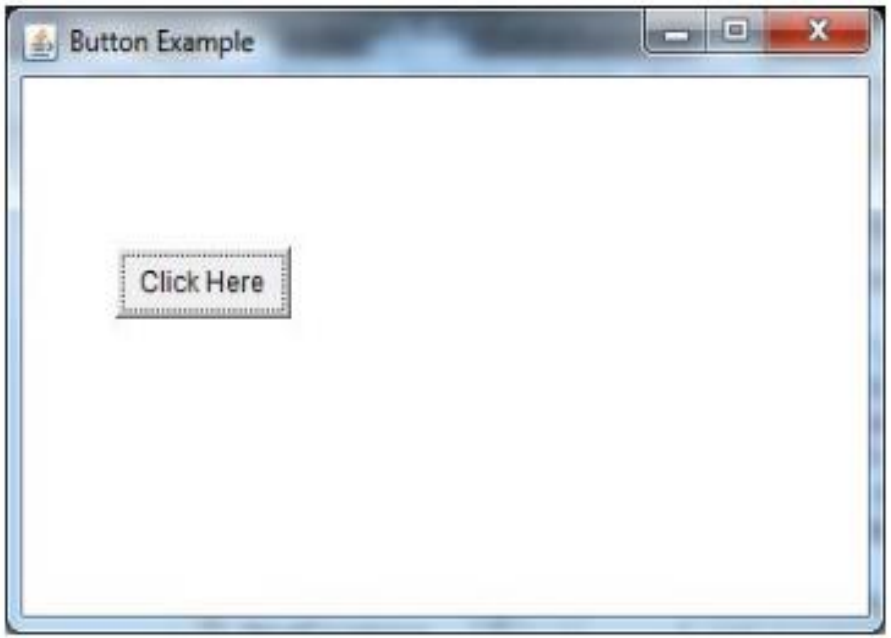


Example 2: Creating Frame using instance of Frame class

```
import java.awt.*;
public class ButtonExample {
public static void main(String[] args) {
    Frame f=new Frame("Button Example");
    Button b=new Button("Click Here");
    b.setBounds(50,100,80,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:

Output:



Example 3

Frame by inheritance

```
import java.awt.*;
import java.awt.event.*;
class Frcheck  extends Frame
{
 TextField t=new TextField(20);
 Button b=new Button("ok");
public Frcheck()
 {
   setSize(300,300);
   setVisible(true);
   setLayout(new FlowLayout());
   add(t);
   add(b);
 }
```

```
public static void main(String args[])
  {
    Frcheck f1=new Frcheck();
  }
}
```

Event classes and interface

Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved, clicked, pressed or released and also when it enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

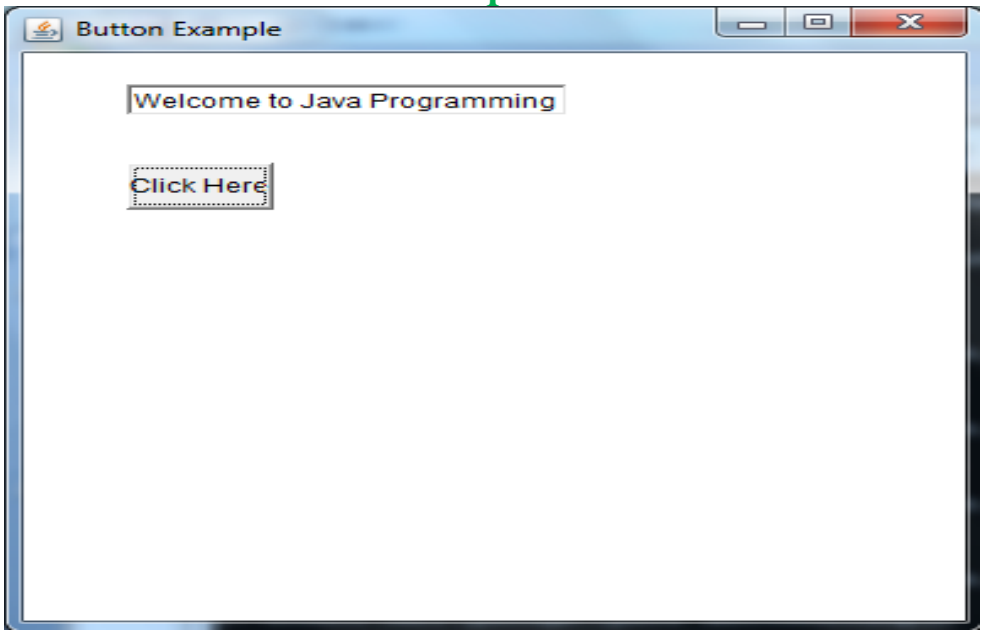
Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces.
- These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
- If we do not implement the predefined interfaces then your class can not act as a listener class for a source object.

Java AWT Button Example with ActionListener

```
import java.awt.*;
import java.awt.event.*;
public class ButtonExample implements ActionListener{
Frame f=new Frame("Button Example");
  TextField tf=new TextField();
   Button b=new Button("Click Here");
    ButtonExample()        {
    f.add(b);   f.add(tf);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    tf.setBounds(50,50, 180,20);
    b.setBounds(50,100,60,30);
    b.addActionListener(this);    }
public void actionPerformed(ActionEvent e){
        tf.setText("Welcome to Java Programming");
     }
public static void main(String[] args) {
ButtonExample b=new ButtonExample();
   } //close main      } //close ButtonExample
```

Output



SOURCES OF EVENT

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.

EVENT LISTENER INTERFACES

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

- **The ActionListener Interface**
  - Defines actionPerformed() method that is invoked when an action event occurs.

    void actionPerformed(ActionEvent *ae*)

- **The AdjustmentListener Interface**
  - Defines adjustmentValueChanged method that is invoked when an adjustment event occurs.

    void adjustmentValueChanged(AdjustmentEvent *ae*)

- **The ComponentListener Interface**
  - Defines four methods that is invoked when a component is resized, moved, shown or hidden

    void componentResized(ComponentEvent *ce*)

    void componentMoved(ComponentEvent *ce*)

    void componentShown(ComponentEvent *ce*)

    void componentHidden(ComponentEvent *ce*)

- **The ContainerListener Interface**
  - Defines two methods. Respective methods are called when a component is added or removed from a container.

    void componentAdded(ContainerEvent *ce*)

    void componentRemoved(ContainerEvent *ce*)

- **The FocusListener Interface :**
  - defines two methods. Respective methods are called when keyboard obtains and loses focus

    void focusGained(FocusEvent *fe*)

    void focusLost(FocusEvent *fe*)

- **The ItemListener Interface**

    void itemStateChanged(ItemEvent *ie*)

- **The KeyListener Interface**

    void keyPressed(KeyEvent *ke*)

    void keyReleased(KeyEvent *ke*)

    void keyTyped(KeyEvent *ke*)

- **The MouseListener Interface**

  void mouseClicked(MouseEvent *me*)

  void mouseEntered(MouseEvent *me*)

  void mouseExited(MouseEvent *me*)

  void mousePressed(MouseEvent *me*)

  void mouseReleased(MouseEvent *me*)

- **The MouseMotionListener Interface**

  void mouseDragged(MouseEvent *me*)

  void mouseMoved(MouseEvent *me*)

- **The MouseWheelListener Interface**

  void mouseWheelMoved(MouseWheelEvent *mwe*)

- **The TextListener Interface**

  void textChanged(TextEvent *te*)

- **The WindowFocusListener Interface**

  void windowGainedFocus(WindowEvent *we*)

  void windowLostFocus(WindowEvent *we*)

- **The WindowListener Interface**

  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*)

Applets

- **Applet** is a special type of program that is embedded in the webpage to generate the dynamic content.
- There are two types of applets:
  - The first type use the Abstract Window Toolkit (AWT) to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created
  - The second type of applets are those based on the Swing class **JApplet. Swing applets use** the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT

Application Program Vs Applet Program

- Applet does not use main() method for initiating the execution of a code. Applet, when loaded, automatically call certain methods of Applet class to start and execute the applet code.
- Applets are not stand-alone programs. Instead, they run within either a web browser or using an applet viewer.
- Applets cannot read from or write into a file in the local system.
- Applets cannot communicate with each other server on the network.
- Applets are restricted from using libraries from other languages such as C or C++.

Applets : Example

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


public class SimpleApplet extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString("A Simple Applet", 20, 20);
        }
}
```

Abstract Window Toolkit, for GUI

applet , for Applet class

Always (public and extends Applet)

Called to redraw applet
Over ridden method[class Applet]

- Applet Viewer is used to execute applets in this section.
- Comment is included at the head of the Java source code file that contains the APPLET tag.

```
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
```

- Do the following three steps:
  1. Edit a Java source file with applet tag.
  2. Compile the program.
  3. Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the applet tag within the comment and execute the applet.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString("A Simple Applet", 20, 20);
        }

}
```

Source code is documented with prototype of HTML statements

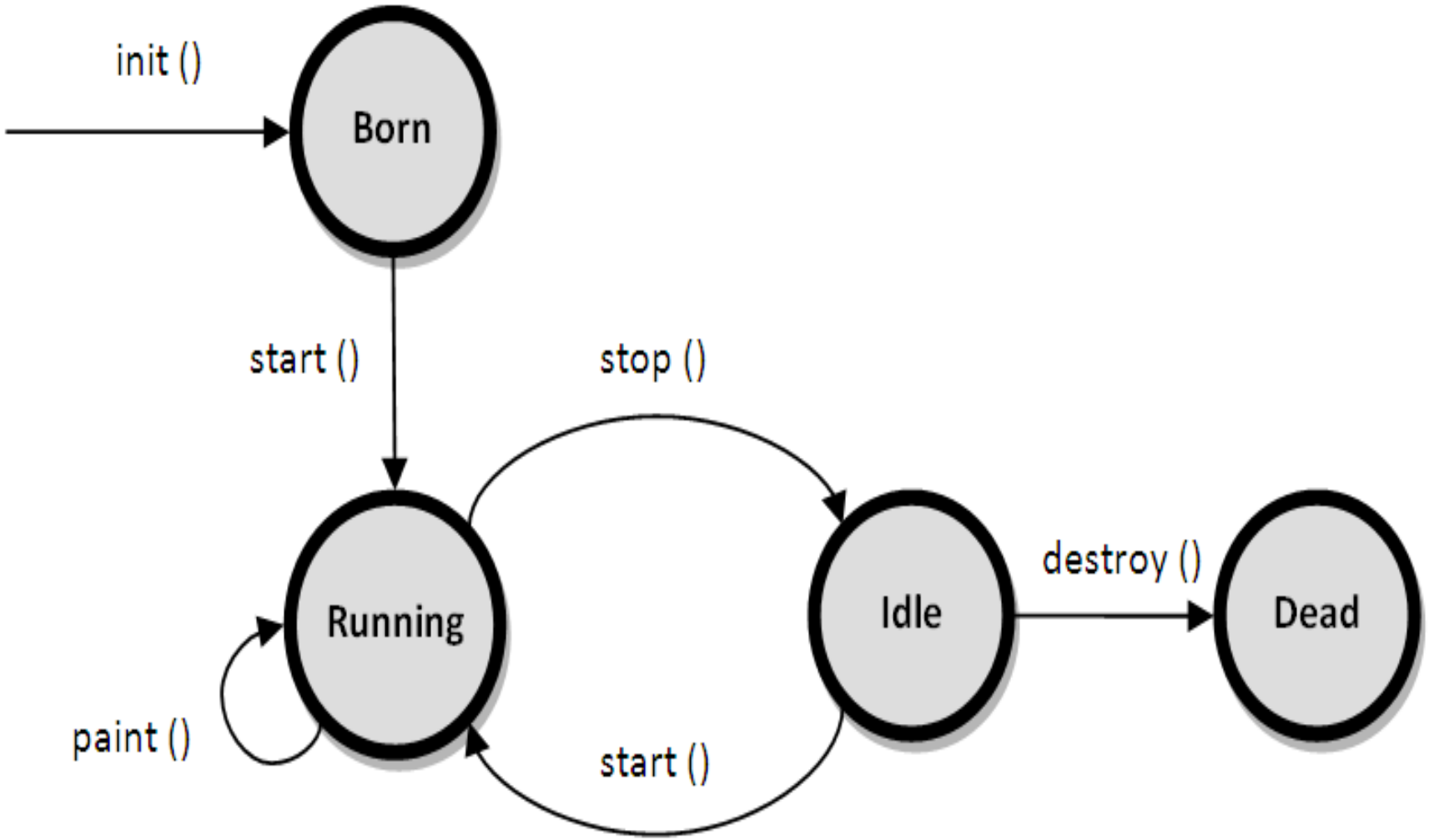
Output

- Save file as SimpleApplet.java
- Compile: javac SimpleApplet.java
- Execute: appletviewer SimpleApplet.java

Applet Architecture

- An applet is a window-based program.
- Applets are event driven. An applet waits until an event occurs.
- The user interacts with the applet when he or she wants.
- The AWT notifies the applet about an event by calling an event handler that has been provided by the applet.
- The applet must take appropriate action and then quickly return control to the AWT.
- Example :
  - If the user clicks a mouse inside the applet's window, a mouse-clicked event is generated
  - If the user presses a key while the applet's window has input focus, a keypress event is generated.

Applet Life Cycle



1. Born on initialization state :

- Applet enters the initialization state when it is first loaded.
- This is achieved by calling **init().**
- This occurs only once in the applet's life cycle.
- At this stage, we may do the following
  - Create objects needed by the applet
  - Initialize variables
  - Load images or fonts.
  - Setup colors.

2. **Running state** :

- Applet enters this state when the system calls the **start()** method
- This occurs automatically after the applet is initialized(**init()**)
- Starting can also occur if the applet is already in stopped(Idle) state.
- **start()** can be called more than once.
- **start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

3. Idle state or Stopped state :

- An applet becomes idle when it is stopped from running. We can do so by calling stop() explicitly.
- Stopping occurs automatically when we leave the page containing the currently running applet.
  - Example : when it goes to another page.
- If the user returns to the page, we can restart them by calling **start( ).**

4. Dead or destroyed state :

- An applet is said to be dead when it is removed from memory.
- This occurs by invoking **destroy().**
- At this point, we should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.
- This occurs only once in the applet life cycle.

5. Display state:

- Applet moves to this state whenever it has to perform some output operations on the screen.
- This happens immediately after the applet enters into the running state.
- **paint()** is called to accomplish this task.

- The **paint( )** method is called each time our applet's output must be redrawn.

- This situation can occur for several reasons. For example,
  - the window in which the applet is running may be overwritten by another window and then uncovered.
  - the applet window may be minimized and then restored.

- The **paint( )** method has one parameter of type **Graphics**. This parameter describes the graphics environment in which the applet is running.

- Applets override a set of methods
  - **init()**, **start()**, **stop()**, and **destroy()** : These are defined by **Applet**
  - **paint()** : It is defined by the AWT **Component** class.

- When an applet begins, the AWT calls the following methods, in this sequence:
  1. **init( )**
  2. **start( )**
  3. **paint( )**

- When an applet is terminated, the following sequence of method calls takes place:
  1. **stop( )**
  2. **destroy( )**

Applet Display Methods

- To output a string to an applet :
  - **drawString( )**, which is a member of the **Graphics** class

    void drawString(String *message*, int *x*, int *y*)

  message is the string to be output beginning at *x*,*y*. In a Java window, the upper-left corner is location 0,0.

- To set the background color of an applet's window, use **setBackground( )**.

  void setBackground(Color *newColor*)

- To set the foreground color, use **setForeground( )**.

  void setForeground(Color *newColor*)

  where *newColor* specifies the new color.

- The class **Color** defines the constants to specify colors:

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

- You can obtain the current settings for the background and foreground colors by calling **getBackground( ) and getForeground( ).**

  Color.getBackground()

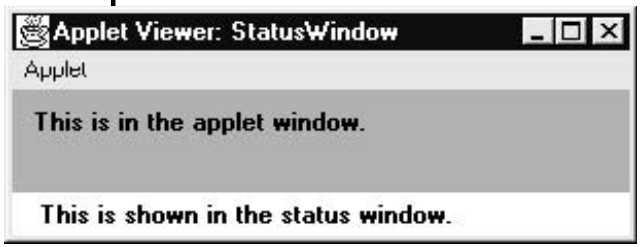
  Color getForeground()

- The default foreground color is black. The default background color is light gray.

Status Window

```
import java.awt.*;
import java.applet.*;
/*
        <applet code="StatusWindow" width=300 height=50>
        </applet>
*/
public class StatusWindow extends Applet
{
        public void init()
        {
                setBackground(Color.cyan);
        }
        public void paint(Graphics g)
        {
                g.drawString("This is in the applet window.", 10, 20);
                showStatus("This is shown in the status window.");
        }
}
```

Output



```
/*  <applet code="ButtonExample2" width=400 height=400>  </applet>  */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ButtonExample2 extends Applet implements ActionListener{
TextField tf=new TextField(20);
Button b1=new Button("OK");
Button b2=new Button("Clear");
   public void init(()            {
   add(b1);add(b2);add(tf);
   b1.addActionListener(this);
   b2.addActionListener(this);     }
public void actionPerformed(ActionEvent e){
  if(e.getSource()==b1)
  tf.setText("Welcome");
  else
  tf.setText(" ");      }     }
```