Unit - 5

Collections Overview

Introduction

- Collections were not part of the original Java release, but were added by J2SE 1.2. Prior to the Collections Framework, Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects.

The Collections Framework was designed to meet several goals.

- First, the framework had to be high-performance. The implementations for the fundamental collections are highly efficient.
- Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- Third, extending and/or adapting a collection had to be easy.

The Collection Interfaces(1)

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

The Collection Interfaces(2)

The Collection Interface

The **Collection** interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection.

Collection is a generic interface that has this declaration:

interface Collection<E>

Here, **E** specifies the type of objects that the collection will hold. **Collection** extends the **Iterable** interface.

The List Interface

The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements. **List** is a generic interface that has this declaration:

interface List<E>

Here, **E** specifies the type of objects that the list will hold.

The Collection Interfaces(3)

The SortedSet Interface

- The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order.

SortedSet is a generic interface that has this declaration:

interface SortedSet<E>

Here, **E** specifies the type of objects that the set will hold.

The NavigableSet Interface

- The **NavigableSet** interface extends **SortedSet** and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.

NavigableSet is a generic interface that has this declaration:

interface NavigableSet<E>

Here, **E** specifies the type of objects that the set will hold.

The Collection Interfaces(4)

The Queue Interface

- The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. **Queue** is a generic interface that has this declaration:

```
interface Queue<E>
```

Here, **E** specifies the type of objects that the queue will hold.

The Deque Interface

- The **Deque** interface extends **Queue** and declares the behavior of a double-ended queue.
- Double-ended queues can function as standard, first-in, first-out queues or as last-in, firstout stacks. **Deque** is a generic interface that has this declaration:

```
interface Deque<E>
```

- Here, **E** specifies the type of objects that the deque will hold. In addition to the methods that it inherits from **Queue**

The Collection Classes(1)

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface.
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

ArrayList Class

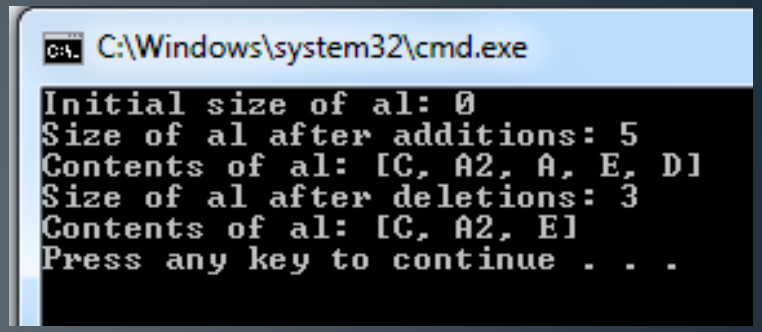
- The **ArrayList** class extends **AbstractList** and implements the **List** interface.

ArrayList is a generic class that has this declaration:

class ArrayList<E>

Here, **E** specifies the type of objects that the list will hold. **ArrayList** supports dynamic arrays that can grow as needed.

```
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>(); // Create an array list.
System.out.println("Initial size of al: " +
al.size());.
al.add("C"); // Add elements to the array list
al.add("A");
al.add("E");
al.add("D");
 al.add(1, "A2");
System.out.println("Size of al after additions: " + al.size());
System.out.println("Contents of al: " + al); // Display the array list.
al.remove("F"); // Remove elements from the array list.
al.remove(2);
System.out.println("Size of al after deletions: " +
al.size());
System.out.println("Contents of al: " + al);
```



The LinkedList Class

The **LinkedList** class extends **AbstractSequentialList** and implements the **List, Deque,** and **Queue** interfaces. It provides a linked-list data structure.

- **LinkedList** is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, **E** specifies the type of objects that the list will hold.

```
// Demonstrate LinkedList.
import java.util.*;
class LinkedListDemo {
public static void main(String args[]) {
// Create a linked list.
LinkedList<String> ll = new
LinkedList<String>();
// Add elements to the linked list.
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll:
" + ll);
// Remove elements from the linked list.
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after
deletion: " + ll);
// Remove first and last elements.
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first
and last: " + ll);
// Get and set a value.
String val = ll.get(2);
ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

C:\Windows\system32\cmd.exe

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
Press any key to continue . . .
```

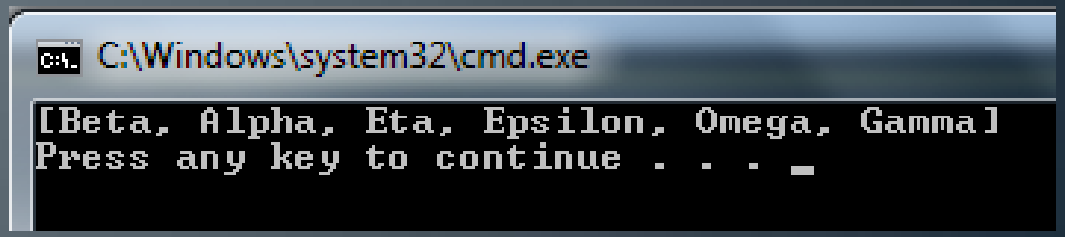
The HashSet Class

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. **HashSet** is a generic class that has this declaration:

```
class HashSet<E>
```

Here, **E** specifies the type of objects that the set will hold.

```
import java.util.*;
class HashSetDemo {
public static void main(String args[]) {
HashSet<String> hs = new HashSet<String>(); // Create a hash set.
hs.add("Beta"); // Add elements to the hash set.
hs.add("Alpha");
hs.add("Eta");
hs.add("Gamma");
hs.add("Epsilon");
hs.add("Omega");
System.out.println(hs);
}
```



The LinkedHashSet Class

The **LinkedHashSet** class extends **HashSet** and adds no members of its own. It is a generic class that has this declaration:

```
class LinkedHashSet<E>
```

Here, **E** specifies the type of objects that the set will hold.

Demo

```
import java.util.LinkedHashSet;
public class Demo  {
   public static void main(String[] args)  {
LinkedHashSet<String> linkedset =
                    new LinkedHashSet<String>();
// Adding element to LinkedHashSet
       linkedset.add("A");
       linkedset.add("B");
       linkedset.add("C");
       linkedset.add("D");
// This will not add new element as A
already exists
       linkedset.add("A");
       linkedset.add("E");
System.out.println("Size of LinkedHashSet
= " +    linkedset.size());
System.out.println("Original
LinkedHashSet:" + linkedset);
System.out.println("Removing D from
LinkedHashSet: " +
                          linkedset.remove("D"));
System.out.println("Trying to Remove Z
which is not "+
"present: " + linkedset.remove("Z"));
       System.out.println("Checking if A is
present=" +
 linkedset.contains("A"));
System.out.println("Updated
LinkedHashSet: " + linkedset);
   }  }
```

C:\Windows\system32\cmd.exe

```
Size of LinkedHashSet = 5
Original LinkedHashSet:[A, B, C, D, E]
Removing D from LinkedHashSet: true
Trying to Remove Z which is not present: false
Checking if A is present=true
Updated LinkedHashSet: [A, B, C, E]
Press any key to continue . . .
```

The TreeSet Class

- **TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface.
- It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

class TreeSet<E>

Here, **E** specifies the type of objects that the set will hold

TreeSet Class Demo

```
import java.util.SortedSet;
import java.util.TreeSet;
public class CreateTreeSetExample {
    public static void main(String[] args) {
        // Creating a
TreeSetSortedSet<String> fruits = new
TreeSet<>();
// Adding new elements to a TreeSet
        fruits.add("Banana");
        fruits.add("Apple");
        fruits.add("Pineapple");
        fruits.add("Orange");
System.out.println("Fruits Set : " + fruits);
// Duplicate elements are ignored
        fruits.add("Apple");
System.out.println("After adding duplicate
element \"Apple\" : " + fruits);
// This will be allowed because it's in
lowercase.

fruits.add("banana");System.out.println("Afte
r adding \"banana\" : " + fruits);
    }
}
```

C:\Windows\system32\cmd.exe

```
Fruits Set : [Apple, Banana, Orange, Pineapple]
After adding duplicate element "Apple" : [Apple, Banana, Orange, Pineapple]
After adding "banana" : [Apple, Banana, Orange, Pineapple, banana]
Press any key to continue . . .
```

Vector

- **Vector** implements a dynamic array.
- It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that duplicate the functionality of methods defined by the Collections Framework.
- With the advent of collections, **Vector** was reengineered to extend **AbstractList** and to implement the **List** interface.
- **Vector** is fully compatible with collections, and a **Vector** can have its contents iterated by the enhanced **for** loop.

Vector is declared like this:

- class Vector<E>

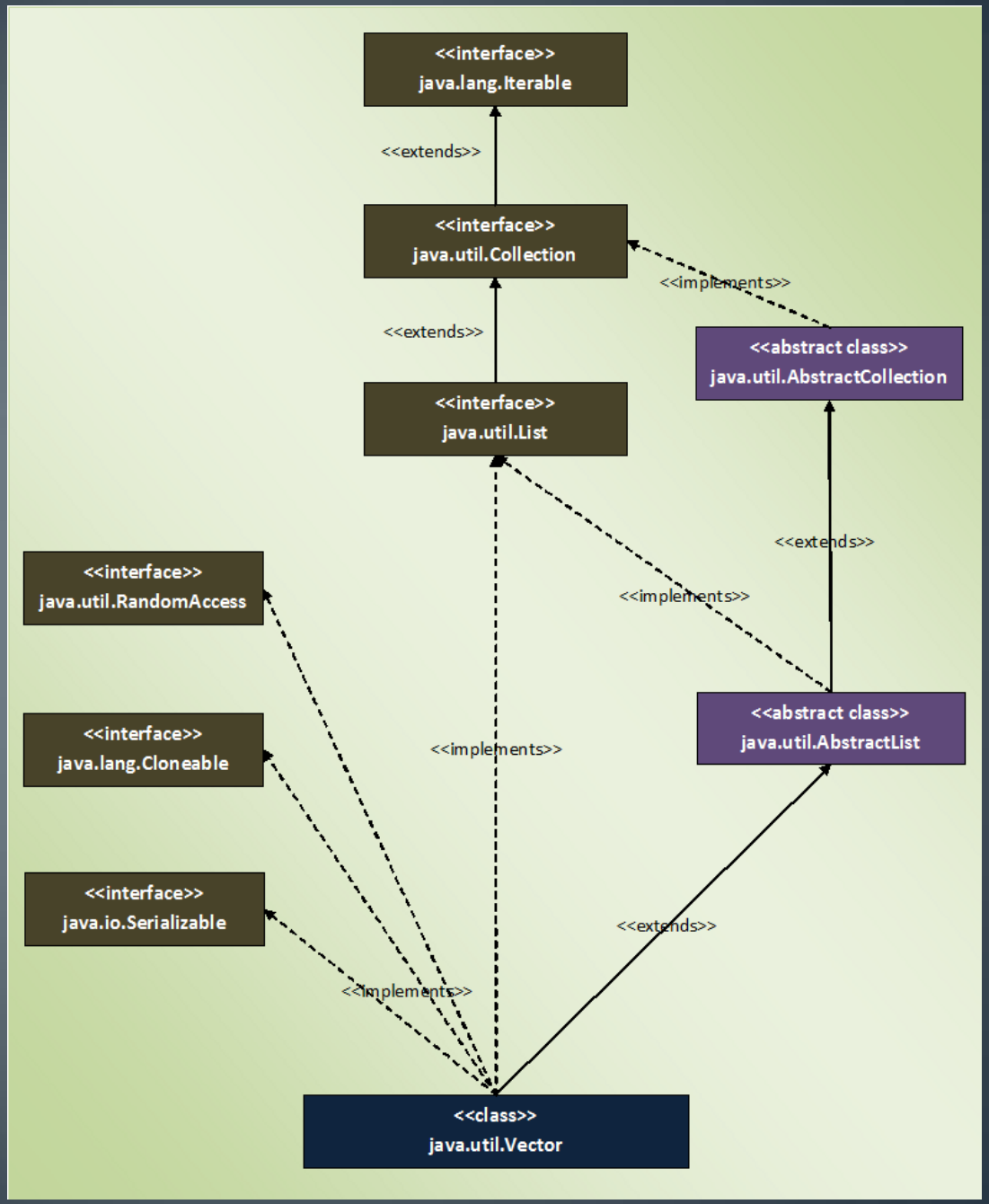
Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:

Vector( )

Vector(int *size*)

Vector(int *size*, int *incr*)


<<interface>>
java.lang.Iterable
<<extends>>
<<interface>>
java.util.Collection
<<implements>>
<<extends>>
<<abstract class>>
java.util.AbstractCollection
<<interface>>
java.util.List
<<extends>>
<<interface>>
java.util.RandomAccess
<<implements>>
<<interface>>
java.lang.Cloneable
<<implements>>
<<abstract class>>
java.util.AbstractList
<<interface>>
java.io.Serializable
<<extends>>
<<implements>>
<<class>>
java.util.Vector

Vector Demo

```
import java.util.*;
public class VectorMethodsDemo      {
  public static void main(String args[]){
  Vector<Integer> vectorObject = new Vector<Integer>(4);
  vectorObject.add(3);
  vectorObject.add(5);
  vectorObject.add(4);
  vectorObject.add(1);
  vectorObject.add(2);
  vectorObject.add(8);
  vectorObject.add(8);
  System.out.println("Values in
          }
```

Vector Demo

```
import java.util.*;
public class VectorMethodsDemo1        {
public static void main(String args[]){
Vector<Integer> vectorObject = new
Vector<Integer>(4);
  vectorObject.add(0,3);
  vectorObject.add(1,5);
  vectorObject.add(2,4);
  vectorObject.add(3,1);
  for(Integer integer: vectorObject)        {
 System.out.println("Index : "
+vectorObject.indexOf(integer)+ " Value:
"+integer);                }
   System.out.println("current capacity of
          Vector object is "
          +vectorObject.capacity());
//Adding element at index 2
  vectorObject.add(2, 10);
  System.out.println("\nAfter adding value at
index 2:\n");
  for(Integer integer: vectorObject)
  {
 System.out.println("Index : "
+vectorObject.indexOf(integer)+ " Value: "
+integer);
   }
   System.out.println("current capacity of
Vector object is "  +vectorObject.capacity());
 }
}
```

C:\Windows\system32\cmd.exe

```
Index : 0 Value:  3
Index : 1 Value:  5
Index : 2 Value:  4
Index : 3 Value:  1
current capacity of Vector object is 4

After adding value at index 2:

Index : 0 Value: 3
Index : 1 Value: 5
Index : 2 Value: 10
Index : 3 Value: 4
Index : 4 Value: 1
current capacity of Vector object is 8
Press any key to continue . . .
```

Stack

- **Stack** is a subclass of **Vector** that implements a standard last-in, first-out stack.
- **Stack** only defines the default constructor, which creates an empty stack. With the release of JDK 5, **Stack** was retrofitted for generics and is declared as shown here:

  class Stack<E>

Here, **E** specifies the type of element stored in the stack.

Method	Description
boolean empty( )	Returns **true** if the stack is empty, and returns **false** if the stack contains elements.
E peek( )	Returns the element on the top of the stack, but does not remove it.
E pop( )	Returns the element on the top of the stack, removing it in the process.
E push(E *element*)	Pushes *element* onto the stack. *element* is also returned.
int search(Object *element*)	Searches for *element* in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned.

```
import java.io.*;
import java.util.*;
  class Test {
 // Pushing element on the top of the stack
static void stack_push(Stack<Integer> stack)
{
        for(int i = 0; i < 5; i++)        {
            stack.push(i);        }    }
// Popping element from the top of the stack
    static void stack_pop(Stack<Integer>
stack)   {
        System.out.println("Pop :");
          for(int i = 0; i < 5; i++)        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);        }    }
// Displaying element on the top of the stack
    static void stack_peek(Stack<Integer>
stack)     {
            Integer element = (Integer)
              stack.peek();
        System.out.println("Element on stack top
              : " + element);
// Searching element in the stack
    static void stack_search(Stack<Integer>
stack, int element)      {
        Integer pos = (Integer)
stack.search(element);
          if(pos == -1)
  System.out.println("Element not found");
  else
  System.out.println("Element is found at
position " + pos);      }
  public static void main (String[] args)      {
        Stack<Integer> stack = new
Stack<Integer>();
        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);      }    }
```

C:\Windows\system32\cmd.exe

```
Pop :
4
3
2
1
0
Element on stack top : 4
Element is found at position 3
Element not found
Press any key to continue . . .
```

Networking

Networking Basics

- Java's networking support is the concept of a socket.
- A socket identifies an endpoint in a network.
- The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term Berkeley socket is also used.
- Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information.
- This is accomplished through the use of a port, which is a numbered socket on a particular machine.
- A server process is said to "listen" to a port until a client connects to it.
- A server is allowed to accept multiple clients connected to the same port number, although each session is unique.
- To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

- Socket communication takes place via a protocol.
- Internet Protocol (IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.
- Transmission Control Protocol (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data.
- A third protocol, User Datagram Protocol (UDP), sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets

The Networking Classes and Interfaces

Authenticator	InetAddress	SocketAddress
CacheRequest	InetSocketAddress	SocketImpl
CacheResponse	InterfaceAddress	SocketPermission
ContentHandler	JarURLConnection	StandardSocketOption
CookieHandler	MulticastSocket	URI
CookieManager	NetPermission	URL
DatagramPacket	NetworkInterface	URLClassLoader

DatagramSocket	PasswordAuthentication	URLConnection
DatagramSocketImpl	Proxy	URLDecoder
HttpCookie	ProxySelector	URLEncoder
HttpURLConnection	ResponseCache	URLPermission (Added by JDK 8.)
IDN	SecureCacheResponse	URLStreamHandler
Inet4Address	ServerSocket	
Inet6Address	Socket	

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption	

Demonstrate InetAddress

```
import java.net.*;
class InetAddressTest
{
public static void main(String args[]) throws UnknownHostException {
InetAddress Address = InetAddress.getLocalHost();
System.out.println(Address);
Address = InetAddress.getByName("www.bmsce.in");
System.out.println(Address);
InetAddress SW[ ] = InetAddress.getAllByName("www.bcci.tv");
for (int i=0; i<SW.length; i++)
System.out.println(SW[i]);
}
}
```

C:\Windows\system32\cmd.exe

```
Admin-PC/10.19.0.46
www.bmsce.in/139.59.4.126
www.bcci.tv/52.77.42.134
www.bcci.tv/52.77.128.120
Press any key to continue . . .
```

```
import java.net.*;
class WriteServer {
public static int serverPort = 998;
public static int clientPort = 999;
public static int buffer_size = 1024;
public static DatagramSocket ds;
public static byte buffer[] = new
byte[buffer_size];
public static void TheServer() throws Exception {
int pos=0;
while (true) {
int c = System.in.read();
switch (c) {
case -1:
System.out.println("Server Quits.");
ds.close();
return;
case '\r':
break;
case '\n':
ds.send(new DatagramPacket(buffer,pos,
InetAddress.getLocalHost(),clientPort));
pos=0;
break;
default:
buffer[pos++] = (byte) c; }  }   }
public static void TheClient() throws
Exception {
while(true) {
DatagramPacket p = new
DatagramPacket(buffer, buffer.length);
ds.receive(p);
System.out.println(new String(p.getData(),
0, p.getLength()));   }            }
public static void main(String args[])
throws Exception {
if(args.length == 1) {
ds = new DatagramSocket(serverPort);
TheServer();
} else {
ds = new DatagramSocket(clientPort);
TheClient();            }            }            }
```

Client

Server

C:\Windows\system32\cmd.exe

C:\Windows\system32\cmd.exe - java WriteServer 1

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin>cd documents

C:\Users\Admin\Documents>java WriteServer 1
```

C:\Windows\system32\cmd.exe

```
Hai from WriteSever
```

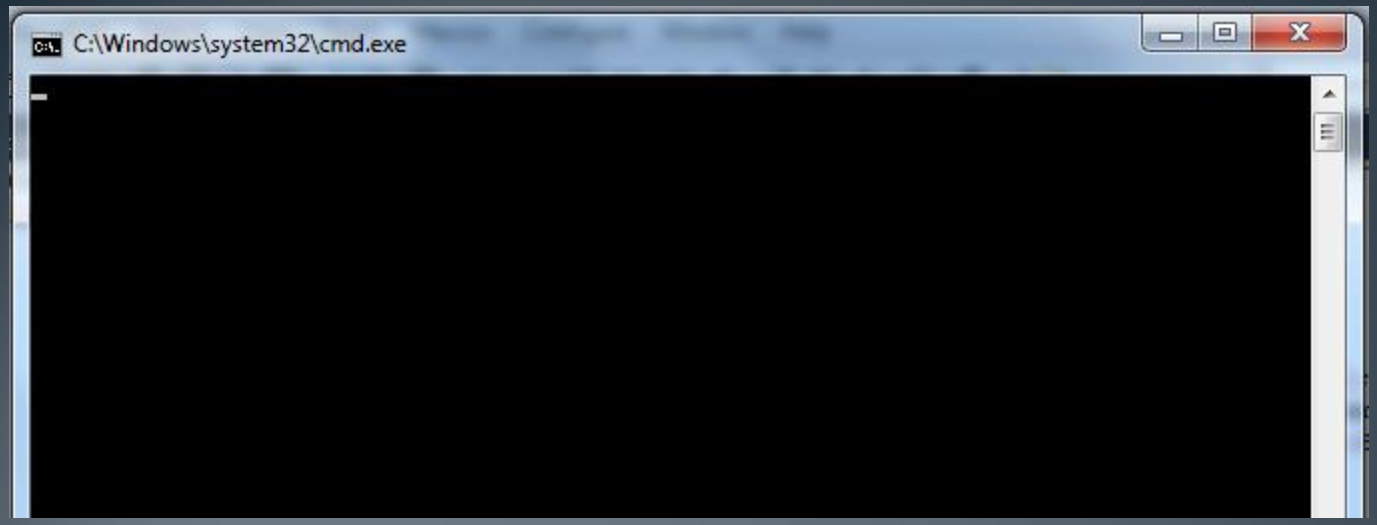
C:\Windows\system32\cmd.exe - java WriteServer 1

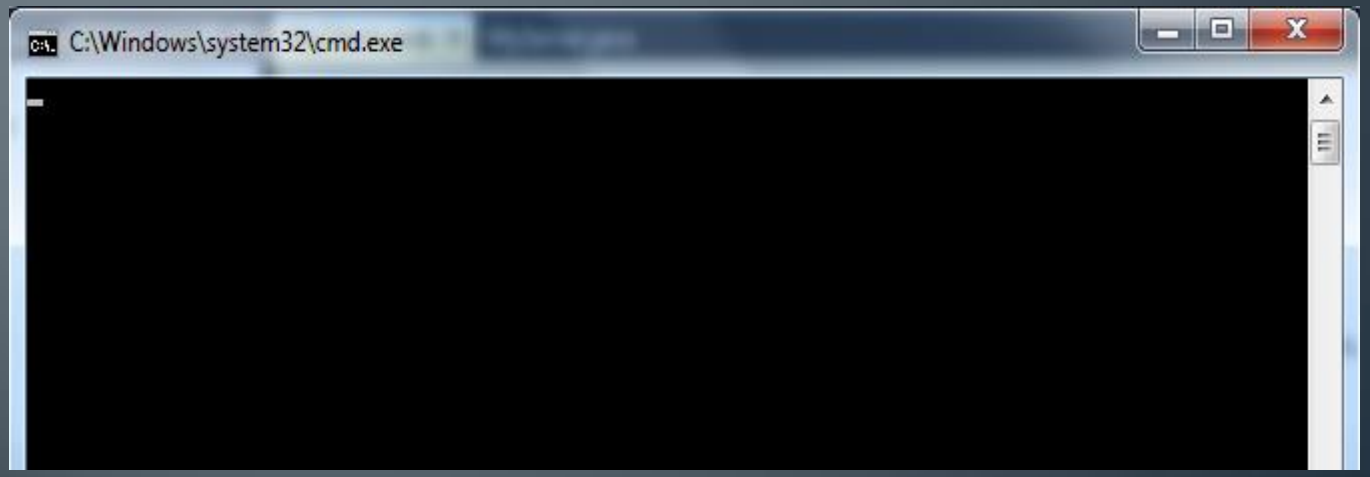
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Admin>cd documents

C:\Users\Admin\Documents>java WriteServer 1
Hai from WriteSever
```

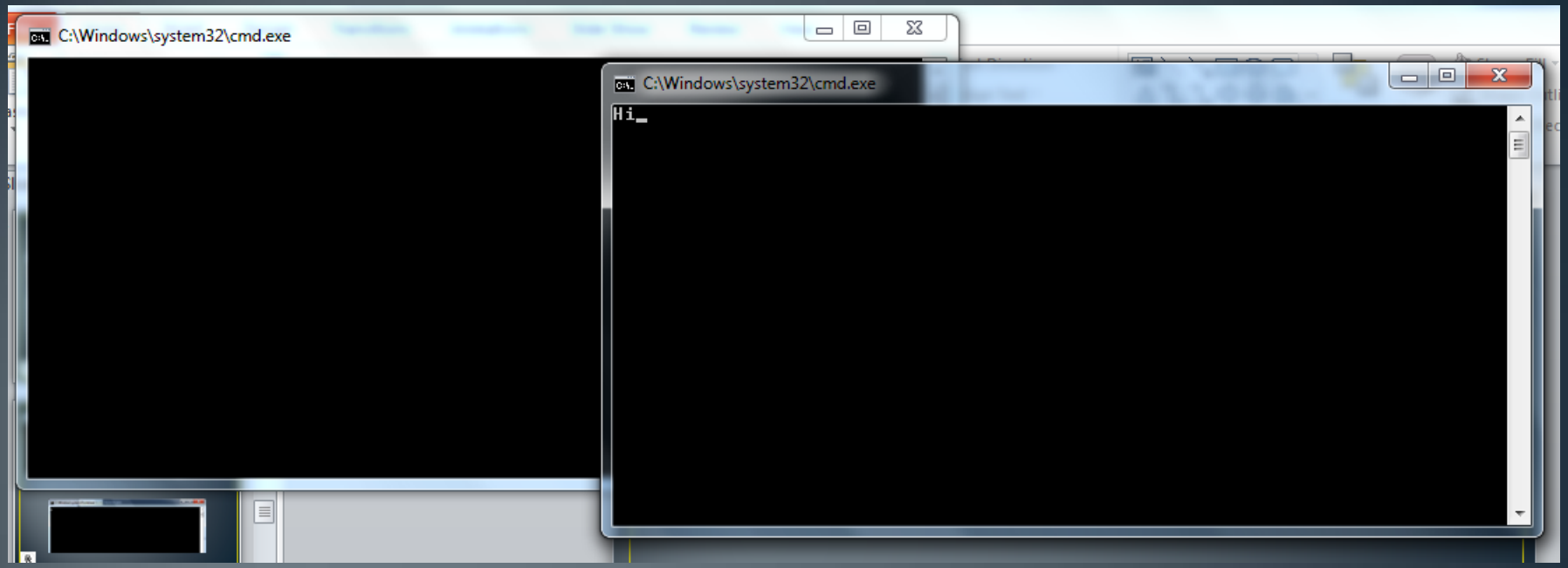
Java Socket Programming (Read-Write both)

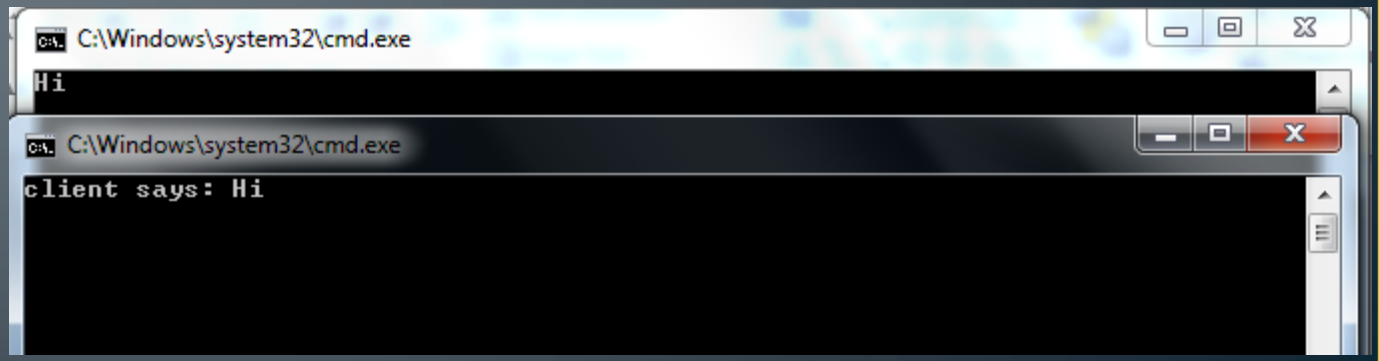
```
import java.net.*;
import java.io.*;
class MyServer{
public static void main(String args[])throws
Exception{
ServerSocket ss=new ServerSocket(3333);
Socket s=ss.accept();

DataInputStream din=new
DataInputStream(s.getInputStream());

DataOutputStream dout=new
DataOutputStream(s.getOutputStream());

BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));

String str="",str2="";
while(!str.equals("stop")){
str=din.readUTF();
System.out.println("client says: "+str);
str2=br.readLine();
dout.writeUTF(str2);
dout.flush();
}
din.close();
s.close();
ss.close();
}}
```

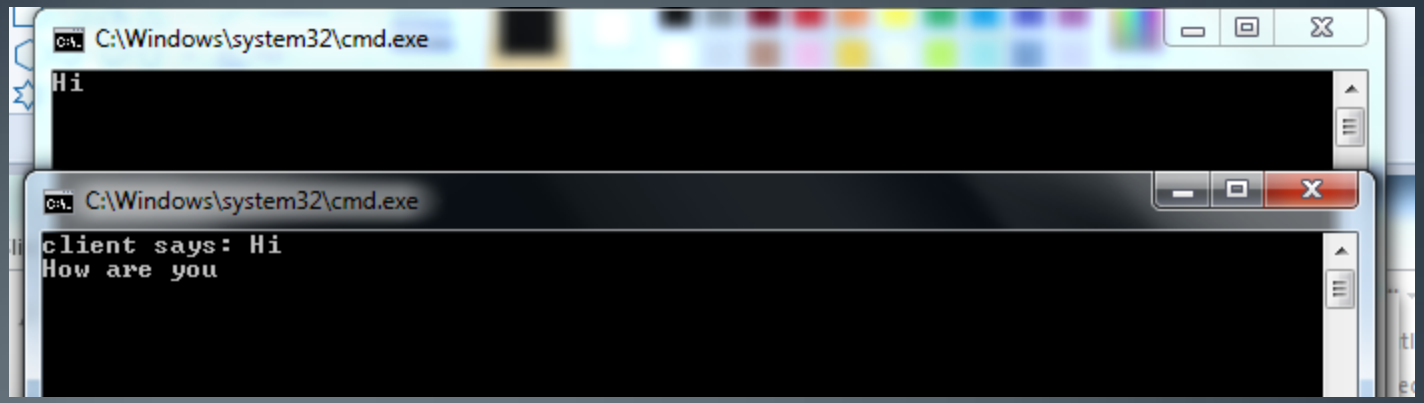
```
import java.net.*;
import java.io.*;
class MyClient{
public static void main(String args[])throws Exception{
Socket s=new Socket("localhost",3333);
DataInputStream din=new DataInputStream(s.getInputStream());
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
  String str="",str2="";
while(!str.equals("stop")){
str=br.readLine();
dout.writeUTF(str);
dout.flush();
str2=din.readUTF();
System.out.println("Server says: "+str2);  }
  dout.close();
s.close();
}}
```

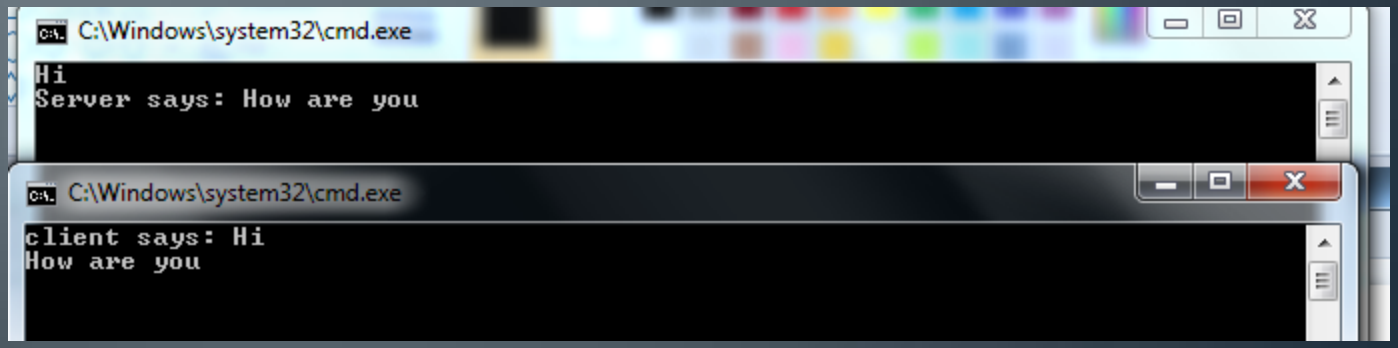

C:\Windows\system32\cmd.exe


C:\Windows\system32\cmd.exe


C:\Windows\system32\cmd.exe
C:\Windows\system32\cmd.exe
Hi


C:\Windows\system32\cmd.exe
Hi
C:\Windows\system32\cmd.exe
client says: Hi


C:\Windows\system32\cmd.exe
Hi
C:\Windows\system32\cmd.exe
client says: Hi
How are you


C:\Windows\system32\cmd.exe
Hi
Server says: How are you
C:\Windows\system32\cmd.exe
client says: Hi
How are you

Applets

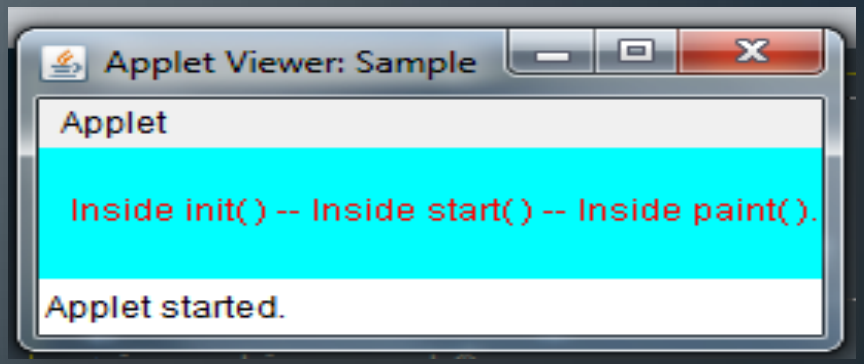
Two Types of Applets

- The first are those based directly on the **Applet** class
- These applets use the Abstract Window Toolkit (AWT) to provide the graphical user interface
- 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**, which inherits **Applet**.
- 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.
- Swing-based applets are now the most popular.
- Traditional AWT-based applets are still used, especially when only a very simple user interface is required.
- Both AWT- and Swing-based applets are valid.
- Applet tag

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

Sample Program

```
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>            */
public class Sample extends Applet{
String msg;
// set the foreground and background colors.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
msg = "Inside init( ) --";
// Initialize the string to be displayed.
public void start() {
msg += " Inside start( ) --";
}
// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```



Applet Event Handling

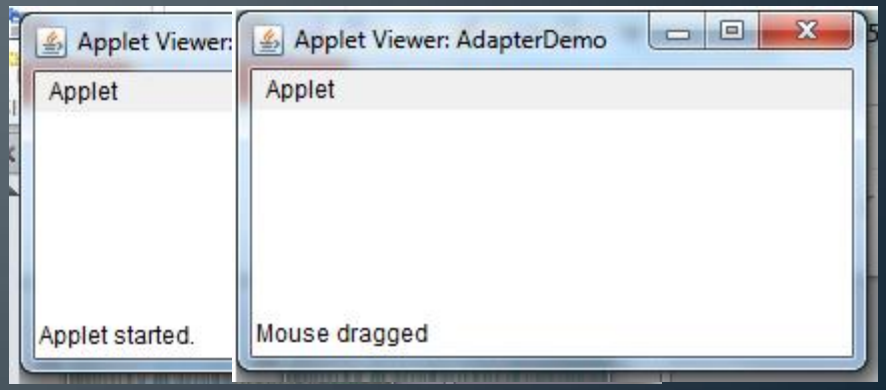
The Delegation Event Model

- 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, 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.

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.
- You 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 you are interested.
- For example, the MouseMotionAdapter class has two methods, mouseDragged( ) and mouseMoved( ), which are the methods defined by the MouseMotionListener interface.
- If you were interested in only mouse drag events, then you could simply extend MouseMotionAdapter and override mouseDragged( ). The empty implementation of mouseMoved( ) would handle the mouse motion events for you.

```
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;     }
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");     }      }
```



Introducing the AWT

Sources of Events

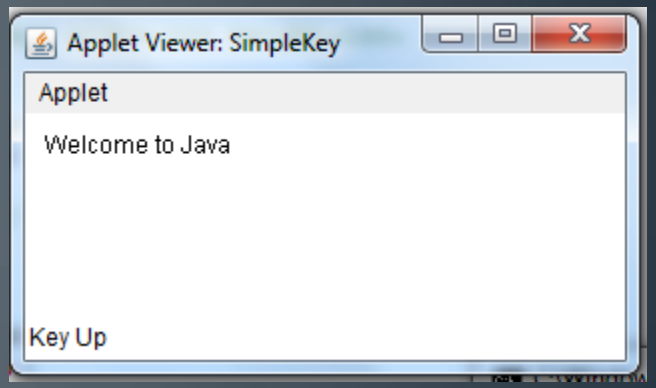
Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events 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.

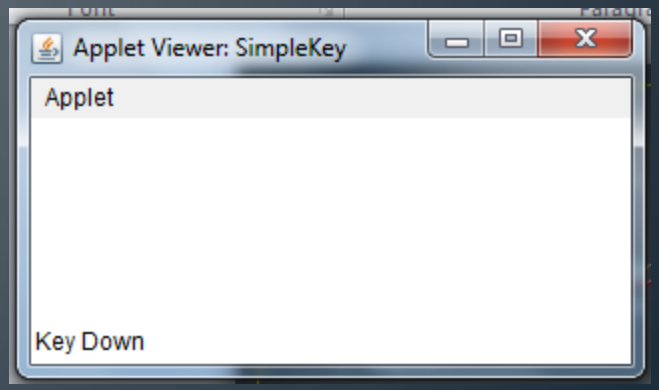
Event Listener Interfaces

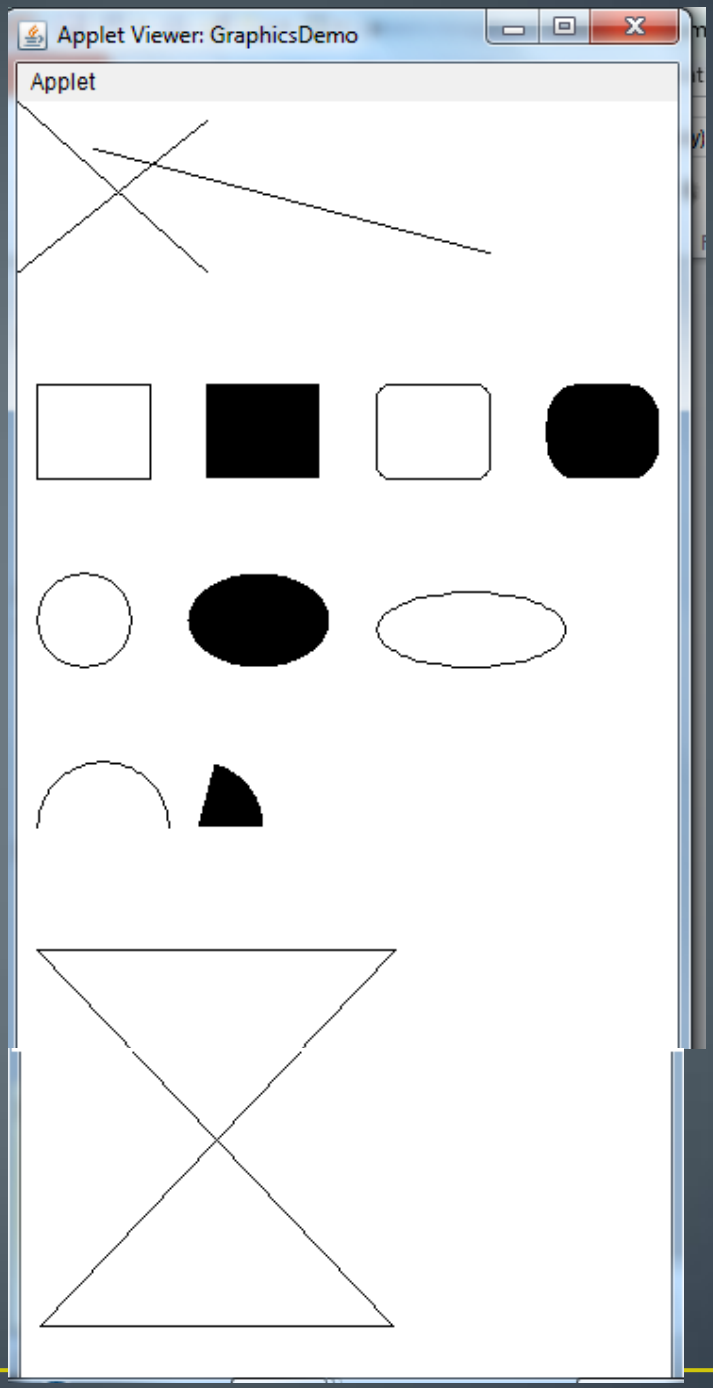
Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize 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.

```
import java.awt.*;
import java.applet.*;
/*    <applet code="GraphicsDemo"
width=350 height=700>
</applet>      */
public class GraphicsDemo extends
Applet {
public void paint(Graphics g) {
// Draw lines.
g.drawLine(0, 0, 100, 90);
g.drawLine(0, 90, 100, 10);
g.drawLine(40, 25, 250, 80);
// Draw rectangles.
g.drawRect(10, 150, 60, 50);
g.fillRect(100, 150, 60, 50);
g.drawRoundRect(190, 150, 60, 50, 15,
15);
g.fillRoundRect(280, 150, 60, 50, 30, 40);
// Draw Ellipses and Circles
g.drawOval(10, 250, 50, 50);
g.fillOval(90, 250, 75, 50);
g.drawOval(190, 260, 100, 40);
// Draw Arcs
g.drawArc(10, 350, 70, 70, 0, 180);
g.fillArc(60, 350, 70, 70, 0, 75);
// Draw a polygon
int xpoints[] = {10, 200, 10, 200, 10};
int ypoints[] = {450, 450, 650, 650,
450};
int num = 5;
g.drawPolygon(xpoints, ypoints, num);
}            }
```

```
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);        } }
```

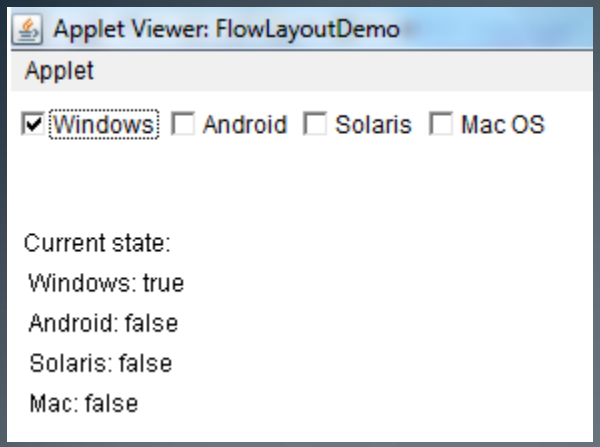


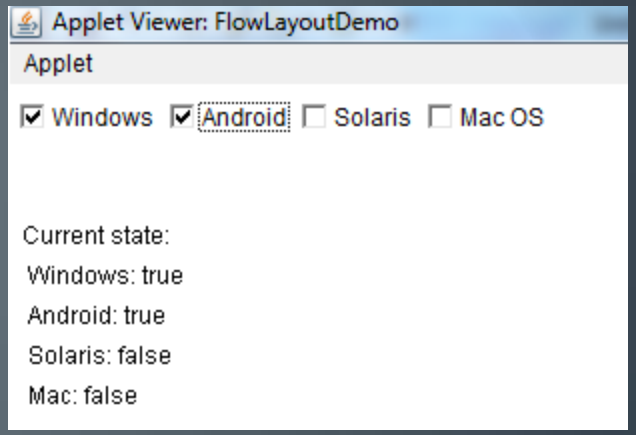



Applet Viewer: GraphicsDemo
Applet

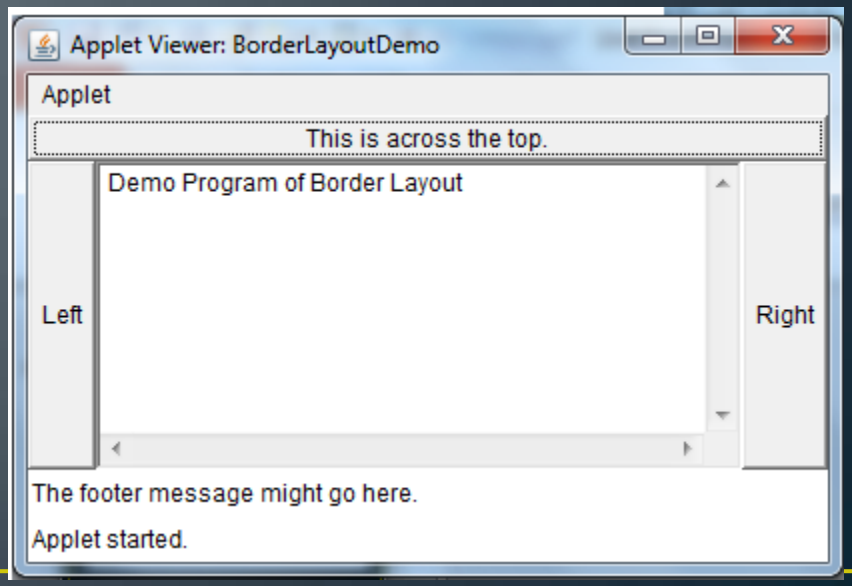
Using AWT Controls, Layout Managers, and Menus

```
// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*    <applet code="FlowLayoutDemo"
width=240 height=200>   </applet>   */
public class FlowLayoutDemo extends Applet
implements ItemListener {
String msg = "";
Checkbox windows, android, solaris, mac;
public void init() {
// set left-aligned flow layout
setLayout(new
FlowLayout(FlowLayout.LEFT));
windows = new Checkbox("Windows", null,
true);
android = new Checkbox("Android");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");
add(windows);
add(android);
add(solaris);
add(mac);
// register to receive item events
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);      }
// Repaint when status of a check box
changes.
public void itemStateChanged(ItemEvent ie) {
repaint();      }
// Display current state of the check boxes.
public void paint(Graphics g) {
msg = "Current state: ";
g.drawString(msg, 6, 80);
msg = " Windows: " + windows.getState();
g.drawString(msg, 6, 100);
msg = " Android: " + android.getState();
g.drawString(msg, 6, 120);
msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
msg = " Mac: " + mac.getState();
g.drawString(msg, 6, 160);
}     }
```

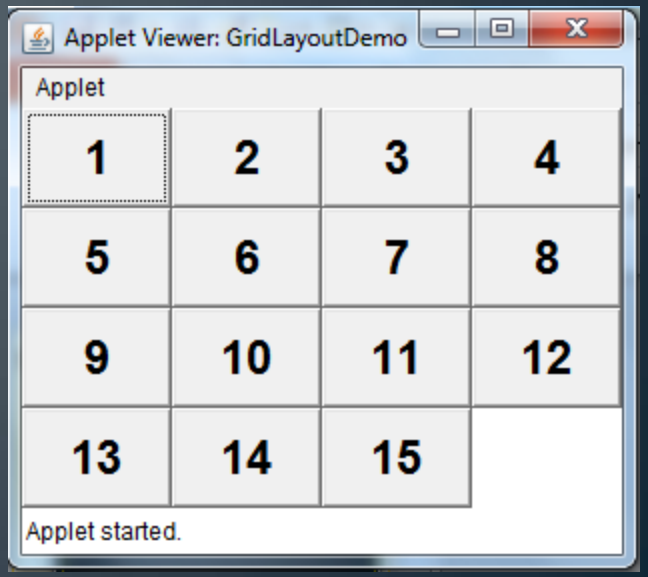

Applet Viewer: FlowLayoutDemo
Applet
Windows
Android
Solaris
Mac OS
Current state:
Windows: true
Android: false
Solaris: false
Mac: false


Applet Viewer: FlowLayoutDemo
Applet
Windows
Android
Solaris
Mac OS
Current state:
Windows: true
Android: true
Solaris: false
Mac: false

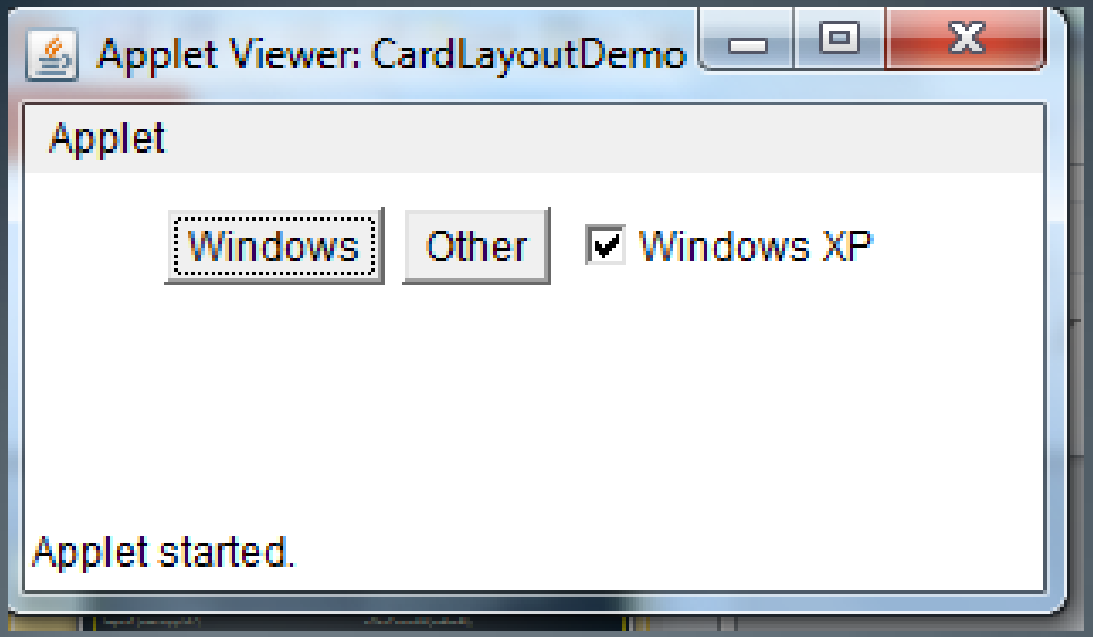
```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
public class BorderLayoutDemo extends Applet {
public void init() {
setLayout(new BorderLayout());
add(new Button("This is across the top."),
BorderLayout.NORTH);
add(new Label("The footer message might go here."),
BorderLayout.SOUTH);
add(new Button("Right"), BorderLayout.EAST);
add(new Button("Left"), BorderLayout.WEST);
String msg = " Demo Program of Border Layout";
add(new TextArea(msg), BorderLayout.CENTER);
}
}
```

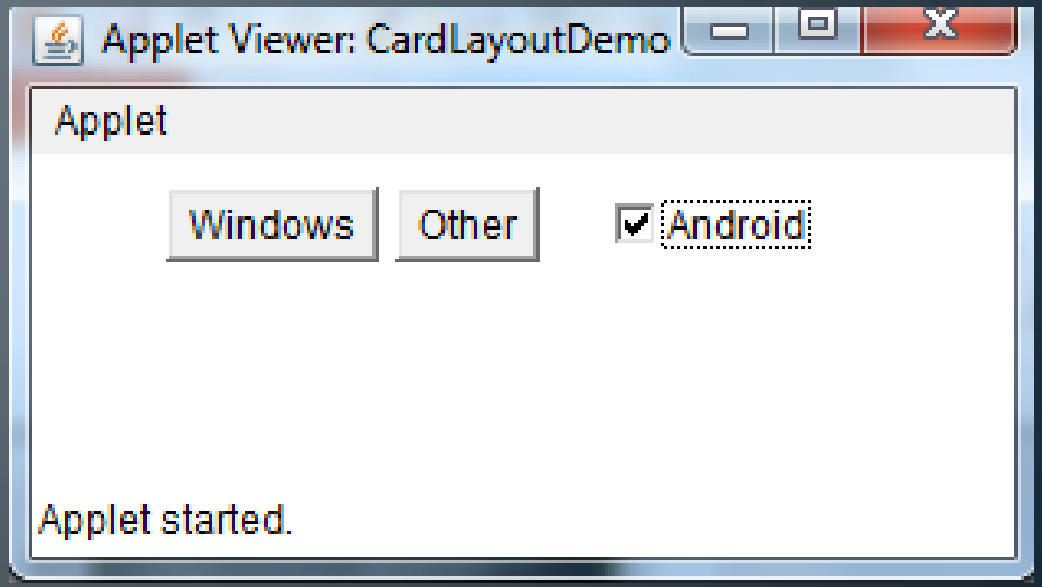


```
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
public class GridLayoutDemo extends Applet {
static final int n = 4;
public void init() {
setLayout(new GridLayout(n, n));
setFont(new Font("SansSerif", Font.BOLD, 24));
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));
}
}
}
```

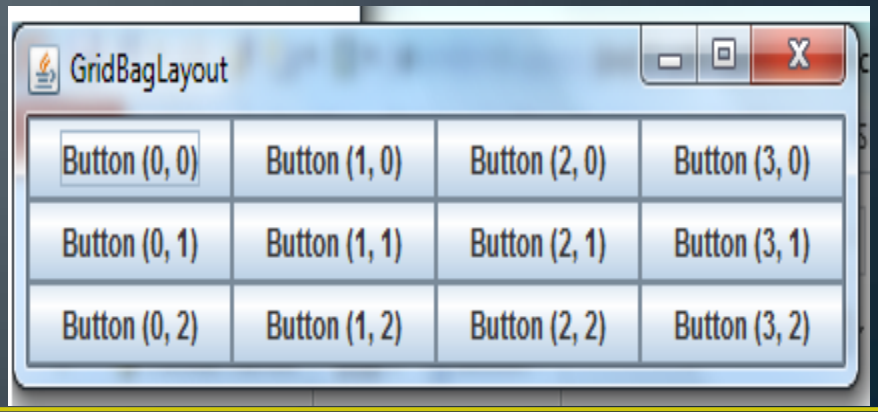


```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*   <applet code="CardLayoutDemo" width=300 height=100>    </applet>    */
public class CardLayoutDemo extends Applet
implements ActionListener, MouseListener {
Checkbox windowsXP, windows7, windows8, android, solaris, mac;
Panel osCards;
CardLayout cardLO;
Button Win, Other;
public void init() {
Win = new Button("Windows");
Other = new Button("Other");
add(Win);
add(Other);
cardLO = new CardLayout();
osCards = new Panel();
osCards.setLayout(cardLO);
// set panel layout to card layout
windowsXP = new Checkbox("Windows XP", null, true);
android = new Checkbox("Android");
// add Windows check boxes to a panel
Panel winPan = new Panel();
winPan.add(windowsXP);
// Add other OS check boxes to a panel
Panel otherPan = new Panel();
otherPan.add(android);
// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");
// add cards to main applet panel
add(osCards);
// register to receive action events
Win.addActionListener(this);
Other.addActionListener(this);
// register mouse events
addMouseListener(this);    }
// Cycle through panels.
public void mousePressed(MouseEvent me) {
cardLO.next(osCards);     }
// Provide empty implementations for the other MouseListener methods.
public void mouseClicked(MouseEvent me) {          }
public void mouseEntered(MouseEvent me) {          }
public void mouseExited(MouseEvent me) {           }
public void mouseReleased(MouseEvent me) {         }
public void actionPerformed(ActionEvent ae) {
                if(ae.getSource() == Win) {
cardLO.show(osCards, "Windows");     }
else {          cardLO.show(osCards, "Other");   }  }   }
```


Applet Viewer: CardLayoutDemo
Applet
Windows
Other
Windows XP
Applet started.

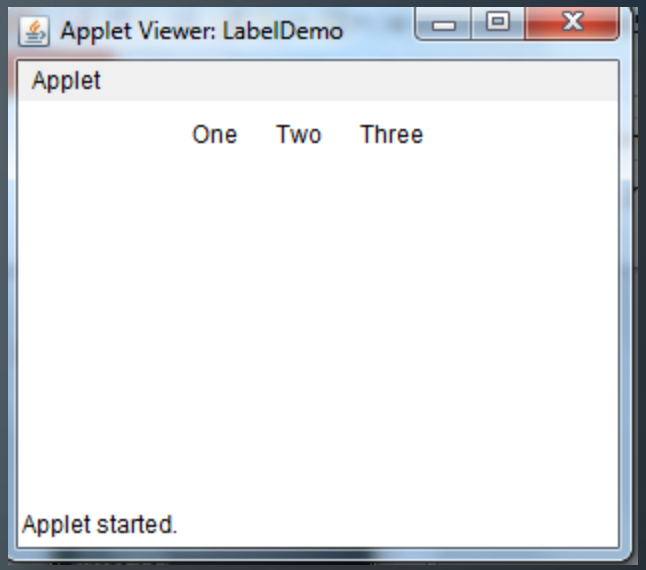

Applet Viewer: CardLayoutDemo
Applet
Windows
Other
Android
Applet started.

```
import java.awt.Container;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class GBLayout {
  public static void main(String[] args) {
    String title = "GridBagLayout";
    JFrame frame = new JFrame(title);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = frame.getContentPane();
    contentPane.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    for (int y = 0; y < 3; y++) {
      for (int x = 0; x < 4; x++) {
        gbc.gridx = x;
        gbc.gridy = y;
        String text = "Button (" + x + ", " + y + ")";
        contentPane.add(new JButton(text), gbc);
      }
```



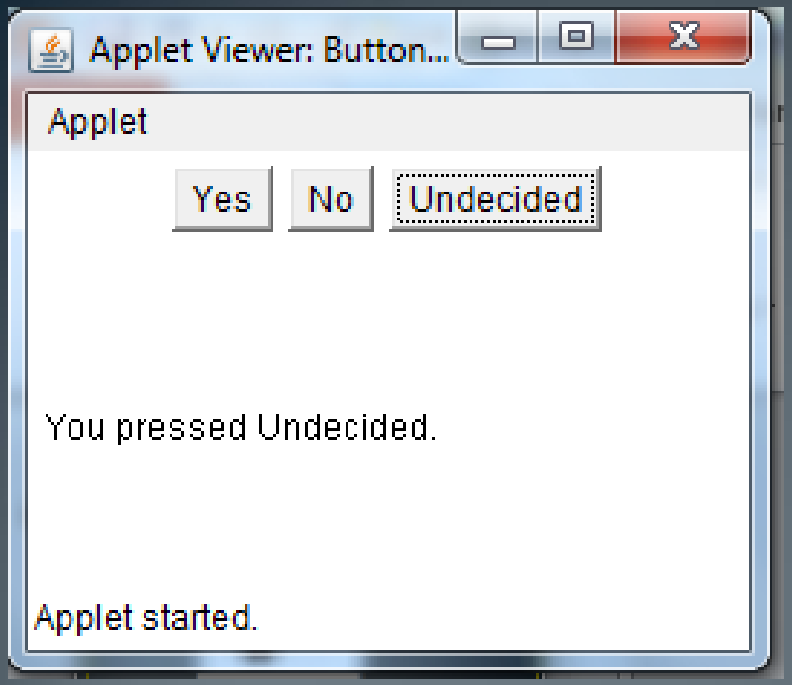
Demonstrate Labels

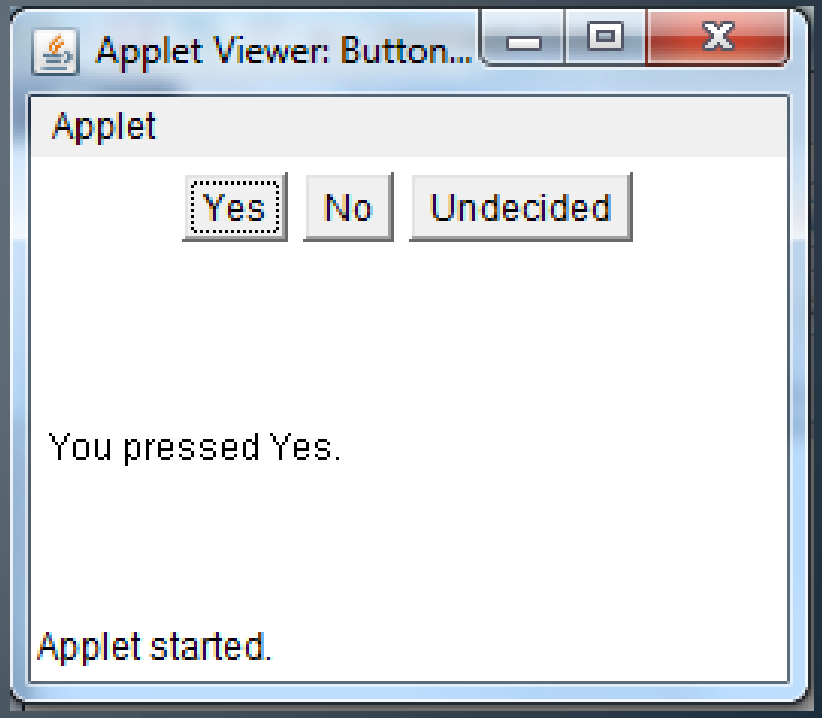
```
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet {
public void init() {
Label one = new Label("One");
Label two = new Label("Two");
Label three = new Label("Three");
// add labels to applet window
add(one);
add(two);
add(three);
}
}
```



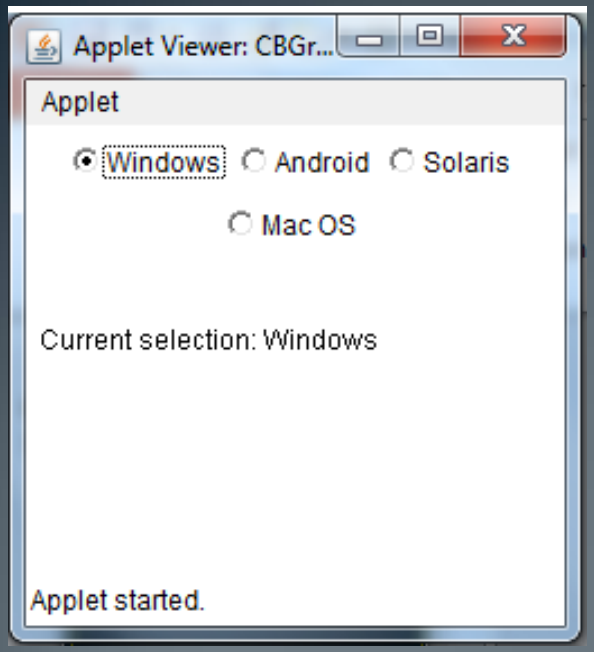
Demonstrate Buttons

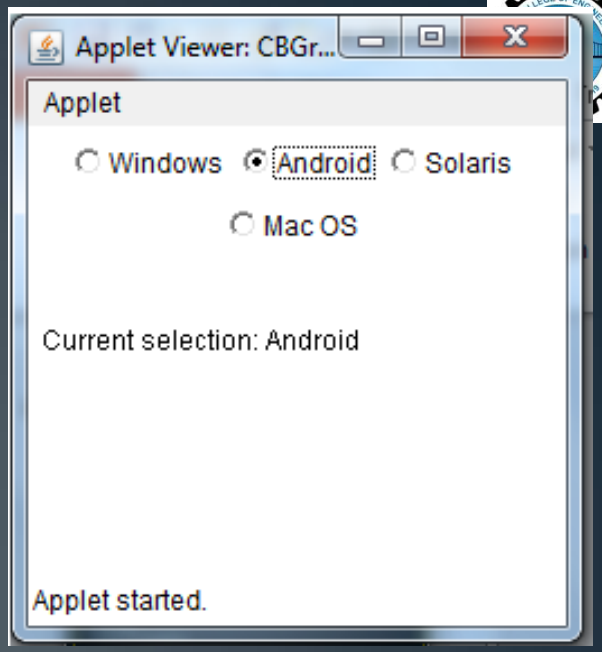
```
import java.awt.*;
 import java.awt.event.*;
 import java.applet.*;
 /*     <applet code="ButtonDemo"
width=250 height=150>
 </applet>      */
 public class ButtonDemo extends Applet
implements ActionListener {
 String msg = "";
 Button yes, no, maybe;
 public void init() {
 yes = new Button("Yes");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

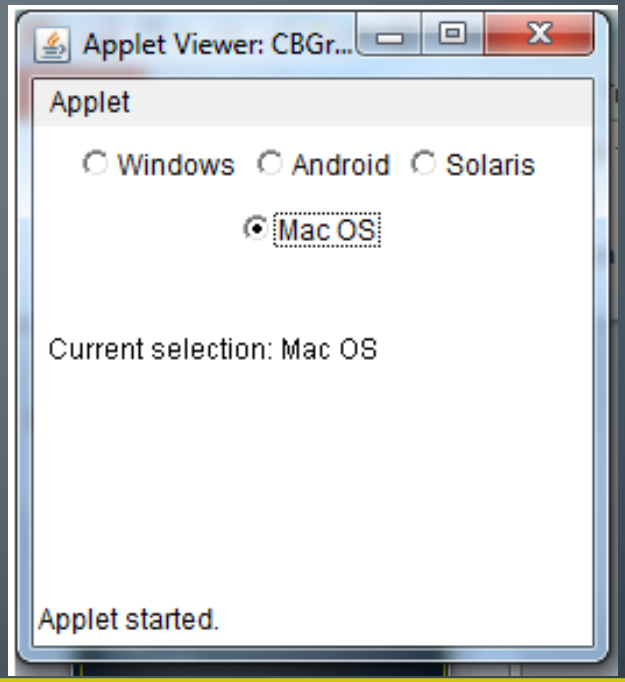

Applet Viewer: Button...
Applet
Yes
No
Undecided
You pressed Undecided.
Applet started.


Applet Viewer: Button...
Applet
Yes
No
Undecided
You pressed Yes.
Applet started.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*  <applet code="CBGroup" width=240 height=200>
</applet>   */
public class CBGroup extends Applet implements ItemListener {
String msg = "";
Checkbox windows, android, solaris, mac;
CheckboxGroup cbg;
public void init() {
cbg = new CheckboxGroup();
windows = new Checkbox("Windows", cbg, true);
android = new Checkbox("Android", cbg, false);
solaris = new Checkbox("Solaris", cbg, false);
mac = new Checkbox("Mac OS", cbg, false);
add(windows);
add(android);
add(solaris);
add(mac);
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g) {
msg = "Current selection: ";
msg += cbg.getSelectedCheckbox().getLabel();
g.drawString(msg, 6, 100);
}
}
```

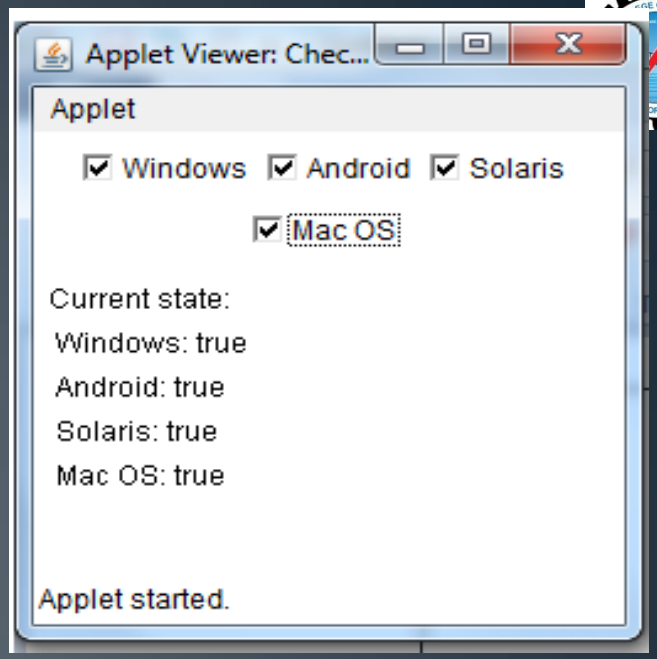

Applet Viewer: CBGr...
Applet
Windows
Android
Solaris
Mac OS
Current selection: Windows
Applet started.

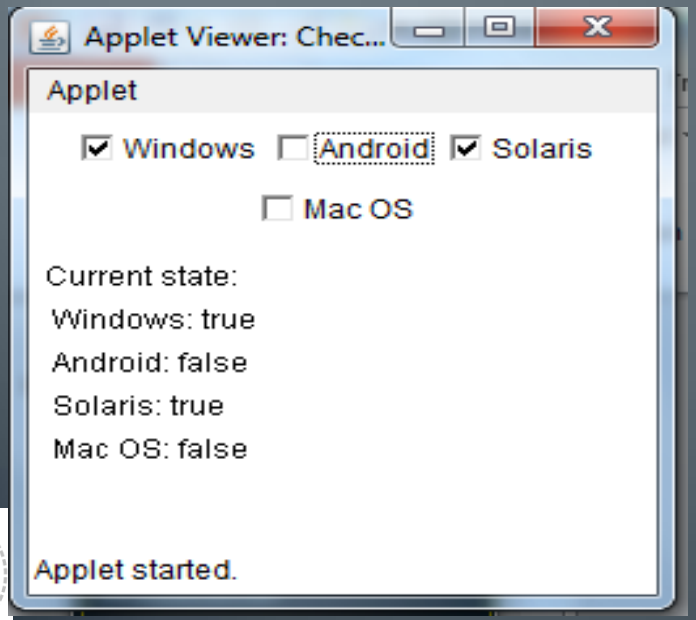

Applet Viewer: CBGr...
Applet
Windows
Android
Solaris
Mac OS
Current selection: Android
Applet started.

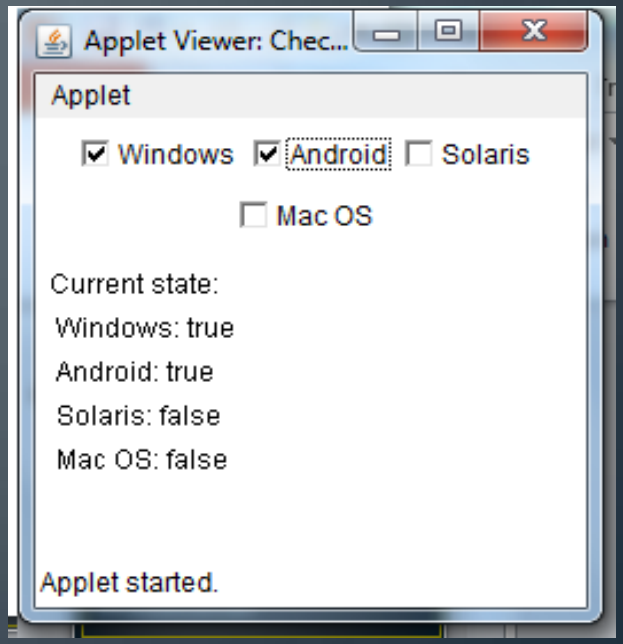

Applet Viewer: CBGr...
Applet
Windows
Android
Solaris
Mac OS
Current selection: Mac OS
Applet started.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*   <applet code="CheckboxDemo"
width=240 height=200>     </applet>     */
public class CheckboxDemo extends Applet
implements ItemListener {
String msg = "";
Checkbox windows, android, solaris, mac;
public void init() {
windows = new Checkbox("Windows", null,
true);
android = new Checkbox("Android");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");
add(windows);
add(android);
add(solaris);
add(mac);
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
public void itemStateChanged(ItemEvent ie)
repaint();   }
// Display current state of the check boxes.
public void paint(Graphics g) {
msg = "Current state: ";
g.drawString(msg, 6, 80);
msg = " Windows: " + windows.getState();
g.drawString(msg, 6, 100);
msg = " Android: " + android.getState();
g.drawString(msg, 6, 120);
msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
msg = " Mac OS: " + mac.getState();
g.drawString(msg, 6, 160);
}
}
```

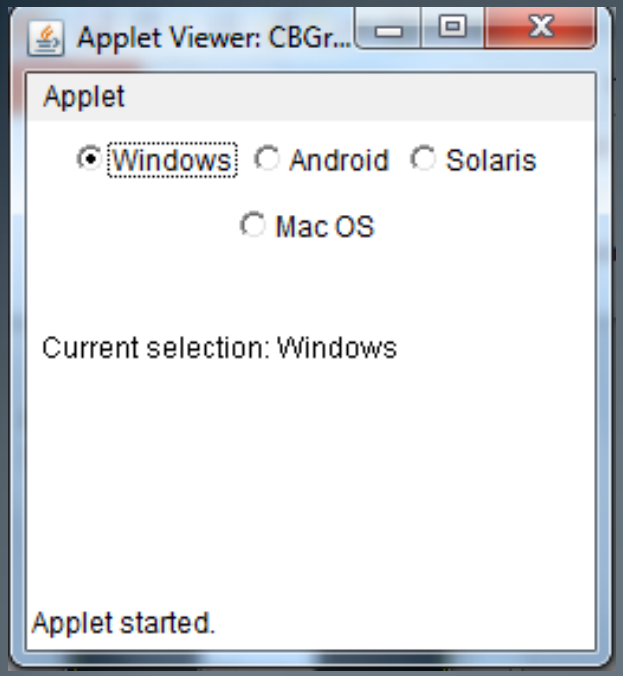

Applet Viewer: Chec...
Applet
Windows
Android
Solaris
Mac OS
Current state:
Windows: true
Android: false
Solaris: false
Mac OS: false
Applet started.

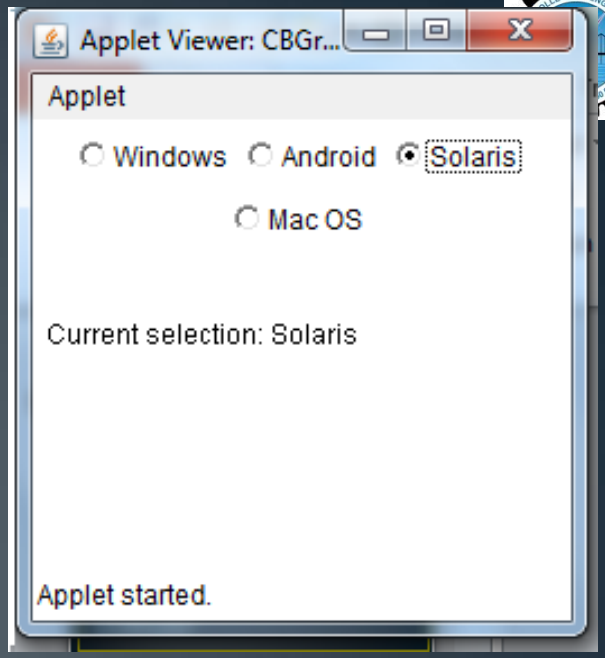

Applet Viewer: Chec...
Applet
Windows
Android
Solaris
Mac OS
Current state:
Windows: true
Android: true
Solaris: true
Mac OS: true
Applet started.

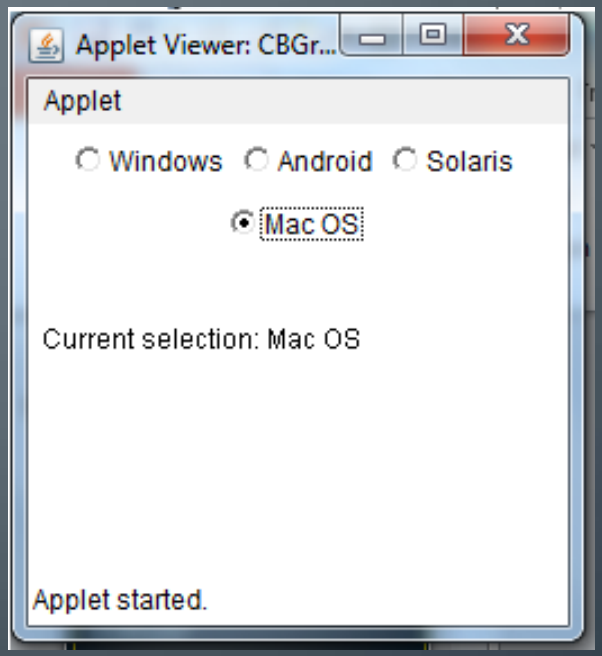

Applet Viewer: Chec...
Applet
Windows
Android
Solaris
Mac OS
Current state:
Windows: true
Android: false
Solaris: true
Mac OS: false
Applet started.


Applet Viewer: Chec...
Applet
Windows
Android
Solaris
Mac OS
Current state:
Windows: true
Android: true
Solaris: false
Mac OS: false
Applet started.

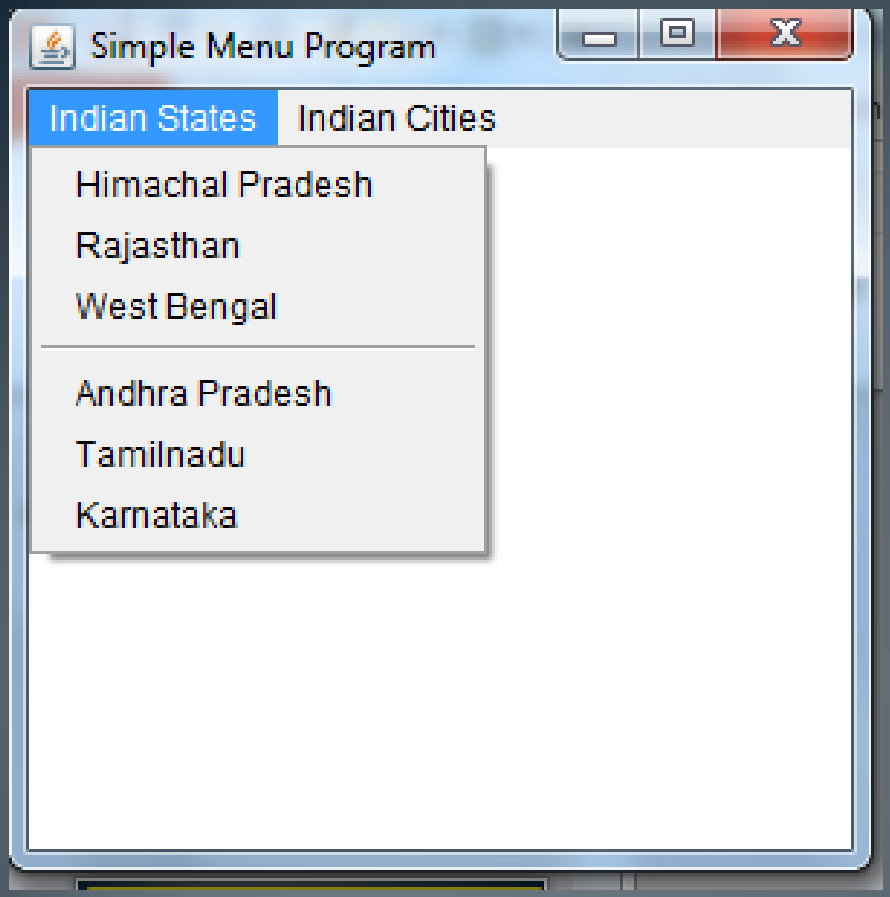
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*     <applet code="CBGroup"
width=240 height=200>    </applet>
*/
public class CBGroup1 extends Applet
implements ItemListener {
String msg = "";
Checkbox windows, android, solaris, mac;
CheckboxGroup cbg;
public void init() {
cbg = new CheckboxGroup();
windows = new Checkbox("Windows",
cbg, true);
android = new Checkbox("Android", cbg,
false);
solaris = new Checkbox("Solaris", cbg,
false);
mac = new Checkbox("Mac OS", cbg,
false);
add(windows);
add(android);
add(solaris);
add(mac);
 windows.addItemListener(this);
 android.addItemListener(this);
 solaris.addItemListener(this);
 mac.addItemListener(this);     }
 public void itemStateChanged(ItemEvent
 ie) {
 repaint();              }
 // Display current state of the check
 boxes.
 public void paint(Graphics g) {
 msg = "Current selection: ";
 msg +=
 cbg.getSelectedCheckbox().getLabel();
 g.drawString(msg, 6, 100);
 }
 }
```

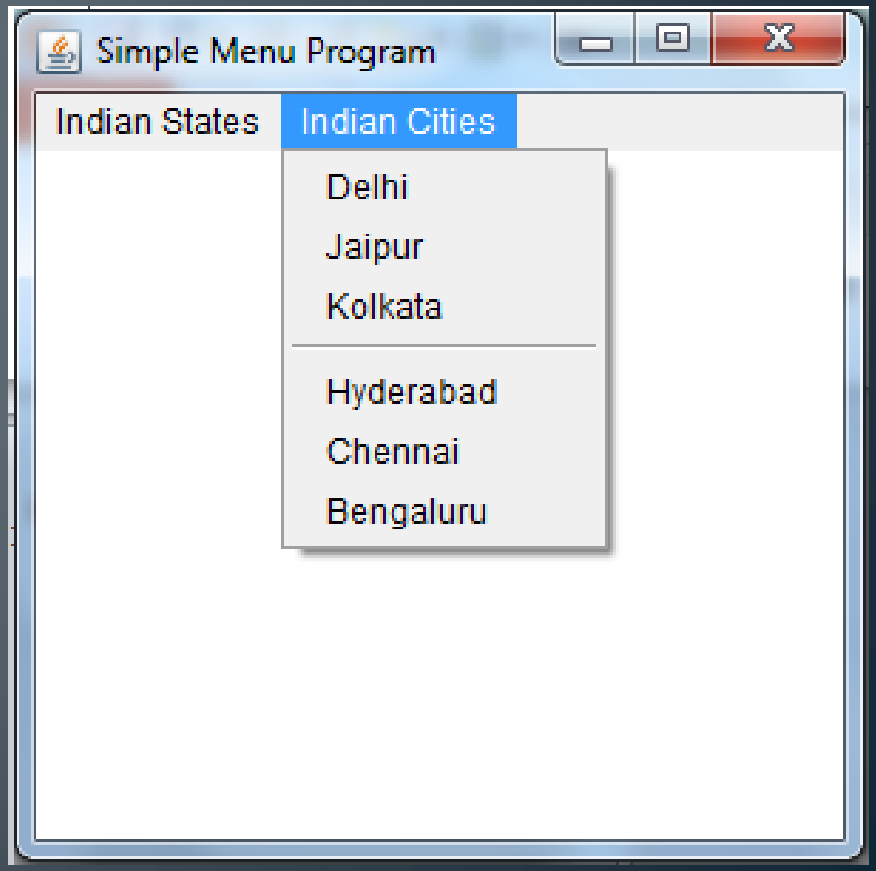

Applet Viewer: CBGr...
Applet
Windows
Android
Solaris
Mac OS
Current selection: Windows
Applet started.


Applet Viewer: CBGr...
Applet
Windows
Android
Solaris
Mac OS
Current selection: Solaris
Applet started.


Applet Viewer: CBGr...
Applet
Windows
Android
Solaris
Mac OS
Current selection: Mac OS
Applet started.

```
import java.awt.*;
import java.awt.event.*;
 public class SimpleMenuExample extends Frame
implements ActionListener   {
  Menu states, cities;
  public SimpleMenuExample()     {
MenuBar mb = new MenuBar();
// begin with creating menu bar
    setMenuBar(mb);
                // add menu bar to frame
     states = new Menu("Indian States");
                // create menus
    cities = new Menu("Indian Cities");
     mb.add(states);
                // add menus to menu bar
    mb.add(cities);
     states.addActionListener(this);
cities.addActionListener(this);
   states.add(new MenuItem("Himachal Pradesh"));
    states.add(new MenuItem("Rajasthan"));
    states.add(new MenuItem("West Bengal"));
    states.addSeparator();
    states.add(new MenuItem("Andhra Pradesh"));
 states.add(new MenuItem("Tamilnadu"));
    states.add(new MenuItem("Karnataka"));
     cities.add(new MenuItem("Delhi"));
    cities.add(new MenuItem("Jaipur"));
    cities.add(new MenuItem("Kolkata"));
    cities.addSeparator();
cities.add(new MenuItem("Hyderabad"));
    cities.add(new MenuItem("Chennai"));
    cities.add(new MenuItem("Bengaluru"));
     setTitle("Simple Menu Program");
// frame creation methods
    setSize(300, 300);
    setVisible(true);        }
  public void actionPerformed(ActionEvent e)    {
    String str = e.getActionCommand();
// know the menu item selected by the user
    System.out.println("You selected " + str);      }
  public static void main(String args[])
  {
    new SimpleMenuExample();
  }
}
```


Simple Menu Program
Indian States
Indian Cities
Himachal Pradesh
Rajasthan
West Bengal
Andhra Pradesh
Tamilnadu
Karnataka


Simple Menu Program
Indian States
Indian Cities
Delhi
Jaipur
Kolkata
Hyderabad
Chennai
Bengaluru

Introduction to Swings

The Origins of Swing

- Swing did not exist in the early days of Java.
- Swings was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit.
- The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
- One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers.
- This means that the look and feel of a component is defined by the platform, not by Java.
- Because the AWT components use native code resources, they are referred to as heavyweight.
- The use of native peers led to several problems.
- First, because of variations between operating systems, a component might look, or even act, differently on different platforms.
- Second, the look and feel of each component was fixed and could not be changed.
- Third, the use of heavyweight components caused some frustrating restrictions.

Swing Is Built on the AWT

- Swing eliminates a number of the limitations inherent in the AWT, Swing *does not* replace it.
- Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java.
- Swing also uses the same event handling mechanism as the AWT.

Two Key Swing Features

Swing Components Are Lightweight

- This means that they are written entirely in Java and do not map directly to platform-specific peers.
- Swing components are more efficient and more flexible.
- Lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system.
- Each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

- Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing.
- This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.
- Separating out the look and feel provides a significant advantage: it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component.

The MVC Connection

A visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

- One component architecture has proven itself to be exceptionally effective: *Model-View-Controller*, or MVC for short.

Components

- A Swing GUI consists of two key items: components and containers.
- All containers are also components.
- The difference between the two is : As the term is commonly used, a component is an independent visual control, such as a push button or slider. A container holds a group of components.
- A container is a special type of component that is designed to hold other components.
- All Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers.

Containers

- Swing defines two types of containers. The first are top-level containers: JFrame, JApplet, JWindow, and JDialog.
- A top-level container must be at the top of a containment hierarchy. The one most commonly used for applications is JFrame. The one used for applets is JApplet.
- The second type of containers supported by Swing are lightweight containers. Lightweight containers do inherit JComponent.
- Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container.

Components

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Containers(1)

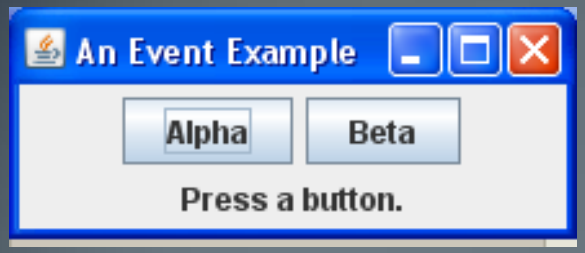
- Swing defines two types of containers.
- The first are top-level containers: JFrame, JApplet, JWindow, and JDialog.
- These containers do not inherit JComponent.
- These inherit the AWT classes Component and Container.
- Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight.
- This makes the top-level containers a special case in the Swing component library.
- As the name implies, a top-level container must be at the top of a containment hierarchy.
- A top-level container is not contained within any other container.
- The one most commonly used for applications is JFrame.
- The one used for applets is JApplet.

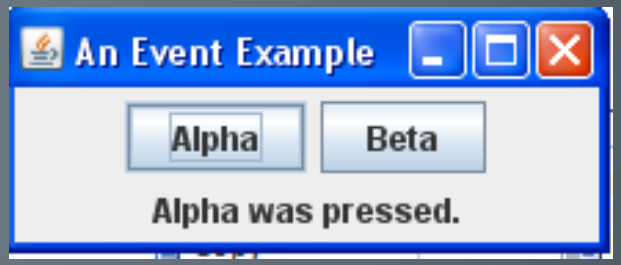
Containers(2)

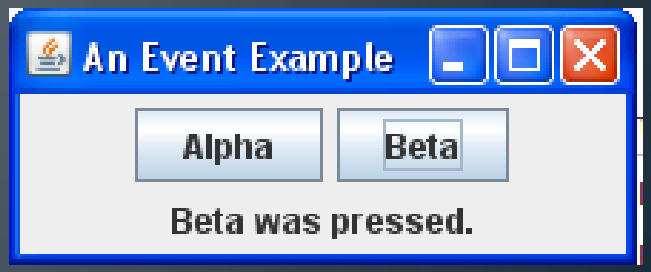
- The second type of containers supported by Swing are lightweight containers.
- Lightweight containers *do* inherit **JComponent**.
- An example of a lightweight container is **JPanel,** which is a general-purpose container.
- Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container.
- Thus, you can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemo {
JLabel jlab;
EventDemo() {
// Create a new JFrame container.
JFrame jfrm = new JFrame("An Event Example");
// Specify FlowLayout for the layout manager.
jfrm.setLayout(new FlowLayout());
// Give the frame an initial size.
jfrm.setSize(220, 90);
// Terminate the program when the user closes
the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
// Make two buttons.
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent ae) {
Text("Alpha was pressed.");
});
// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent ae) {
jlab.setText("Beta was pressed.");
}
});
// Add the buttons to the content pane.
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
// Create a text-based label.
jlab = new JLabel("Press a button.");
// Add the label to the content pane.
jfrm.add(jlab);
// Display the frame.
jfrm.setVisible(true);
}
public static void main(String args[]) {
// Create the frame on the event dispatching
thread.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new EventDemo();
}
});
}
}
```

output


An Event Example
Alpha
Beta
Press a button.

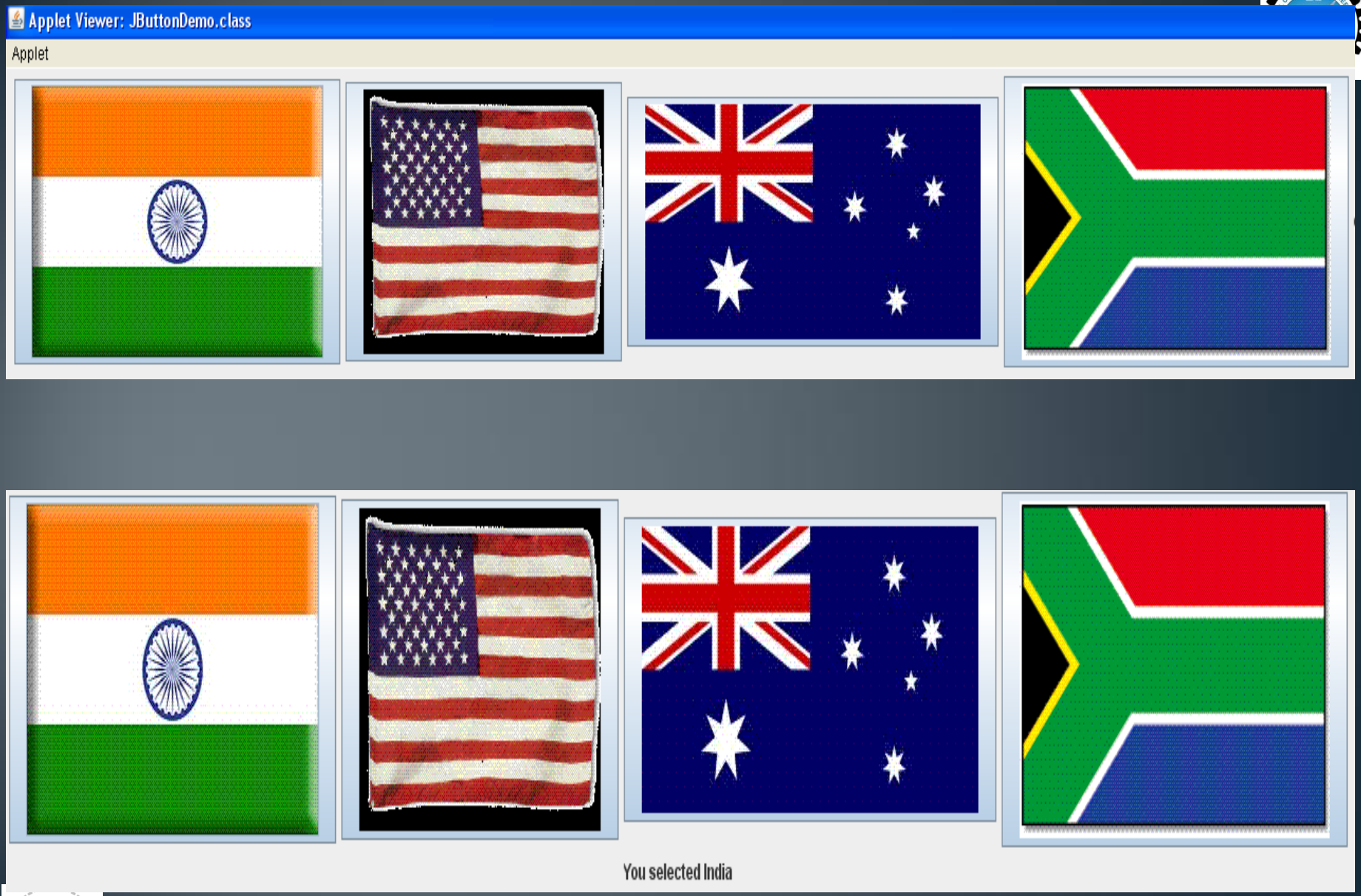

An Event Example
Alpha
Beta
Alpha was pressed.


An Event Example
Alpha
Beta
Beta was pressed.

```
// Demonstrate JLabel and ImageIcon.
import java.awt.*;
import javax.swing.*;
/*          <applet code="JLabelDemo" width=250 height=200>
            </applet>*/
public class JLabelDemo extends JApplet {
public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();          }          }          );
} catch (Exception exc) {
System.out.println("Can't create because of " + exc);     }          }
private void makeGUI() {
// Create an icon.
ImageIcon ii = new ImageIcon("IndianFlag.jpg");
// Create a label.
JLabel jl = new JLabel("Jaihind", ii, JLabel.CENTER);
// Add the label to the content pane.
add(jl);     }          }
```

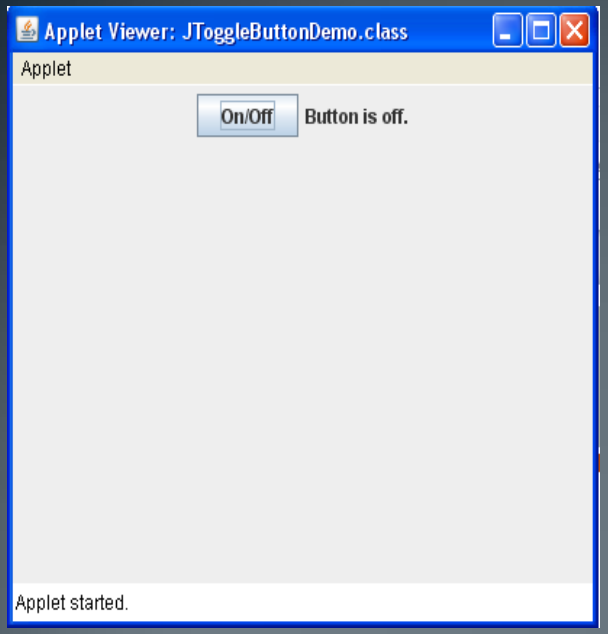

Applet Viewer: JLabelDemo
Applet
Jaihind
Applet started.

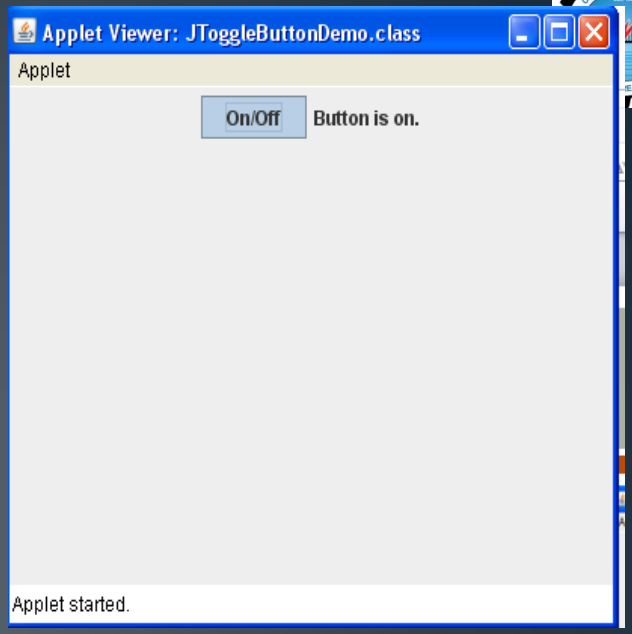
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*  <applet code="JButtonDemo" width=250
height=450>              </applet> */
public class JButtonDemo extends JApplet
implements ActionListener {
JLabel jlab;
public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();}              }              );              }
catch (Exception exc) {
System.out.println("Can't create because of " + exc);} }
private void makeGUI() {
// Change to flow layout.
setLayout(new FlowLayout());
ImageIcon france = new ImageIcon("indian.gif");
JButton jb = new JButton(france);
jb.setActionCommand("India");
jb.addActionListener(this);
add(jb);
ImageIcon germany = new
ImageIcon("america.gif");
jb = new JButton(germany);
jb.setActionCommand("Germany");
jb.addActionListener(this);
add(jb);
ImageIcon italy = new ImageIcon("australia.gif");
jb = new JButton(italy);
jb.setActionCommand("Italy");
jb.addActionListener(this);
add(jb);
ImageIcon japan = new
ImageIcon("southafrica.gif");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
add(jb);
// Create and add the label to content pane.
jlab = new JLabel("Choose a Flag");
add(jlab);   }
// Handle button events.
public void actionPerformed(ActionEvent ae) {
jlab.setText("You selected " +
ae.getActionCommand());
}              }
```

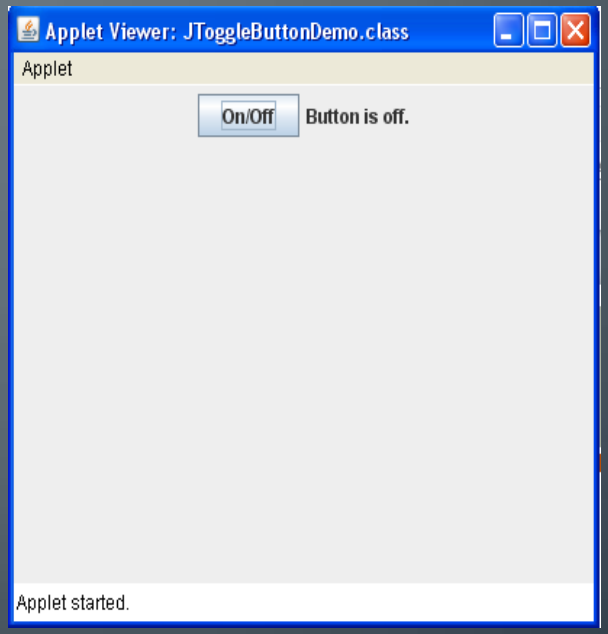

Applet Viewer: JButtonDemo.class
Applet
You selected India

Demonstrate JToggleButton.

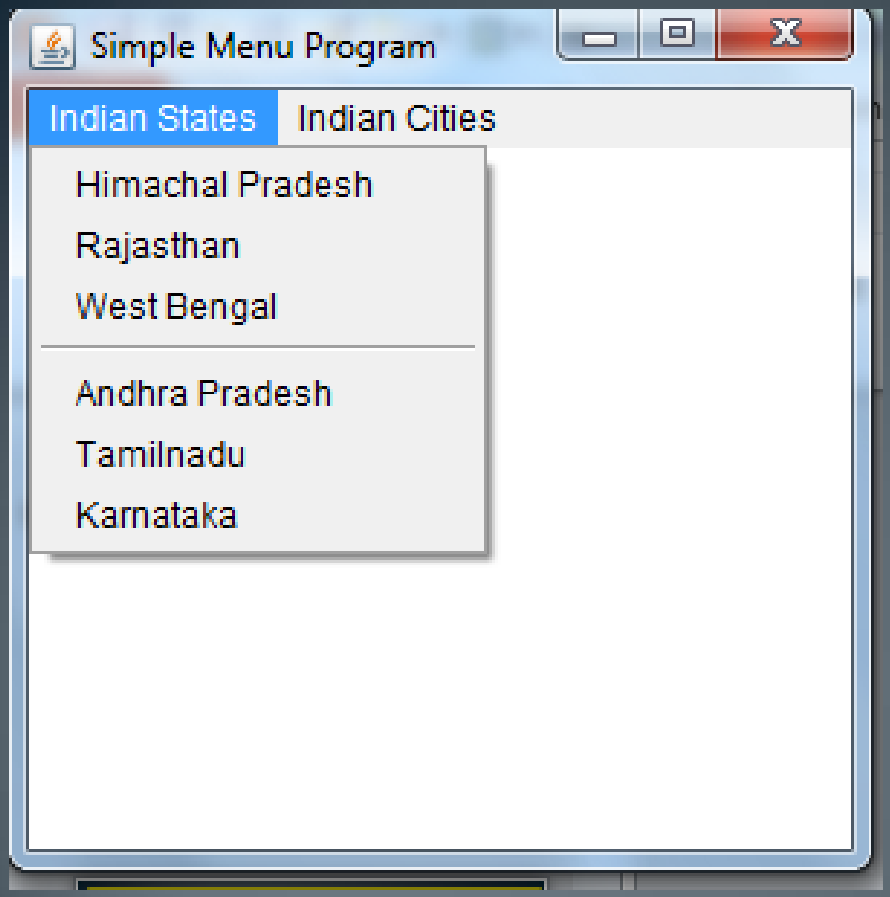
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*   <applet code="JToggleButtonDemo"
width=200 height=80>                </applet> */
public class JToggleButtonDemo extends JApplet {
JLabel jlab;
JToggleButton jtbn;
public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI(); }              }              );
} catch (Exception exc) {
System.out.println("Can't create because of " +
exc);   }      }
private void makeGUI() {
setLayout(new FlowLayout());
// Create a label.
jlab = new JLabel("Button is off.");
// Make a toggle button.
jtbn = new JToggleButton("On/Off");
// Add an item listener for the toggle button.
jtbn.addItemListener(new ItemListener() {
public void itemStateChanged(ItemEvent ie) {
if(jtbn.isSelected())
jlab.setText("Button is on.");
else
jlab.setText("Button is off.");
}
});
// Add the toggle button and label to the content
pane.
add(jtbn);
add(jlab);
}
}
```

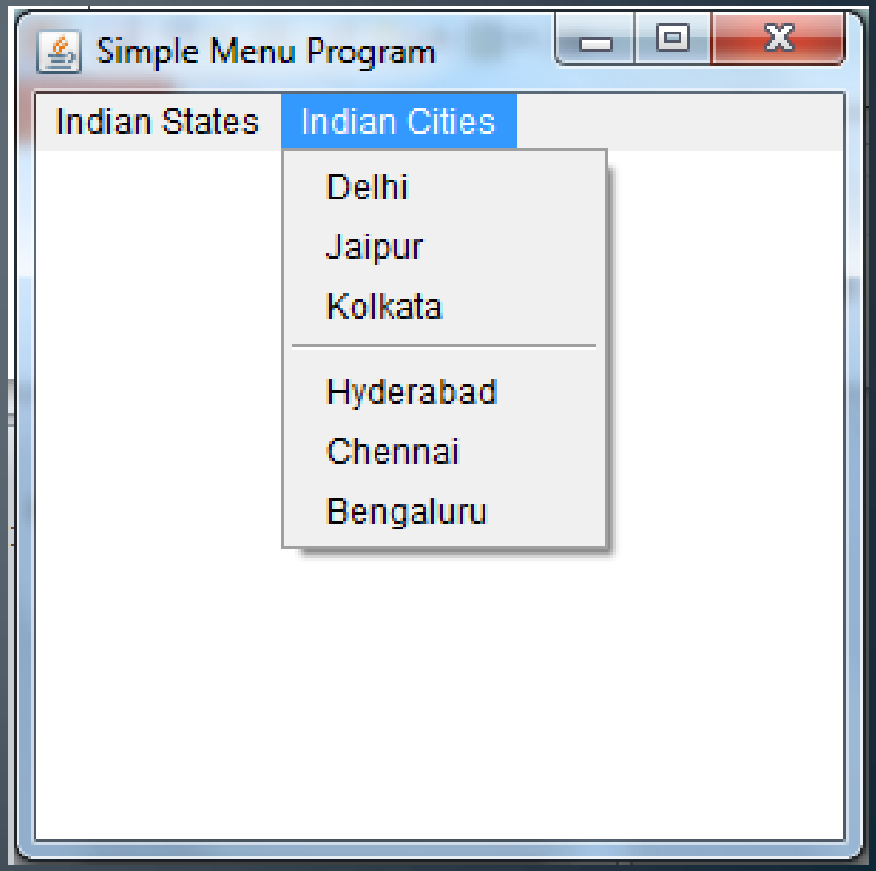

Applet Viewer: JToggleButtonDemo.class
Applet
On/Off
Button is off.
Applet started.


Applet Viewer: JToggleButtonDemo.class
Applet
On/Off
Button is on.
Applet started.


Applet Viewer: JToggleButtonDemo.class
Applet
On/Off
Button is off.
Applet started.

```
import java.awt.*;
import java.awt.event.*;
 public class SimpleMenuExample extends Frame
implements ActionListener   {
  Menu states, cities;
  public SimpleMenuExample()     {
MenuBar mb = new MenuBar();
// begin with creating menu bar
    setMenuBar(mb);
                // add menu bar to frame
     states = new Menu("Indian States");
                // create menus
    cities = new Menu("Indian Cities");
     mb.add(states);
                // add menus to menu bar
    mb.add(cities);
     states.addActionListener(this);
cities.addActionListener(this);
   states.add(new MenuItem("Himachal Pradesh"));
    states.add(new MenuItem("Rajasthan"));
    states.add(new MenuItem("West Bengal"));
    states.addSeparator();
    states.add(new MenuItem("Andhra Pradesh"));
 states.add(new MenuItem("Tamilnadu"));
    states.add(new MenuItem("Karnataka"));
     cities.add(new MenuItem("Delhi"));
    cities.add(new MenuItem("Jaipur"));
    cities.add(new MenuItem("Kolkata"));
    cities.addSeparator();
cities.add(new MenuItem("Hyderabad"));
    cities.add(new MenuItem("Chennai"));
    cities.add(new MenuItem("Bengaluru"));
     setTitle("Simple Menu Program");
// frame creation methods
    setSize(300, 300);
    setVisible(true);        }
  public void actionPerformed(ActionEvent e)    {
    String str = e.getActionCommand();
// know the menu item selected by the user
    System.out.println("You selected " + str);      }
  public static void main(String args[])
  {
    new SimpleMenuExample();
  }
}
```

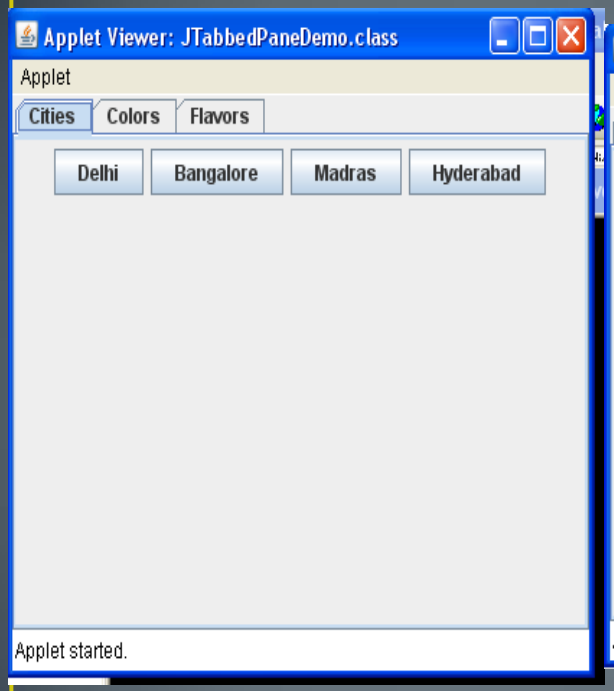

Simple Menu Program
Indian States
Indian Cities
Himachal Pradesh
Rajasthan
West Bengal
Andhra Pradesh
Tamilnadu
Karnataka

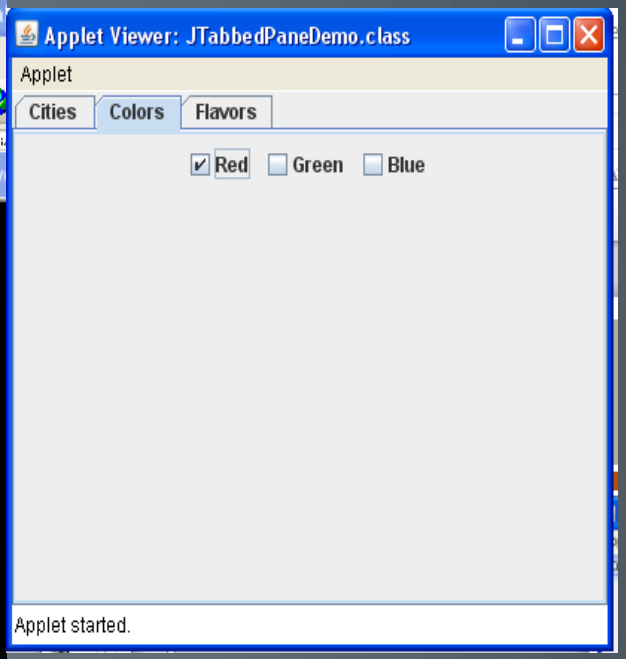

Simple Menu Program
Indian States
Indian Cities
Delhi
Jaipur
Kolkata
Hyderabad
Chennai
Bengaluru

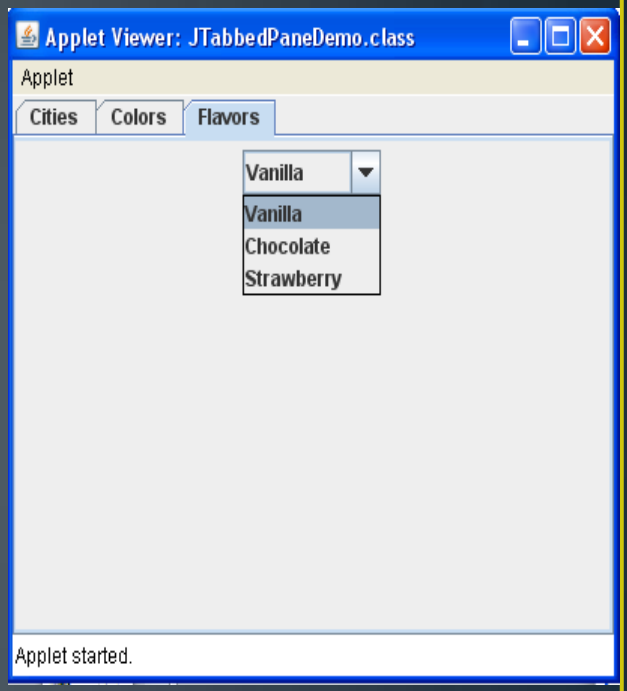
Demonstrate JTabbedPane

```
import javax.swing.*;
/*   <applet code="JTabbedPaneDemo"
width=400 height=100>
                </applet>       */
public class JTabbedPaneDemo extends JApplet {
public void init() {
try {           SwingUtilities.invokeAndWait(
                new Runnable() {
                public void run() {
                makeGUI();   }   }     );
} catch (Exception exc) {
System.out.println("Can't create because of " +
exc);    }      }
private void makeGUI() {
JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Flavors", new FlavorsPanel());
add(jtp);     }       }
// Make the panels that will be added to the
tabbed pane.
class CitiesPanel extends JPanel {
 CitiesPanel() {
 b1 = new JButton("Delhi");
add(b1);
JButton b2 = new JButton("Bangalore");
add(b2);

JButton b3 = new JButton("Madras");
add(b3);
JButton b4 = new JButton("Hyderabad");
add(b4);    }              }
class ColorsPanel extends JPanel {
public ColorsPanel() {
JCheckBox cb1 = new JCheckBox("Red");
add(cb1);
JCheckBox cb2 = new JCheckBox("Green");
add(cb2);
JCheckBox cb3 = new JCheckBox("Blue");
add(cb3);                    }      }
class FlavorsPanel extends JPanel {
public FlavorsPanel() {
JComboBox jcb = new JComboBox();
jcb.addItem("Vanilla");
jcb.addItem("Chocolate");
jcb.addItem("Strawberry");
add(jcb);
}              }
```

Output


Applet Viewer: JTabbedPaneDemo.class
Applet
Cities
Colors
Flavors
Delhi
Bangalore
Madras
Hyderabad
Applet started.


Applet Viewer: JTabbedPaneDemo.class
Applet
Cities
Colors
Flavors
Red
Green
Blue
Applet started.


Applet Viewer: JTabbedPaneDemo.class
Applet
Cities
Colors
Flavors
Vanilla
Vanilla
Chocolate
Strawberry
Applet started.


Success
is dependent on effort.
All the Best