Abstract Window Toolkit

Java.awt PACKAGE

- The Abstract Window Toolkit (AWT) contains numerous classes and methods that allow you to create and manage GUI applications

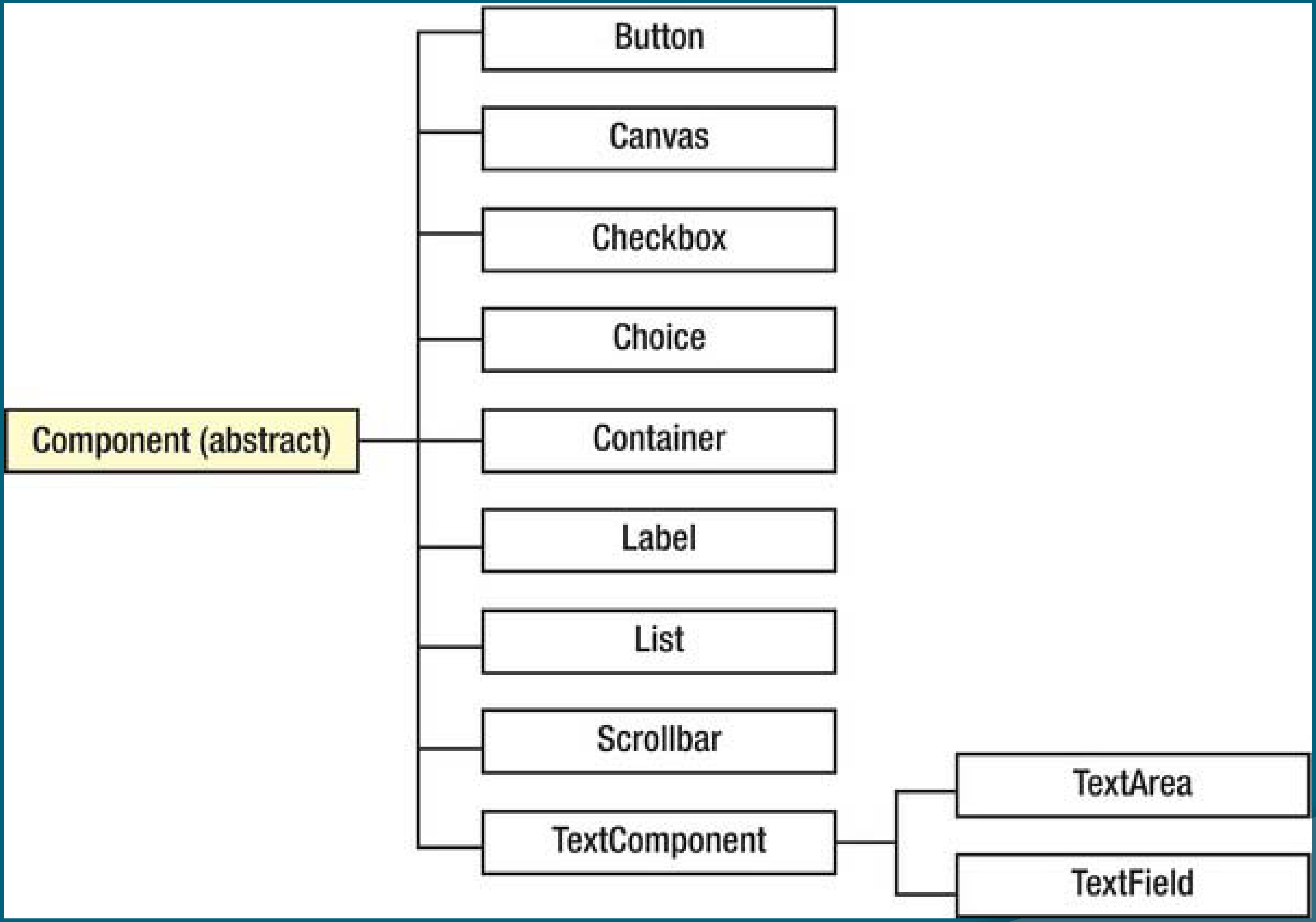
- The AWT classes are contained in the **java.awt** package.

- This package helps you to create different types of windows and other GUI components that can be placed on the windows.

AWT CONTROLS

- Controls are components that allow a user to interact with your application in various ways
- The AWT supports the following types of controls:
  - **Labels**
  - **Push buttons**
  - **Check boxes**
  - **Choice lists**
  - **Lists**
  - **Scroll bars**
  - **Text editing**
- These controls are subclasses of Component

Component class hierarchy



WINDOWS FUNDAMENTALS

- Buttons, labels, textfields, and other components cannot be placed directly on the screen

- They need to be placed in a container window that is placed directly on the screen

- The two most common windows are:

  - derived from **Panel**, which is used by applets,
  - derived from **Frame**, which creates a standard window

WINDOWS FUNDAMENTALS

- **Panel** is a window that does not contain a title bar, menu bar, or border.

- When you run an applet using an applet viewer, the applet viewer provides the title and border.

- **Frame** is a subclass of Window and has a title bar, menu bar, borders, and resizing corners. Used to create stand-alone GUI applications

Working with Frame Windows

- It used to create child windows within applets, and top-level or child windows for applications.

CONSTRUCTORS:

- Frame( )
- Frame(String title)   // title of the window

METHODS:

- void setSize(int Width, int Height)
- void setVisible(boolean visibleFlag)
- void setTitle (String newTitle)

FRAMES

When working with Frame objects, there are basically three steps involved to get a Frame window to appear on the screen:

1. Instantiate the Frame object
2. Give the Frame object a size using setSize()
3. Make the Frame appear on the screen by invoking setVisible(true).

ADDING & REMOVING CONTROLS

- To include a control in a window, you must add it to the window.

- First create an instance of the desired control and then add it to a window by calling **add( ),** which is defined by **Container.**

```
Component add(Component c)
```

- To remove a control from a window:

```
void remove(Component obj)
void removeall()
```

RESPONDING TO CONTROLS

- Except for labels, which are passive controls, all controls generate events when they are accessed by the user.
- In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor.
- Once a listener has been registered, events are automatically sent to it

LABELS

- A label is an object of type Label, and it contains a string, which it displays.
- Labels are passive controls that do not support any interaction with the user.
- **Constructors:**

```
Label( )
Label(String str)
Label(String str, int how)
```

- The value of how must be one of these three constants: **Label.LEFT, Label.RIGHT, or Label.CENTER.**

LABELS

Methods:

void setText(String str)

String getText()

void setAlignment(int how)

int getAlignment()

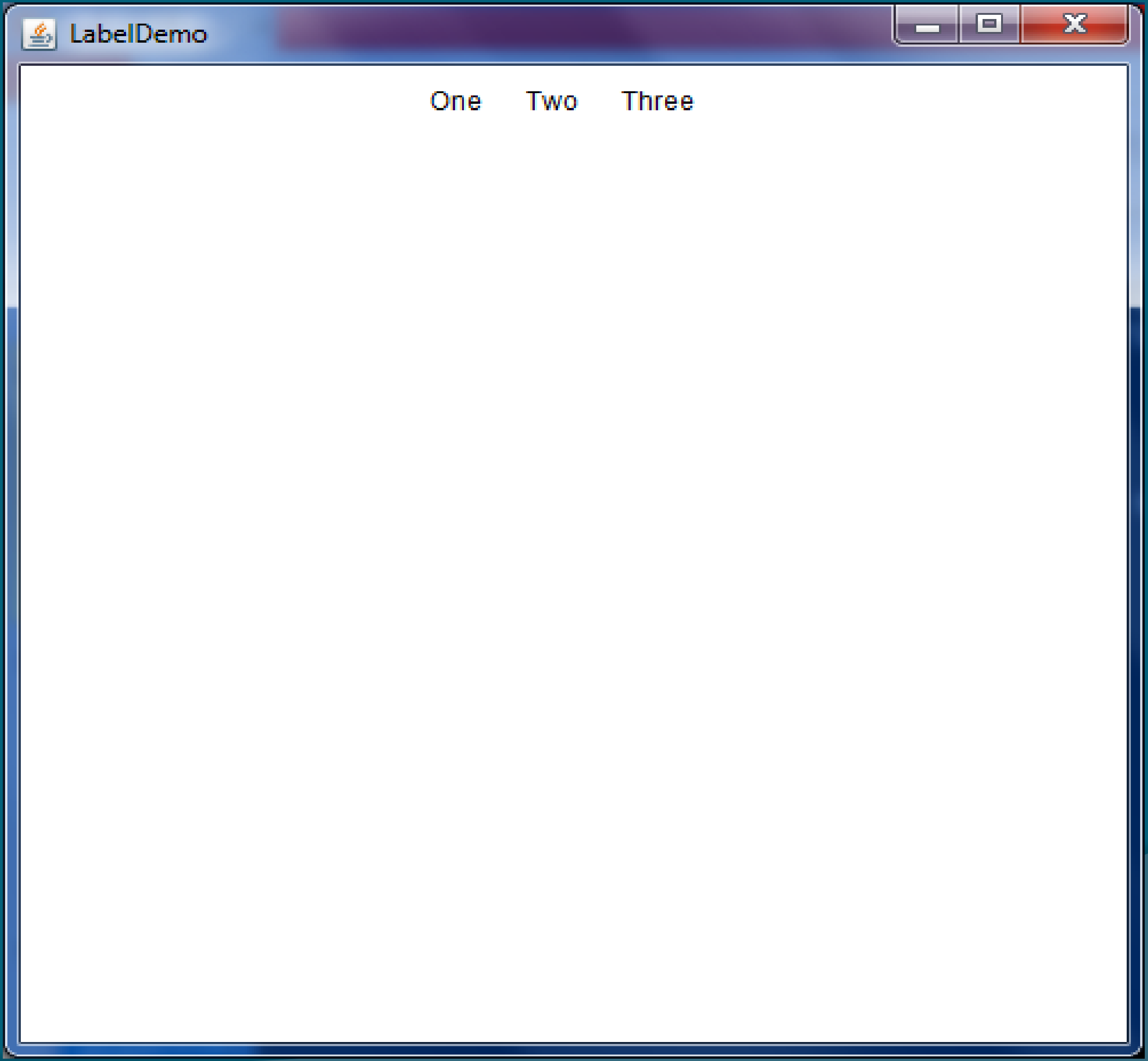
LABELS

```
import java.awt.*;

class LabelDemo extends Frame
{
 LabelDemo()
   {
   super("LabelDemo");
setLayout(new FlowLayout());
   setSize(200,200);
   Label one = new Label("One");
   Label two = new Label("Two");
   Label three = new Label("Three");
   add(one);
   add(two);
   add(three);
   setVisible(true);
   }
}
```

LABELS

```
class Labl
{

  public static void main (String args[])
  {

    LabelDemo l1=new LabelDemo();
  }
}
```

LABELS



One Two Three

BUTTONS

- It is the most widely used control that contains a label and that generates an event when it is pressed

CONSTRUCTORS

Button( )

Button(String str)

METHODS

void setLabel(String str)

String getLabel( )

BUTTONS

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
  /*
  <applet code="ButtonDemo" width=250 height=150>
  </applet>
  */
public class ButtonDemo extends Applet implements ActionListener {
String msg = "";
Button yes, no, maybe;
public void init() {
     yes = new Button("Yes");
     no = new Button("No");
     maybe = new Button("Undecided");
```

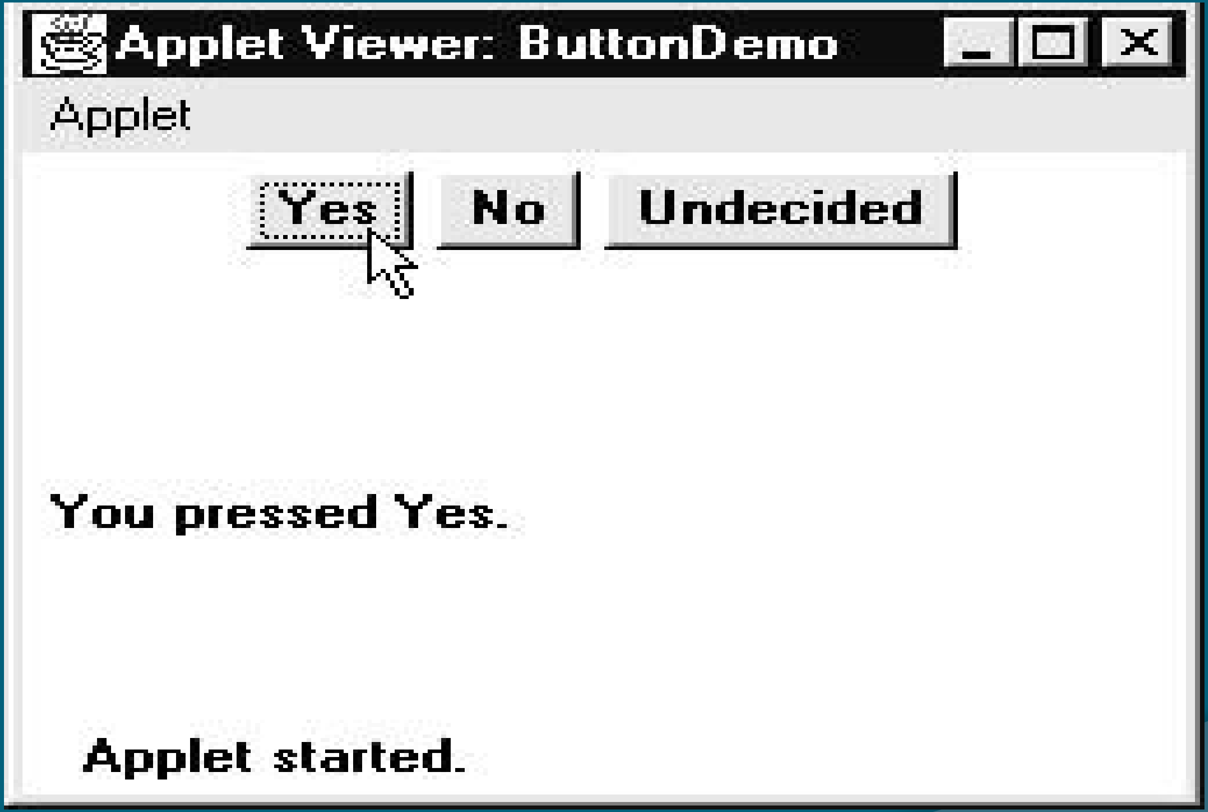
BUTTONS

```
    add(yes);
    add(no);
    add(maybe);


    yes.addActionListener(this);
    no.addActionListener(this);
    maybe.addActionListener(this);
  }
public void actionPerformed(ActionEvent ae)
{
   String str = ae.getActionCommand();
   if(str.equals("Yes")) {
        msg = "You pressed Yes.";
   }
```

BUTTONS

```
    else if(str.equals("No")) {
            msg = "You pressed No.";
    }
    else {
        msg = "You pressed Undecided.";
    }
    repaint();
}
  public void paint(Graphics g)
  {
     g.drawString(msg, 6, 100);
  }
}
```

BUTTONS



```
You pressed Yes.

Applet started.
```

CHECK BOXES

- A *check box* is a control that is used to turn an option on or off.

- It consists of a small box that can either contain a check mark or not. There is a label associated with each check box

- **CONSTRUCTORS**

```
Checkbox( )
Checkbox(String str)
Checkbox(String str, boolean on)
Checkbox(String str, boolean on, CheckboxGroup cbGroup)
```

CHECK BOXES

METHODS

boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)

CHECK BOXES:EXAMPLE

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*<applet code="CheckBoxDemo" width=250 height=300>
</applet>
*/
public class CheckBoxDemo extends Applet implements ItemListener
{
    String msg = "";
    Checkbox Win98, winNT, Win7;
    public void init() {
    Win98 = new Checkbox("Windows 98/XP", null, true);
    winNT = new Checkbox("Windows NT/2000");
    Win7  = new Checkbox("Windows 7");
```

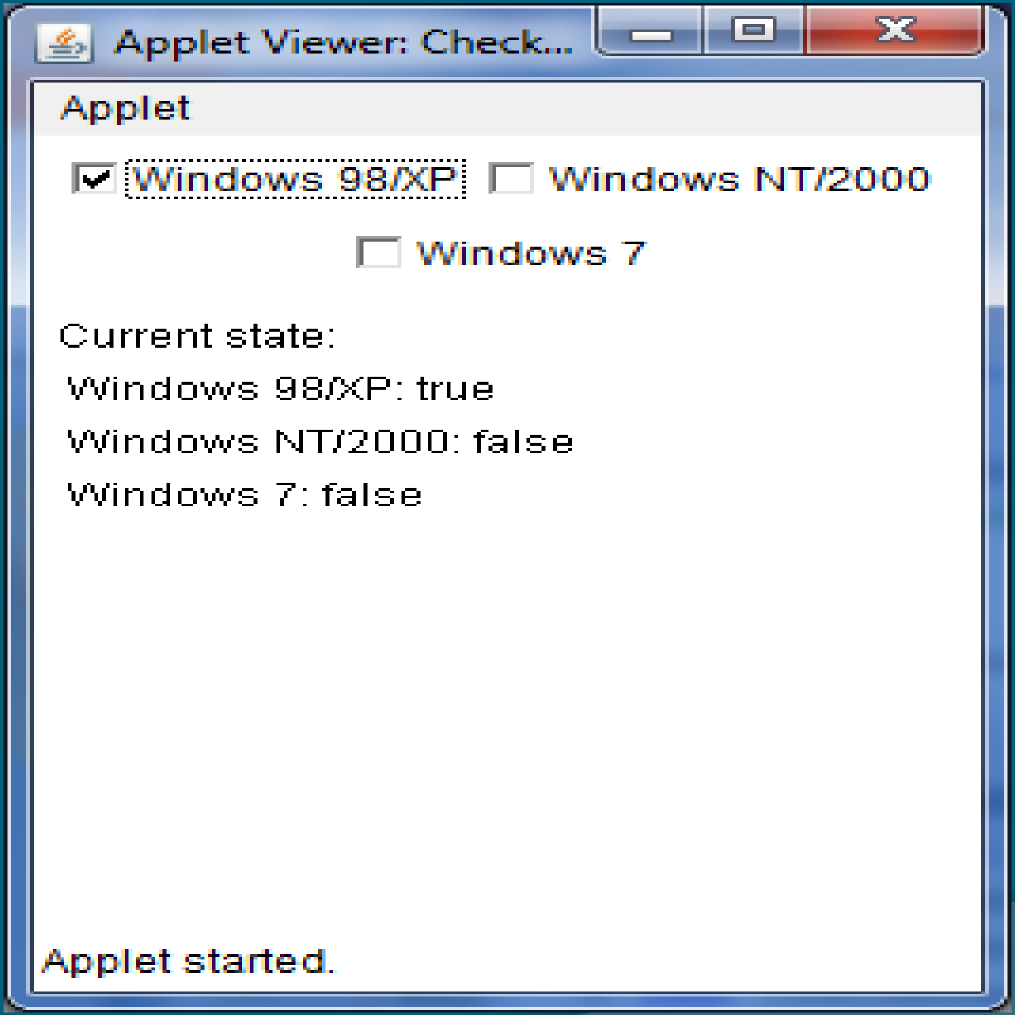
CHECK BOXES:EXAMPLE

```
     add(Win98);
     add(winNT);
     add(Win7);
     Win98.addItemListener(this);
     winNT.addItemListener(this);
     Win7.addItemListener(this);
}   //end of init()

public void itemStateChanged(ItemEvent ie) {
    repaint();
}
```

CHECK BOXES:EXAMPLE

```
public void paint(Graphics g) {
   msg = "Current state: ";
         g.drawString(msg, 6, 80);
   msg = " Windows 98/XP: " + Win98.getState();
            g.drawString(msg, 6, 100);
   msg = " Windows NT/2000: " + winNT.getState();
            g.drawString(msg, 6, 120);
   msg = " Windows 7: " + Win7.getState();
              g.drawString(msg, 6, 140);
   }  //end of paint()
}
```

CHECK BOXES



```
Current state:
 Windows 98/XP: true
 Windows NT/2000: false
 Windows 7: false
```

Applet started.

CHECK BOX GROUP

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- These check boxes are often called radio buttons
- Check box groups are objects of type **CheckboxGroup**

CONSTRUCTOR

CheckboxGroup()

METHODS

Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox which)

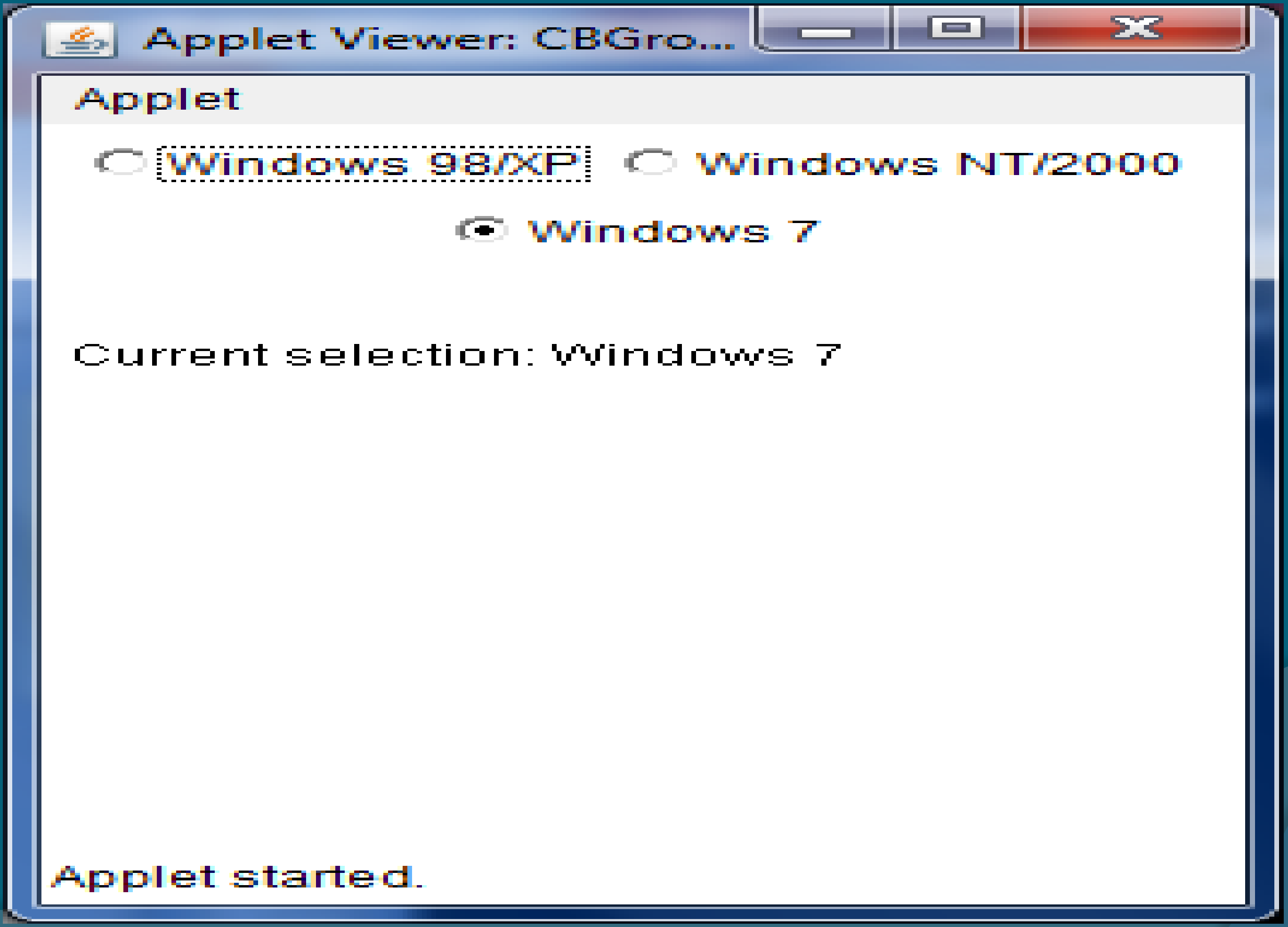
CHECK BOX GROUP: EXAMPLE

```
public class CBGroup extends Applet implements ItemListener
{
    String msg = "";
    Checkbox Win98, winNT, Win7;
    CheckboxGroup cbg;
    public void init() {
        cbg = new CheckboxGroup();
       Win98 = new Checkbox("Windows 98/XP", cbg, true);
       winNT = new Checkbox("Windows NT/2000",cbg,false);
       Win7  = new Checkbox("Windows 7",cbg,true);
       add(Win98);
       add(winNT);
       add(Win7);
```

CHECK BOX GROUP: EXAMPLE

```
        Win98.addItemListener(this);
        winNT.addItemListener(this);
        Win7.addItemListener(this);
    } //end of init()
    public void itemStateChanged(ItemEvent ie) {
         repaint();
    }
    public void paint(Graphics g) {
        msg = "Current selection: ";
        msg += cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg, 6, 100);
    }
}
```

CHECK BOX GROUP: EXAMPLE



Windows 98/XP ◯ Windows NT/2000 ◯ [✓ Windows 7]

```
Current selection: Windows 7
```

Applet started.

LIST

- The List class provides a compact, multiple-choice, scrolling selection list.

- It can also be created to allow multiple selections.

CONSTRUCTORS

  - List( )
  - List(int numRows) ***//(visible rows)***
  - List(int numRows, boolean multipleSelect)

- To add a selection to the list, call add( ).
  - void add(String name)
  - void add(String name, int index)

LIST

METHODS

- **String getSelectedItem( )**
- **int getSelectedIndex( )**
- **int getItemCount( )**
- **void select(int index)**
- **String getItem(int index)**

Each time a **List** item is double-clicked, an **ActionEvent** object is generated.

Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated

LIST:EXAMPLE

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener
{
    List os, browser;
    String msg = "";
    public void init()
     {
        os = new List(4, true);
        browser = new List(4, false);
        os.add("Windows 98/XP");
        os.add("Windows NT/2000");
```

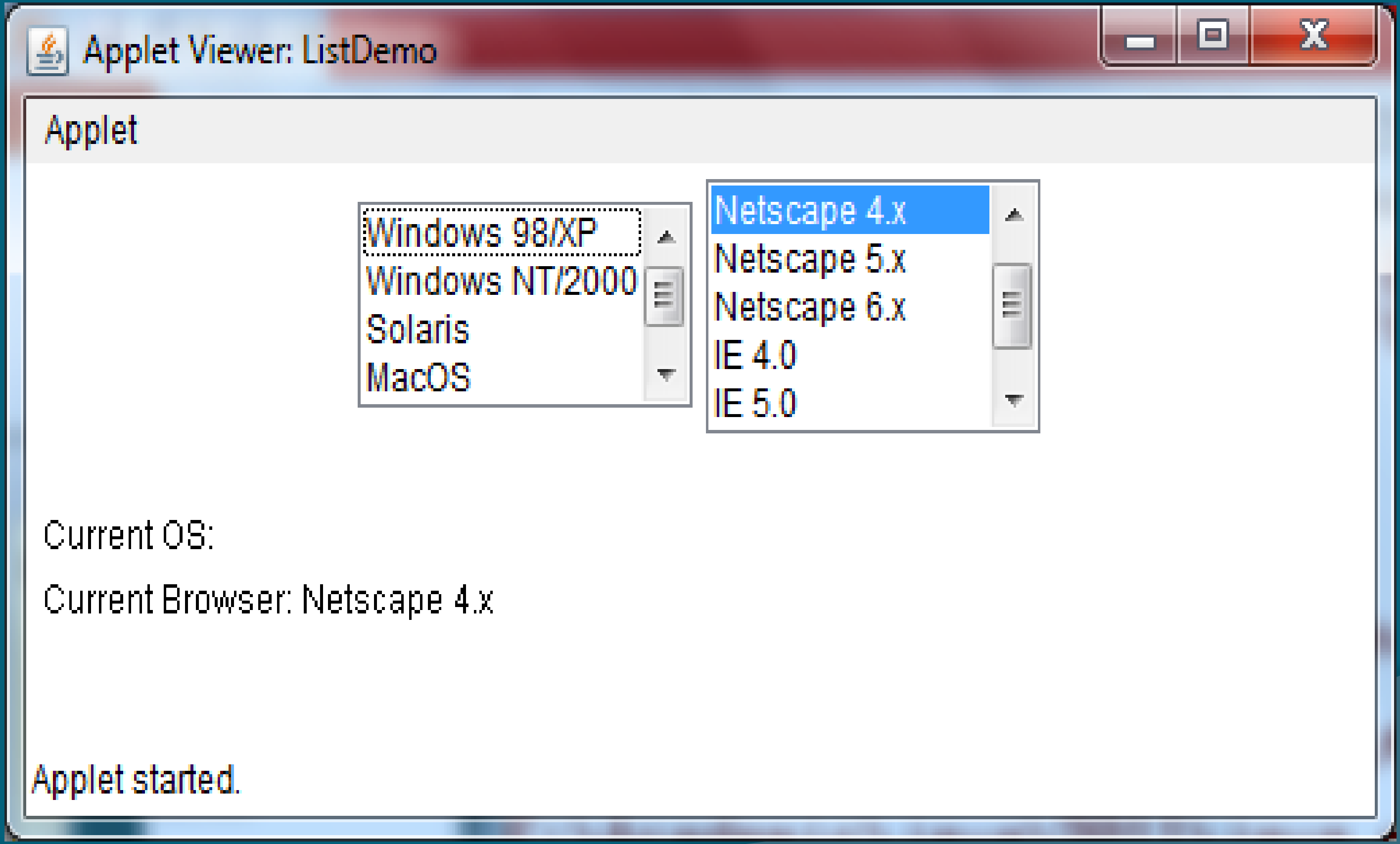
LIST:EXAMPLE

```
        os.add("Solaris");
        os.add("MacOS");
    // add items to browser list
    browser.add("Netscape 3.x");
    browser.add("Netscape 4.x");
    browser.add("Netscape 5.x");
    browser.add("Netscape 6.x");
    browser.add("Internet Explorer 4.0");
    browser.add("Internet Explorer 5.0");
    browser.add("Internet Explorer 6.0");
    browser.select(1);
    // add lists to window
    add(os);
    add(browser);
    // register to receive action events
    os.addActionListener(this);
    browser.addActionListener(this);
}
```

LIST:EXAMPLE

```
public void actionPerformed(ActionEvent ae)
{
  repaint();
}

// Display current selections.
public void paint(Graphics g)
{
      int idx[];
      msg = "Current OS: ";
      idx = os.getSelectedIndexes();
      for(int i=0; i<idx.length; i++)
      msg += os.getItem(idx[i]) + "  ";
      g.drawString(msg, 6, 120);
      msg = "Current Browser: ";
      msg += browser.getSelectedItem();
      g.drawString(msg, 6, 140);
} }
```

LIST: EXAMPLE



Windows 98/XP
Windows NT/2000
Solaris
MacOS

Netscape 4.x
Netscape 5.x
Netscape 6.x
IE 4.0
IE 5.0

Current OS:

Current Browser: Netscape 4.x

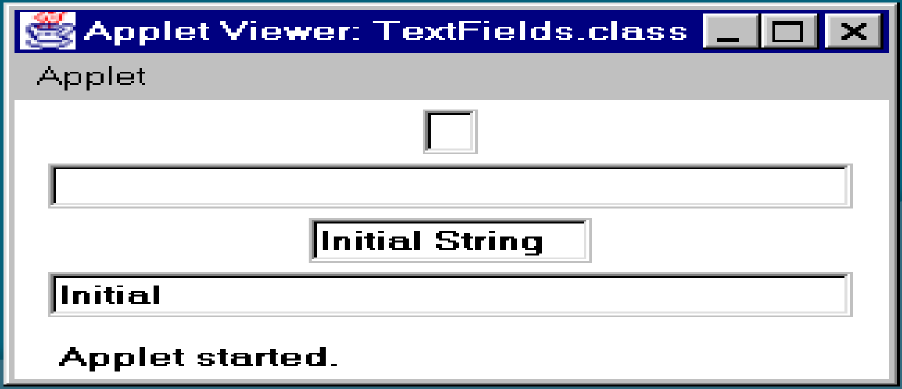
Applet started.

TEXTFIELD

- **TextField** is a subclass of **TextComponent**. It displays a single line of optionally editable text.

CONSTRUCTORS

- TextField( )
- TextField(int numChars) //(width)
- TextField(String str)
- TextField(String str, int numChars)



TEXTFIELD

METHODS

String getText( )
void setText(String *str*)
String getSelectedText( )
void select(int *startIndex*, int *endIndex*)
boolean isEditable( )
void setEditable(boolean *canEdit*)

- When cursor is inside textfield and the user presses ENTER, an **ActionEvent** object is generated.

- When characters are entered by a user in the text fields, **TextEvent** object is generated

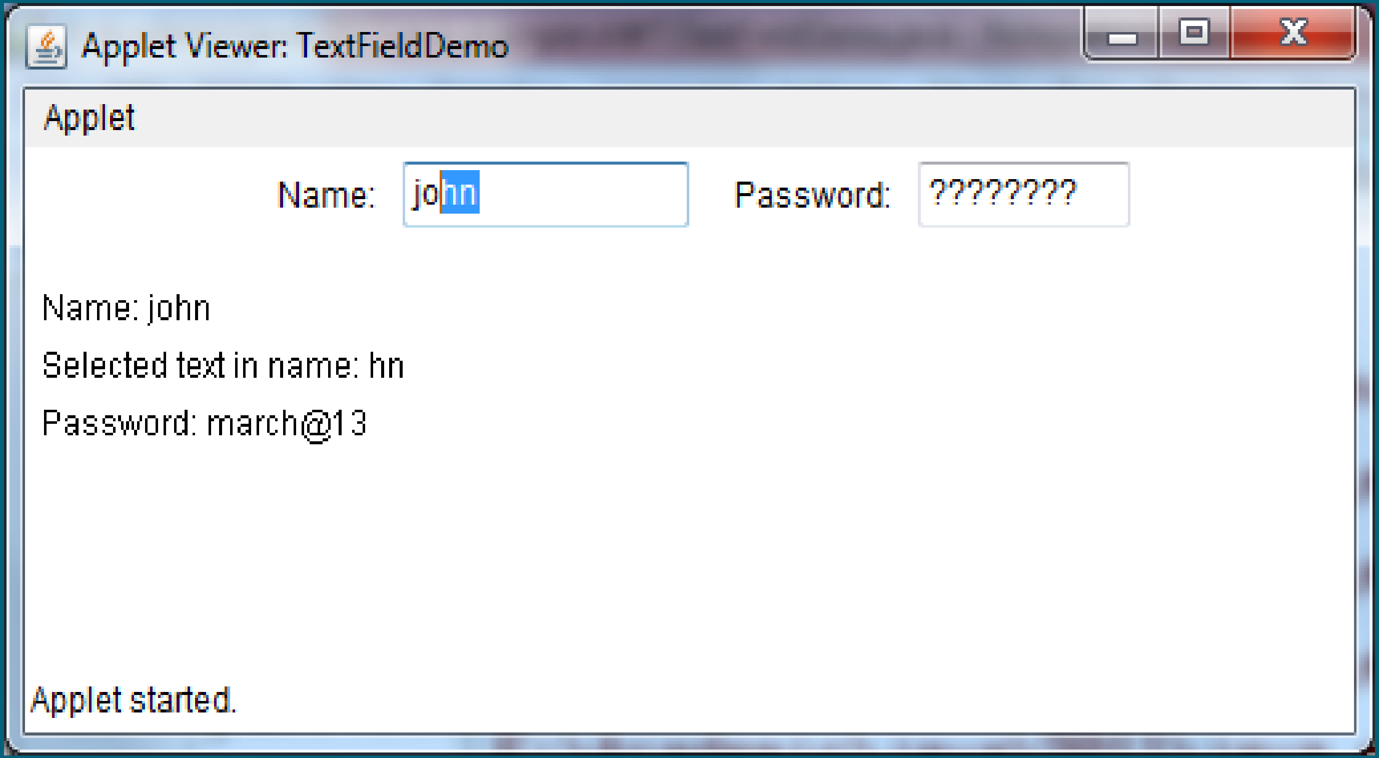
TEXT FIELD:Example

```
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=500 height=180>
</applet>
*/
public class TextFieldDemo extends Applet implements TextListener
{
        TextField name, pass;
        public void init()
        {
            Label namep = new Label("Name: ", Label.RIGHT);
            Label passp = new Label("Password: ", Label.RIGHT);
            name = new TextField(12);
            pass = new TextField(8);
            pass.setEchoChar('?');
            add(namep);
```

TEXT FIELD:Example

```
    add(name);
    add(passp);
    add(pass);
    name.addTextListener(this);
    pass.addTextListener(this);
}
public void textValueChanged(TextEvent ae) {
    repaint();
}
public void paint(Graphics g)
 {
    g.drawString("Name: " + name.getText(), 6, 60);
    g.drawString("Selected text in name: "
    + name.getSelectedText(), 6, 80);
    g.drawString("Password: " + pass.getText(), 6, 100);
} }
```

TEXT FIELD:Example



```
Name: john
Selected text in name: hn
Password: march@13
```

TEXTAREA

- **TextArea** is simple multiline editor

CONSTRUCTORS

- TextArea( )
- TextArea(int numLines, int numChars)
- TextArea(String str)
- TextArea(String str, int numLines, int numChars)
- TextArea(String str, int numLines, int numChars, int sBars)

SBAR VALUES:

- SCROLLBARS_BOTH
- SCROLLBARS_NONE
- SCROLLBARS_HORIZONTAL_ONLY
- SCROLLBARS_VERTICAL_ONLY

TEXTAREA

METHODS

- void append(String str)
- void insert(String str, int index)
- void replaceRange(String str, int startIndex, int endIndex)

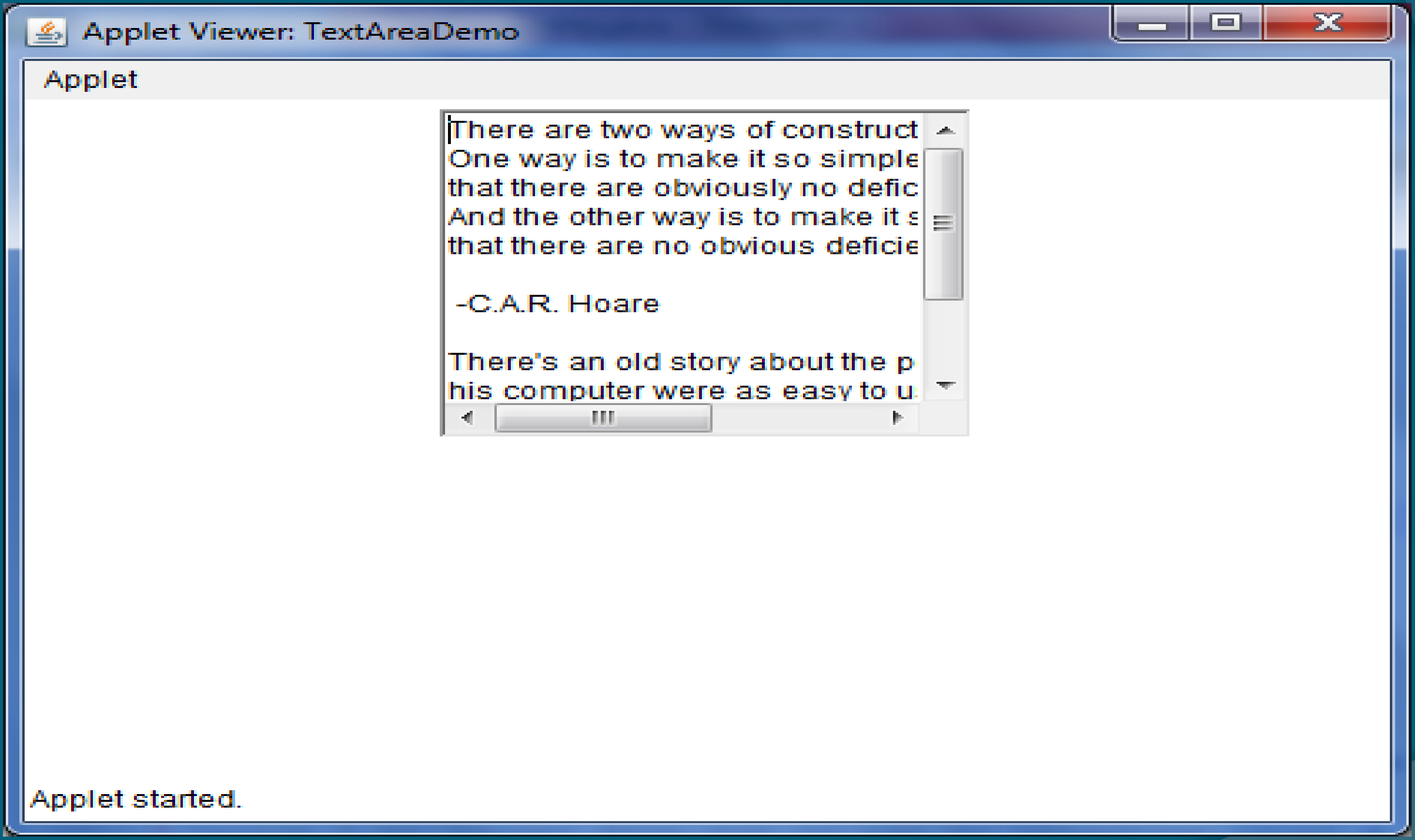
EXAMPLE

```
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=500 height=350>
</applet>
*/
```

TEXTAREA

```
public class TextAreaDemo extends Applet {
public void init()
 {
        String val = "There are two ways of constructing " +
        "a software design.\n" +
        "One way is to make it so simple\n" +
        "that there are obviously no deficiencies.\n" +
        "And the other way is to make it so complicated\n" +
        "that there are no obvious deficiencies.\n\n" +
        " -C.A.R. Hoare\n\n" +
        "There's an old story about the person who wished\n" +
        "his computer were as easy to use as his telephone.\n" +
        "That wish has come true,\n" +
        "since I no longer know how to use my telephone.\n\n" +
        " -Bjarne Stroustrup, AT&T, (inventor of C++)";
        TextArea text = new TextArea(val, 10,
        25,TextArea.SCROLLBARS_BOTH);
        add(text);
}
}
```

TEXTAREA



```
There are two ways of construct
One way is to make it so simple
that there are obviously no defic
And the other way is to make it s
that there are no obvious deficie

 -C.A.R. Hoare

There's an old story about the p
his computer were as easy to u
```

Applet started.

Menu Bars and Menu

- Menu is a common feature in many programs which allows the program to fit many commands in a hierarchical structure without taking much space on the screen

- This concept is implemented using the classes **MenuBar, Menu, and MenuItem.**

- A menu bar contains one or more **Menu** objects**.**

- Each **Menu** object contains a list of **MenuItem** objects.

- Each **MenuItem** object represents something that can be selected by the user.

MENU AND MENU ITEM

CONSTRUCTORS- MENUBAR

- MenuBar( )

CONSTRUCTORS- MENU

- Menu( )
- Menu(String optionName)   // name of menu
- Menu(String optionName, boolean removable)

CONSTRUCTORS- MENU ITEM

- MenuItem( )
- MenuItem(String itemName)

Menu Bars and Menu

METHODS:

- void setEnabled(boolean enabledFlag)
- boolean isEnabled( )
- void setLabel(String newName)
- String getLabel( )

Checkable menu item

Checkable menu item can be created by using a subclass of **MenuItem** called **CheckboxMenuItem.**

CONSTRUCTORS:

- CheckboxMenuItem( )
- CheckboxMenuItem(String itemName)
- CheckboxMenuItem(String itemName, boolean on)

METHODS:

- boolean getState( )
- void setState(boolean checked)

EXAMPLE:

```
import java.awt.*;
class MenuFrame extends Frame
{
   String msg = "";
   MenuFrame(String title) {
         super(title);

   // create menu bar and add it to frame
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);
       Menu edit = new Menu("Edit");
```

EXAMPLE:

```
MenuItem item6, item7, item8, item9;
     edit.add(item6 = new MenuItem("Cut"));
     edit.add(item7 = new MenuItem("Copy"));
     edit.add(item8 = new MenuItem("Paste"));
     item8.setEnabled(false);
     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);
```

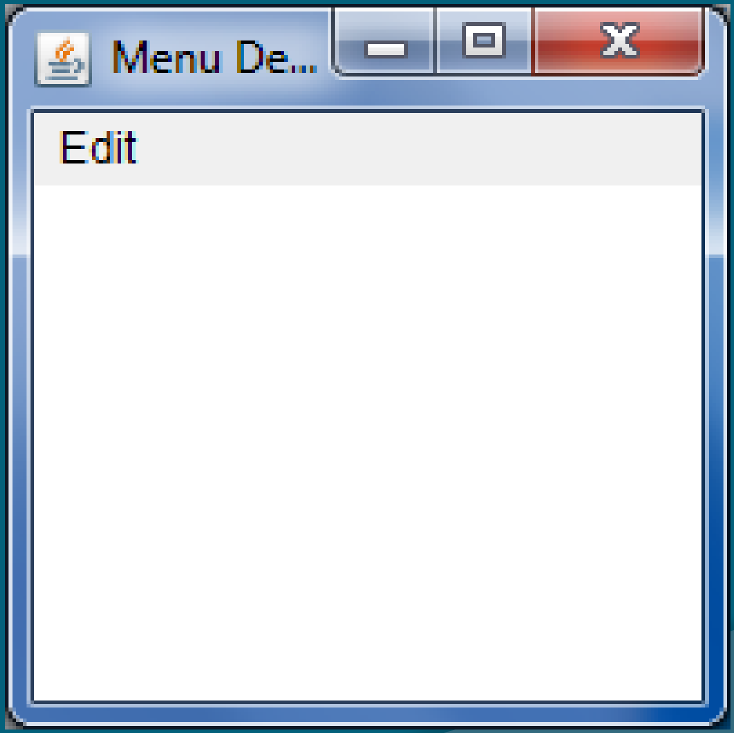
EXAMPLE:

```
// these are checkable menu items
CheckboxMenuItem debug,test;
   debug = new CheckboxMenuItem("Debug");
   edit.add(debug);
   test = new CheckboxMenuItem("Testing");
   edit.add(test);
mbar.add(edit);
}
}
```

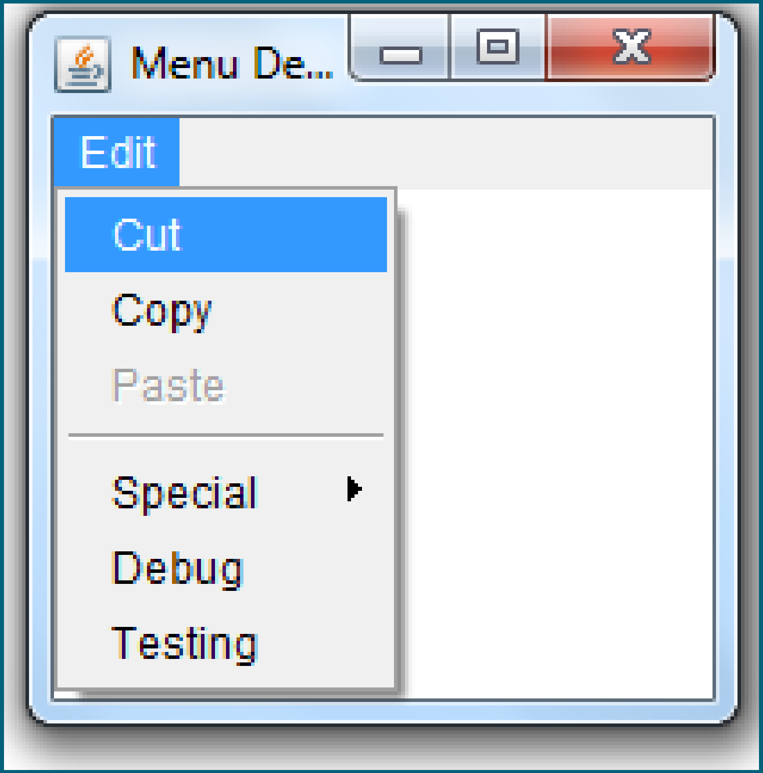
EXAMPLE:

```
public class MenuBarDemo  {
public static void main (String args[])
{
   Frame f;
  f = new MenuFrame("Menu Demo");
  f.setSize(200,200);
  f.setVisible(true);
}
}
```

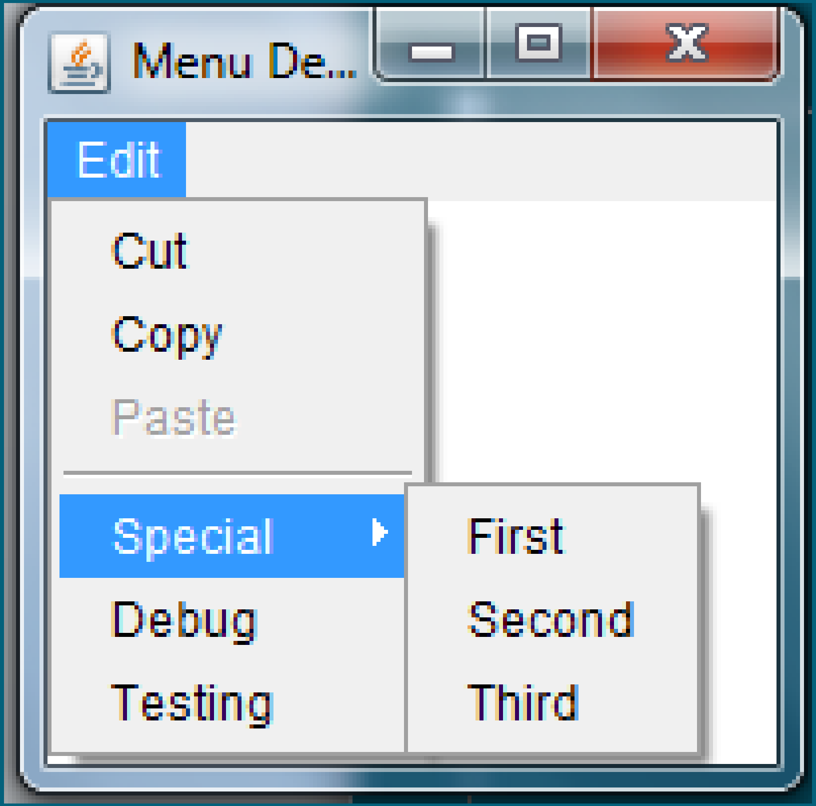
EXAMPLE:



EXAMPLE:



EXAMPLE:


Menu De...
Edit
Cut
Copy
Paste
Special
Debug
Testing
First
Second
Third

EVENT HANDLING WITH MENUS

- Menus only generate events when an item of type **MenuItem or CheckboxMenuItem** is selected.
- Each time a menu item is selected, an **ActionEvent** object is generated**.**
- Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated.
- You must implement the **ActionListener and ItemListener** interfaces in order to handle these menu events.
- The **getItem()** method of ItemEvent returns a reference to the item that generated this event.

ADAPTER CLASSES

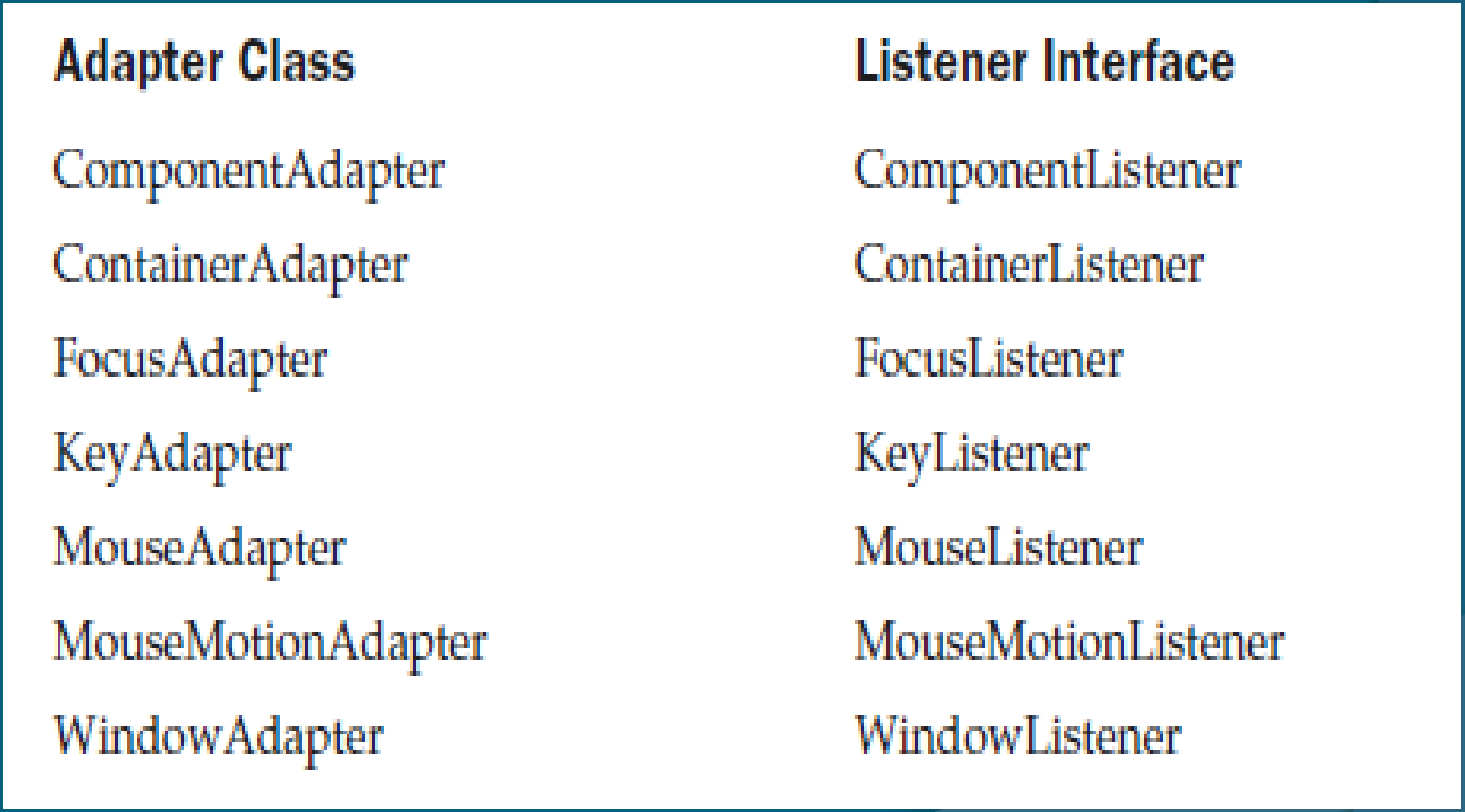
- Java provides a special feature, called an adapter class, that can simplify the creation of event handlers in certain situations.

- 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.

- Define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

Commonly Used Adapter Classes



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

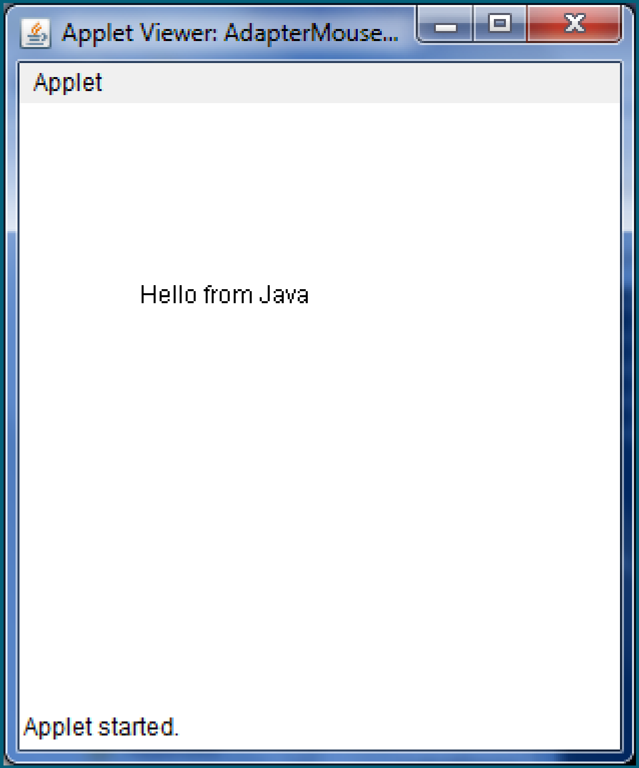
ADAPTER :EXAMPLE

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code="AdapterMouseDemo" width=300
height=300>
</applet>*/
public class AdapterMouseDemo extends Applet {
 String s="Hello from Java";
public void init()
{ addMouseListener(new MyMouseAdapter(this));
}
public void paint(Graphics g)
{  g.drawString(s,60,100); }
}
```

ADAPTER :EXAMPLE

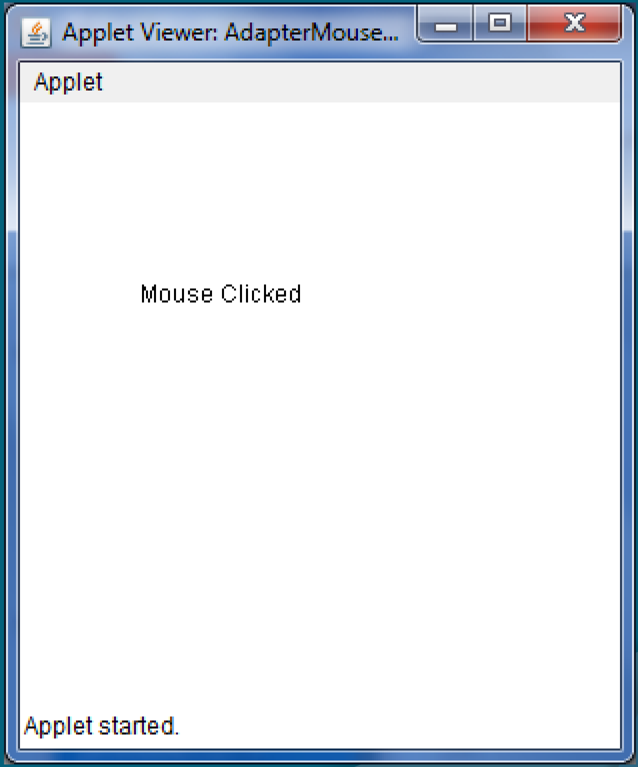
```
class MyMouseAdapter extends MouseAdapter {

AdapterMouseDemo adapterDemo;

public MyMouseAdapter(AdapterMouseDemo ad)
{
    adapterDemo = ad;
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me)
{
   adapterDemo.s="Mouse Clicked";
   adapterDemo.repaint();
}
}
```

ADAPTER :EXAMPLE



```
Hello from Java
```

```
Applet started.
```



```
Mouse Clicked
```

```
Applet started.
```

LAYOUT MANAGER

- AWT comes with several Layout Managers which automatically arranges your controls within a window by using some type of algorithm.

- All of the components that we have shown so far have been positioned by the default layout manager.

- The layout manager is set by the **setLayout()** method. If no call to **setLayout()** is made, then the default layout manager is used.

LAYOUT MANAGER

- Java has several predefined **LayoutManager** classes

- The Layout managers available are:
  - **FlowLayout**
  - **BorderLayout**
  - **GridLayout**
  - **CardLayout**

FLOWLAYOUT

- FlowLayout is the default layout manager.

- FlowLayout implements a simple layout style, which is similar to how words flow in a text editor

CONSTRUCTORS

- FlowLayout( )
- FlowLayout(int how)
- FlowLayout(int how , int horz , int vert)

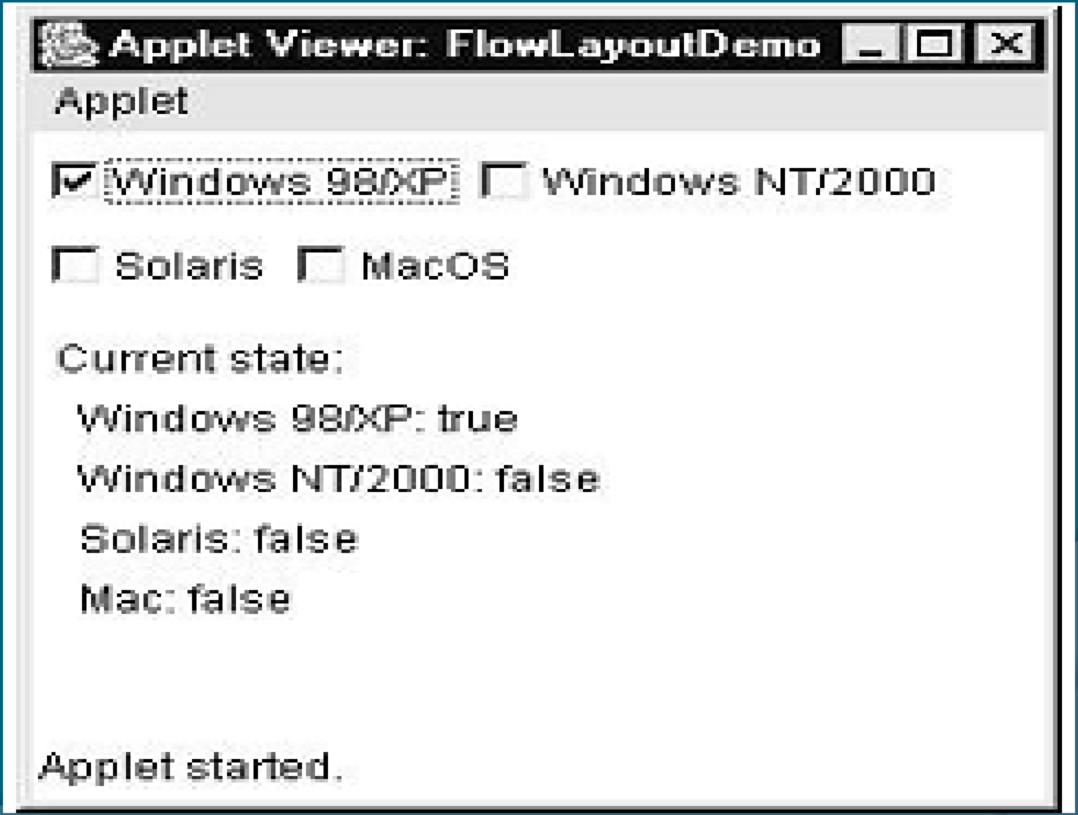
how can take values **FlowLayout.LEFT,FlowLayout.CENTER and FlowLayout.RIGHT**

FLOWLAYOUT

- Components are given their preferred size.
- The order in which the components are added determines their order in the container. The first component added appears to the left, and subsequent components flow in from the right.
- If the container is not wide enough to display all of the components, the components wrap around to a new line.
- You can control whether the components are centered, left-justified, or right-justified.
- You can control the vertical and horizontal gap between components

FLOWLAYOUT

```
public void init() {
// set left-aligned flow layout
setLayout(new FlowLayout(FlowLayout.LEFT));
---------
----------
}
```



```
Current state:
  Windows 98/XP: true
  Windows NT/2000: false
  Solaris: false
  Mac: false
```

Applet started.

BORDERLAYOUT

- The BorderLayout class implements a common layout style for top-level windows.
- It has four narrow, fixed-width components at the edges and one large area in the center.
- The four sides are referred to as north, south, east, and west. The middle area is called the center.
- Only one component can be added to a given region, and the size of the component is determined by the region it appears in.

CONSTRCUTORS

- BorderLayout( )
- BorderLayout(int horz , int vert)

BORDERLAYOUT

- When a component is added, you pass in an int to the add() method that denotes the region of the container in which the component is to be added
- A component added to the north or south will stretch themselves horizontally to fit the entire region
- A component added to the east or west will stretch themselves vertically
- A component added to the center stretch itself vertically and horizontally to fill leftover region

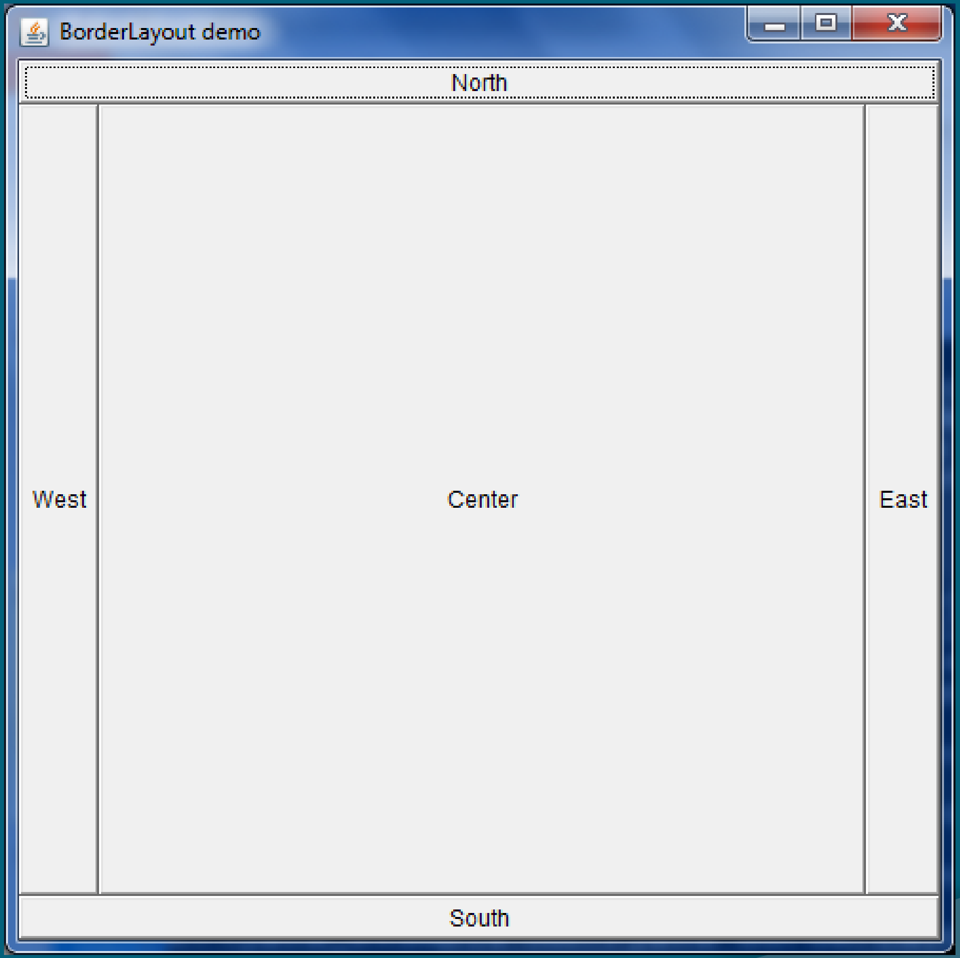
BORDERLAYOUT

```
import java.awt.*;
public class BorderLayoutDemo extends Frame
{
    private Button north, south, east, west, center;
    public BorderLayoutDemo(String title)
    {
        super("BorderLayout demo");
        setLayout(new BorderLayout());
        setSize(500,500);
        north = new Button("North");
        south = new Button("South");
        east = new Button("East");
        west = new Button("West");
        center = new Button("Center");
```

BORDERLAYOUT

```
     this.add(north, BorderLayout.NORTH);
     this.add(south, BorderLayout.SOUTH);
     this.add(east, BorderLayout.EAST);
     this.add(west, BorderLayout.WEST);
     this.add(center, BorderLayout.CENTER);
 }
    public static void main(String [] args)
    {
       Frame f = new BorderLayoutDemo("BorderLayout demo");
       f.setVisible(true);
    }
 }
```

BORDERLAYOUT



GridLayout

- GridLayout lays out components in a two- dimensional grid.

- When you instantiate a GridLayout, you define the number of rows and columns.

CONSTRUCTORS

- GridLayout( )
- GridLayout(int numRows, int numColumns)
- GridLayout(int numRows, int numColumns , int horz , int vert)

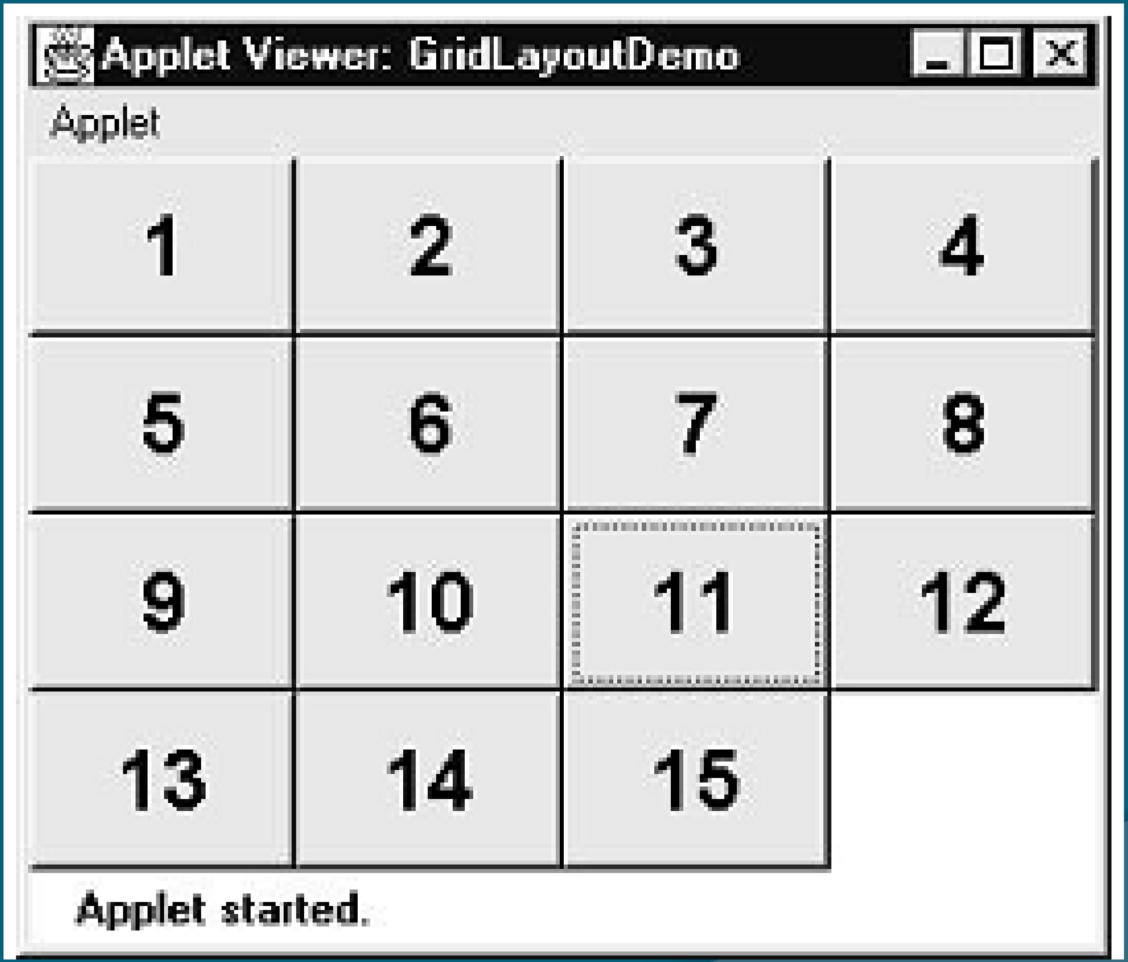
GridLayout

- Only one component can be added to each region of the grid.
- Each region of the grid will have the same size. When the container is resized, each region of the grid will be resized accordingly.
- No components get their preferred height or width. Each component in the container is the same size, which is the current size of the regions of the grid.
- The order in which components are added determines their locations in the grid.

GridLayout:EXAMPLE

```
public class GridLayoutDemo extends Applet {
static final int n = 4;
public void init()
{  setLayout(new GridLayout(n, n));
   for(int i = 0; i < n; i++) {
   for(int j = 0; j < n; j++) {
   int k = i * n + j;
   if(k > 0)
   add(new Button("" + k));
}}
}}
```

GridLayout:EXAMPLE



Applet started.

CardLayout

The CardLayout class is unique among the other layout managers.

Other layout managers try to display all the components added to the container, while card layout manager displays only one component at a time..

This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.

CONSTRUCTORS

```
    CardLayout( )
    CardLayout(int horz , int vert )
```

CardLayout

- Each component added to the container, using CardLayout, is treated as a **card**.
- It arranges GUI components into a "deck" of cards where only the top card is visible, any card in the deck can be placed at the top
- Each card can be either a component or container.
- .

CardLayout

- When cards are added to a panel, they are usually given a name.
  - void add(Component Obj, Object name);
- After you have created a deck, your program activates a card by calling one of the following methods defined by CardLayout:
  - void first(Container deck)
  - void last(Container deck)
  - void next(Container deck)
  - void previous(Container deck)
  - void show(Container deck , String cardName)

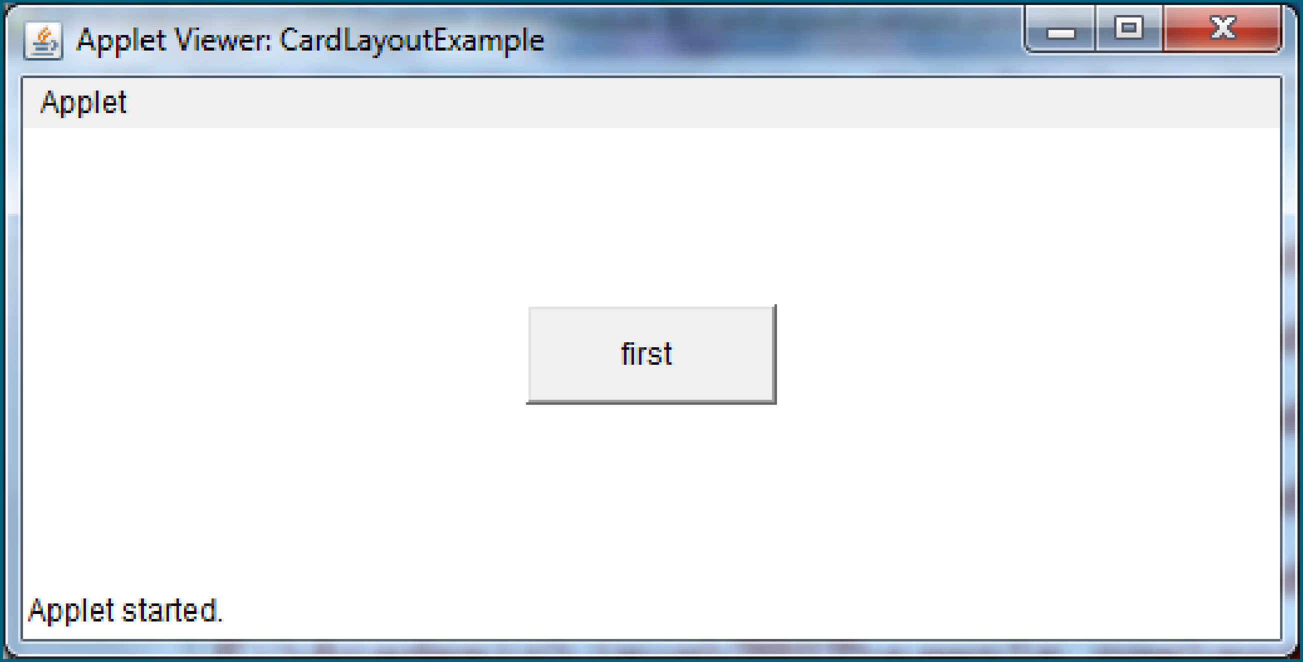
CardLayout

```
public class CardLayoutExample extends Applet implements

ActionListener
{
  CardLayout card;
   Button Btnfirst,BtnSecond,BtnThird;
   public void init()
   {
         card = new CardLayout(200,70);
         setLayout(card);
          Btnfirst = new Button("first ");
          BtnSecond = new Button ("Second");
          BtnThird = new Button("Third");
         add(Btnfirst,"card1");
```

CardLayout

```
      add(BtnSecond,"card2");
        add(BtnThird,"card3");
        Btnfirst.addActionListener(this);
       BtnSecond.addActionListener (this);
       BtnThird.addActionListener(this);
   }
   public void actionPerformed(ActionEvent e)
   {
       card.next(this);
   }
}
```

CardLayout



Applet started.

CardLayout

- Each card can be a Panel which can use any layout manager.
- This layout is used whenever there is a need for a number of panels, each with a different layout to be displayed one by one.
- The main panel must have CardLayout selected as its layout manager.
- First a panel is created that contains the deck and also a panel for each card in the deck.
- Next, you add to the appropriate panel the components that form each card.
- You then add these panels to the panel for which **CardLayout** is the layout manager.

CardLayout

```
public class Card1 extends Applet implements ActionListener
{
  Panel cardPanel;// the container that will hold the various "cards"
  Panel firstP, secondP, thirdP;
  Panel buttonP;                    // panel to hold three buttons
  Button first, second, third;    // the three buttons
  CardLayout ourLayout;             // the card layout object
```

CardLayout

```
public void init()
 {
   //create cardPanel which is the panel that will contain the three "cards"
   cardPanel = new Panel();
   //create the CardLayout object
   ourLayout = new CardLayout();
   //set card Panel's layout to be our Card Layout
   cardPanel.setLayout (ourLayout);
   //create three dummy panels (the "cards") to show
   firstP = new Panel();
   firstP.setBackground(Color.blue);
   secondP = new Panel();
   secondP.setBackground(Color.yellow);
   thirdP = new Panel();
   thirdP.setBackground(Color.green);
```

CardLayout

```
//create three buttons and add ActionListener
    first = new Button("First");
    first.addActionListener(this);
    second = new Button("Second");
    second.addActionListener(this);
    third = new Button("Third");
    third.addActionListener(this);
    //create Panel for the buttons and add the buttons to it
    buttonP = new Panel();   // default Layout manager is FlowLayout
    buttonP.add(first);
    buttonP.add(second);
    buttonP.add(third);
```

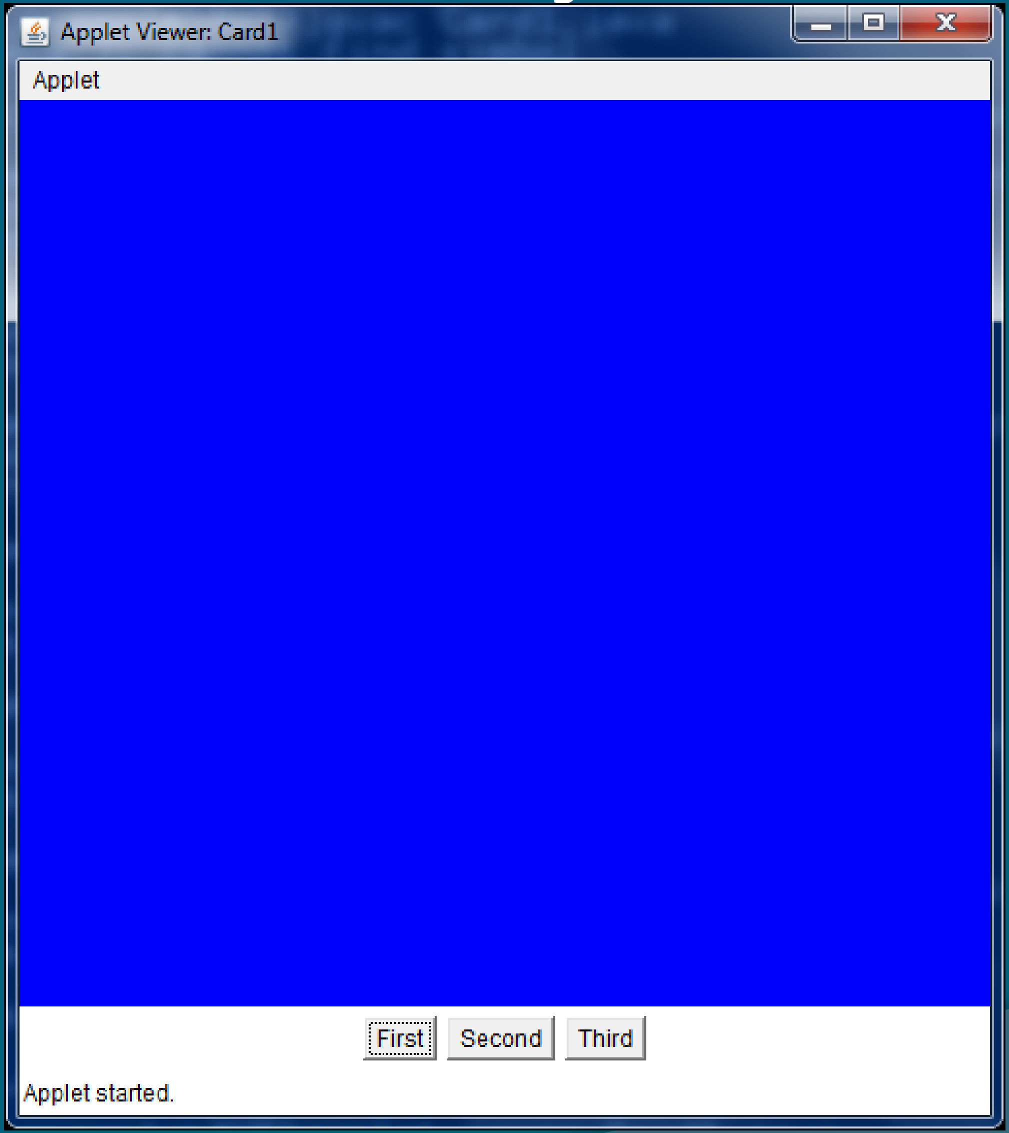
CardLayout

```
//setLayout for applet to be BorderLayout
   this.setLayout(new BorderLayout());
   //button Panel goes South, card panels go Center
   this.add(buttonP, BorderLayout.SOUTH);
   this.add(cardPanel, BorderLayout.CENTER);

   // add the three card panels to the card panel container
  // first one added is the visible one when applet appears
   cardPanel.add(firstP, "First");      //blue
   cardPanel.add(secondP, "Second");   //yellow
   cardPanel.add(thirdP, "Third");      //green
 }
```

CardLayout

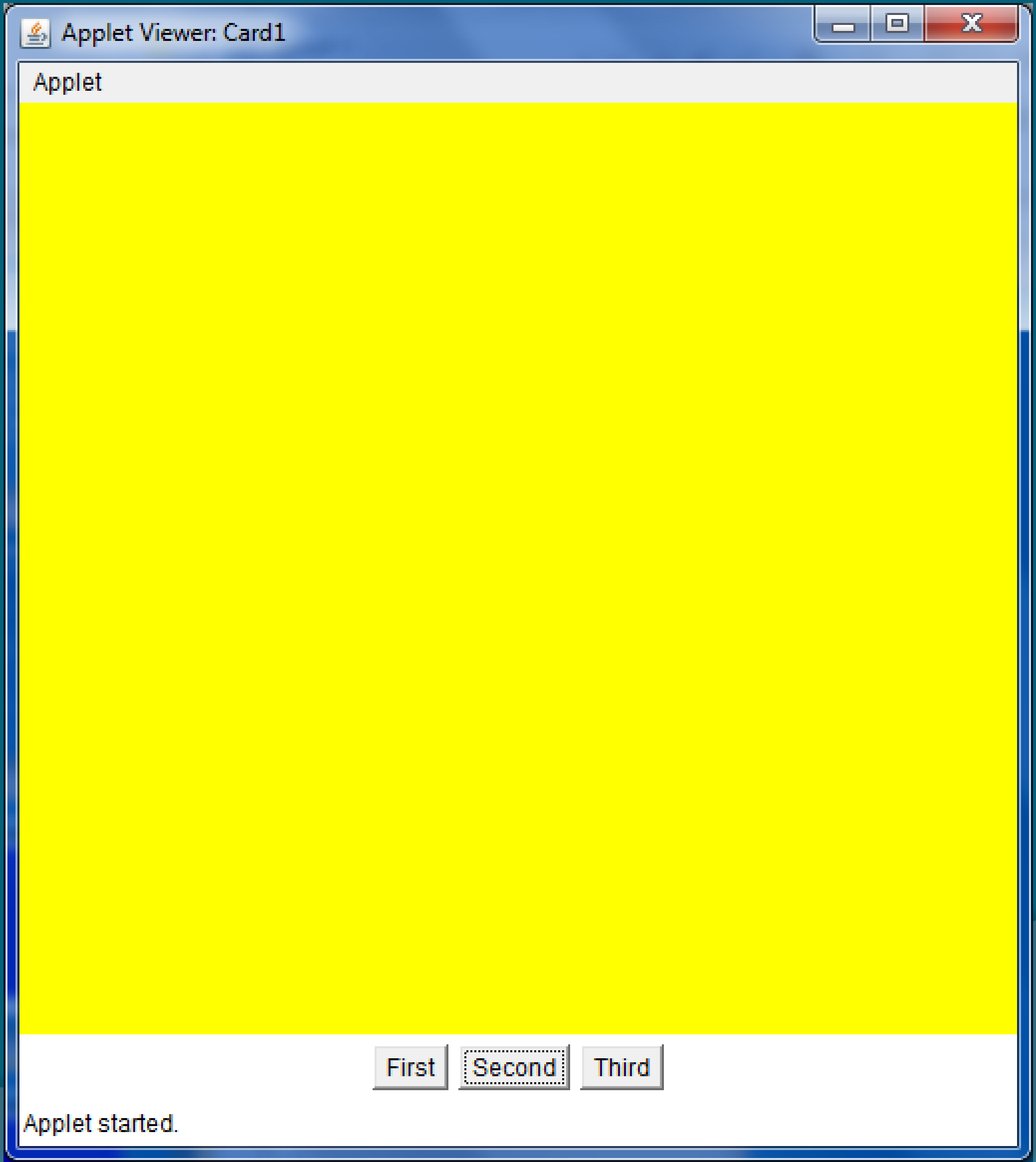
```
public void actionPerformed(ActionEvent e)
 {
   if (e.getSource() == first)
     ourLayout.show(cardPanel, "First");


   if (e.getSource() == second)
     ourLayout.show(cardPanel, "Second");


   if (e.getSource() == third)
     ourLayout.show(cardPanel, "Third");
 }
}
```

CardLayout



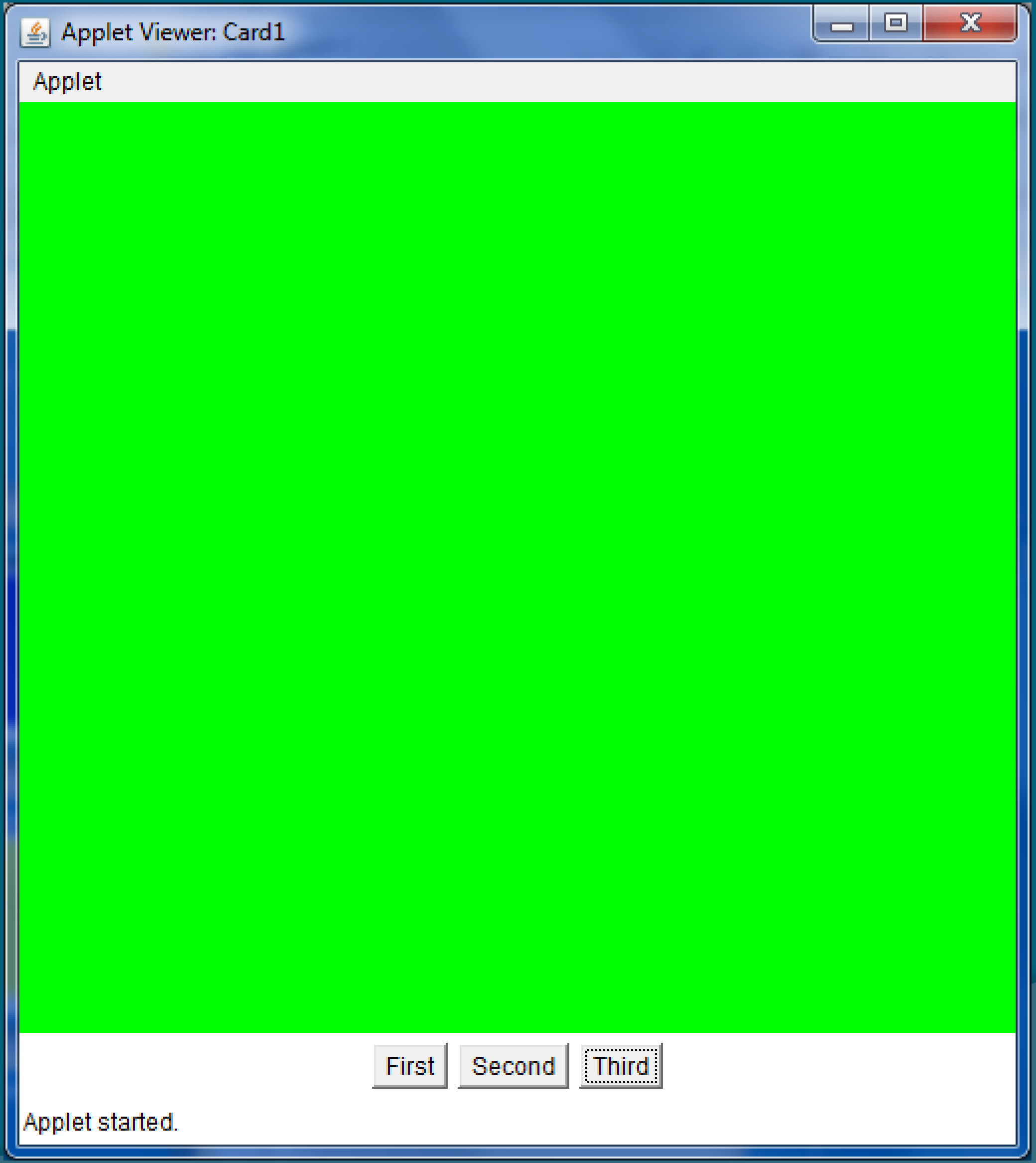
Applet started.

CardLayout



Applet started.

CardLayout



Applet started.