SWING - QUICK GUIDE

SWING - OVERVIEW

Swing API is set of extensible GUI Components to ease developer's life to create JAVA based Front End/ GUI Applications. It is build upon top of AWT API and acts as replacement of AWT API as it has almost every control corresponding to AWT controls. Swing component follows a Model-View-Controller architecture to fulfill the following criterias.

- A single API is to be sufficient to support multiple look and feel.
- API is to model driven so that highest level API is not required to have the data.
- API is to use the Java Bean model so that Builder Tools and IDE can provide better services to the developers to use it.

MVC Architecture

Swing API architecture follows loosely based MVC architecture in the following manner.

- A Model represents component's data.
- View represents visual representation of the component's data.
- Controller takes the input from the user on the view and reflects the changes in Component's data.
- Swing component have Model as a seperate element and View and Controller part are clubbed in User Interface elements. Using this way, Swing has pluggable look-and-feel architecture.

Swing features

- **Light Weight** - Swing component are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- **Rich controls** - Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, table controls
- **Highly Customizable** - Swing controls can be customized in very easy way as visual apperance is independent of internal representation.
- **Pluggable look-and-feel**- SWING based GUI Application look and feel can be changed at run time based on available values.

SWING - ENVIRONMENT

This section guides you on how to download and set up Java on your machine. Please follow the following steps to set up the environment.

Java SE is freely available from the link Download Java. So you download a version based on your operating system.

Follow the instructions to download java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you would need to set environment variables to point to correct installation directories:

Setting up the path for windows 2000/XP:

Assuming you have installed Java in *c:\Program Files\java\jdk* directory:

- Right-click on 'My Computer' and select 'Properties'.

- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting up the path for windows 95/98/ME:

Assuming you have installed Java in *c:\Program Files\java\jdk* directory:

- Edit the 'C:\autoexec.bat' file and add the following line at the end: 'SET PATH=%PATH%;C:\Program Files\java\jdk\bin'

Setting up the path for Linux, UNIX, Solaris, FreeBSD:

Environment variable PATH should be set to point to where the java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

Example, if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH=/path/to/java:$PATH'

Popular Java Editors:

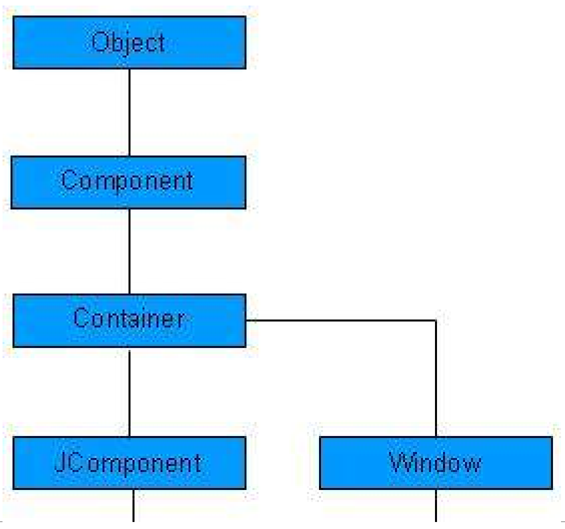
To write your java programs you will need a text editor. There are even more sophisticated IDE available in the market. But for now, you can consider one of the following:

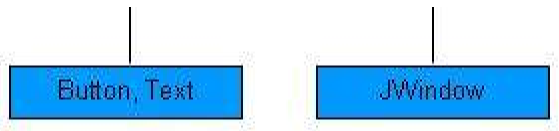
- **Notepad:** On Windows machine you can use any simple text editor like Notepad *Recommendedforthistutorial*, TextPad.
- **Netbeans:**is a Java IDE that is open source and free which can be downloaded from http://www.netbeans.org/index.html.
- **Eclipse:** is also a java IDE developed by the eclipse open source community and can be downloaded from http://www.eclipse.org/.

SWING - CONTROLS

Every user interface considers the following three main aspects:

- **UI elements** : Thes are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex which we will cover in this tutorial.
- **Layouts:** They define how UI elements should be organized on the screen and provide a final look and feel to the GUI *GraphicalUserInterface*. This part will be covered in Layout chapter.
- **Behavior:** These are events which occur when the user interacts with UI elements. This part will be covered in Event Handling chapter.





Every SWING controls inherits properties from following Component class hiearchy.

Sr. No.	Class & Description
1	Component A Container is the abstract base class for the non menu user-interface controls of SWING. Component represents an object with graphical representation
2	Container A Container is a component that can contain other SWING components.
3	JComponent A JComponent is a base class for all swing UI components. In order to use a swing component that inherits from JComponent, component must be in a containment hierarchy whose root is a top-level Swing container.

SWING UI Elements:

Following is the list of commonly used controls while designed GUI using SWING.

Sr. No.	Control & Description
1	JLabel A JLabel object is a component for placing text in a container.
2	JButton This class creates a labeled button.
3	JColorChooser A JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.
4	JCheck Box A JCheckBox is a graphical component that can be in either an **on** *true* or **off** *false* state.
5	JRadioButton The JRadioButton class is a graphical component that can be in either an **on** *true* or **off* *false* state. in a group.
6	JList A JList component presents the user with a scrolling list of text items.

7	JComboBox A JComboBox component presents the user with a to show up menu of choices.
8	JTextField A JTextField object is a text component that allows for the editing of a single line of text.
9	JPasswordField A JPasswordField object is a text component specialized for password entry.
10	JTextArea A JTextArea object is a text component that allows for the editing of a multiple lines of text.
11	ImageIcon A ImageIcon control is an implementation of the Icon interface that paints Icons from Images
12	JScrollbar A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
13	JOptionPane JOptionPane provides set of standard dialog boxes that prompt users for a value or informs them of something.
14	JFileChooser A JFileChooser control represents a dialog window from which the user can select a file.
15	JProgressBar As the task progresses towards completion, the progress bar displays the task's percentage of completion.
16	JSlider A JSlider lets the user graphically select a value by sliding a knob within a bounded interval.
17	JSpinner A JSpinner is a single line input field that lets the user select a number or an object value from an ordered sequence.

SWING - EVENT HANDLING

What is an Event?

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

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

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

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

Steps involved in event handling

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

Points to remember about listener

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

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener will listen to it's events the the source must register itself to the listener.

Event Handling Example

Create the following java program using any editor of your choice in say **D:/ > SWING > com > tutorialspoint > gui >**

SwingControlDemo.java

```java
package com.tutorialspoint.gui;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SwingControlDemo {

   private JFrame mainFrame;
   private JLabel headerLabel;
   private JLabel statusLabel;
   private JPanel controlPanel;

   public SwingControlDemo(){
      prepareGUI();
   }

   public static void main(String[] args){
      SwingControlDemo swingControlDemo = new SwingControlDemo();
      swingControlDemo.showEventDemo();
   }

   private void prepareGUI(){
      mainFrame = new JFrame("Java SWING Examples");
      mainFrame.setSize(400,400);
      mainFrame.setLayout(new GridLayout(3, 1));

      headerLabel = new JLabel("",JLabel.CENTER );
      statusLabel = new JLabel("",JLabel.CENTER);

      statusLabel.setSize(350,100);
      mainFrame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent windowEvent){
         System.exit(0);
         }
      });
      controlPanel = new JPanel();
      controlPanel.setLayout(new FlowLayout());

      mainFrame.add(headerLabel);
      mainFrame.add(controlPanel);
      mainFrame.add(statusLabel);
      mainFrame.setVisible(true);
   }

   private void showEventDemo(){
      headerLabel.setText("Control in action: Button");

      JButton okButton = new JButton("OK");
      JButton submitButton = new JButton("Submit");
      JButton cancelButton = new JButton("Cancel");

      okButton.setActionCommand("OK");
      submitButton.setActionCommand("Submit");
      cancelButton.setActionCommand("Cancel");

      okButton.addActionListener(new ButtonClickListener());
      submitButton.addActionListener(new ButtonClickListener());
      cancelButton.addActionListener(new ButtonClickListener());

      controlPanel.add(okButton);
      controlPanel.add(submitButton);
      controlPanel.add(cancelButton);

      mainFrame.setVisible(true);
   }

   private class ButtonClickListener implements ActionListener{
      public void actionPerformed(ActionEvent e) {
         String command = e.getActionCommand();
         if( command.equals( "OK" ))  {
            statusLabel.setText("Ok Button clicked.");
         }
         else if( command.equals( "Submit" ) )  {
            statusLabel.setText("Submit Button clicked.");
         }
         else  {
            statusLabel.setText("Cancel Button clicked.");
         }
      }
   }
}
```

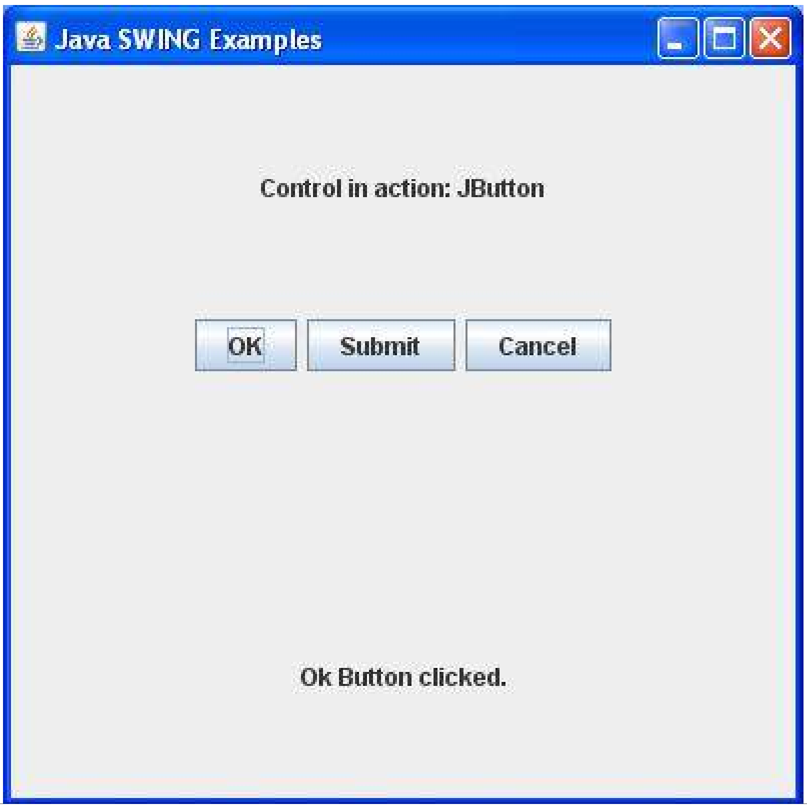
Compile the program using command prompt. Go to **D:/ > SWING** and type the following command.

```
D:\AWT>javac com\tutorialspoint\gui\SwingControlDemo.java
```

If no error comes that means compilation is successful. Run the program using following command.

```
D:\AWT>java com.tutorialspoint.gui.SwingControlDemo
```

Verify the following output



SWING - EVENT CLASSES

The Event classes represent the event. Java provides us various Event classes but we will discuss those which are more frequently used.

EventObject class

It is the root class from which all event state objects shall be derived. All Events are constructed with a reference to the object, the **source**, that is logically deemed to be the object upon which the Event in question initially occurred upon. This class is defined in java.util package.

Class declaration

Following is the declaration for **java.util.EventObject** class:

```
public class EventObject
   extends Object
      implements Serializable
```

Field

Following are the fields for **java.util.EventObject** class:

- **protected Object source** -- The object on which the Event initially occurred.

Class constructors

S.N.	Constructor & Description
1	**EventObject***Objectsource* Constructs a prototypical Event.

Class methods

S.N.	Method & Description
1	**Object getSource** The object on which the Event initially occurred.
2	**String toString** Returns a String representation of this EventObject.

Methods inherited

This class inherits methods from the following classes:

- java.lang.Object

SWING Event Classes:

Following is the list of commonly used event classes.

Sr.	Control & Description

No.	
1	AWTEvent It is the root event class for all SWING events. This class and its subclasses supercede the original java.awt.Event class.
2	ActionEvent The ActionEvent is generated when button is clicked or the item of a list is double clicked.
3	InputEvent The InputEvent class is root event class for all component-level input events.
4	KeyEvent On entering the character the Key event is generated.
5	MouseEvent This event indicates a mouse action occurred in a component.
6	WindowEvent The object of this class represents the change in state of a window.
7	AdjustmentEvent The object of this class represents the adjustment event emitted by Adjustable objects.
8	ComponentEvent The object of this class represents the change in state of a window.
9	ContainerEvent The object of this class represents the change in state of a window.
10	MouseMotionEvent The object of this class represents the change in state of a window.
11	PaintEvent The object of this class represents the change in state of a window.

SWING - EVENT LISTENERS

The Event listener represent the interfaces responsible to handle events. Java provides us various Event listener classes but we will discuss those which are more frequently used. Every method of an event listener method has a single argument as an object which is subclass of EventObject class. For example, mouse event listener methods will accept instance of MouseEvent, where MouseEvent derives from EventObject.

EventListner interface

It is a marker interface which every listener interface has to extend.This class is defined in java.util package.

Class declaration

Following is the declaration for **java.util.EventListener** interface:

```
public interface EventListener
```

SWING Event Listener Interfaces:

Following is the list of commonly used event listeners.

td>ActionListener

This interface is used for receiving the action events.

Sr. No.	
2	ComponentListener This interface is used for receiving the component events.
3	ItemListener This interface is used for receiving the item events.
4	KeyListener This interface is used for receiving the key events.
5	MouseListener This interface is used for receiving the mouse events.
6	WindowListener This interface is used for receiving the window events.
7	AdjustmentListener This interface is used for receiving the adjusmtent events.
8	ContainerListener This interface is used for receiving the container events.
9	MouseMotionListener This interface is used for receiving the mouse motion events.
10	FocusListener This interface is used for receiving the focus events.

SWING - EVENT ADAPTERS

Adapters are abstract classes for receiving various events. The methods in these classes are empty. These classes exists as convenience for creating listener objects.

SWING Adapters:

Following is the list of commonly used adapters while listening GUI events in SWING.

Sr. No.	Adapter & Description
1	FocusAdapter An abstract adapter class for receiving focus events.
2	KeyAdapter An abstract adapter class for receiving key events.
3	MouseAdapter An abstract adapter class for receiving mouse events.
4	MouseMotionAdapter An abstract adapter class for receiving mouse motion events.
5	WindowAdapter An abstract adapter class for receiving window events.

SWING - LAYOUTS

Introduction

Layout means the arrangement of components within the container. In other way we can say that placing the components at a particular position within the container. The task of layouting the controls is done automatically by the Layout Manager.

Layout Manager

The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager. It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

- It is very tedious to handle a large number of controls within the container.
- Oftenly the width and height information of a component is not given when we need to arrange them.

Java provide us with various layout manager to position the controls. The properties like size,shape and arrangement varies from one layout manager to other layout manager. When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the LayoutManager interface.

Following are the interfaces defining functionalities of Layout Managers.

Sr. No.	Interface & Description
1	LayoutManager The LayoutManager interface declares those methods which need to be implemented by the class whose object will act as a layout manager.
2	LayoutManager2 The LayoutManager2 is the sub-interface of the LayoutManager. This interface is for those classes that know how to layout containers based on layout constraint object.

AWT Layout Manager Classes:

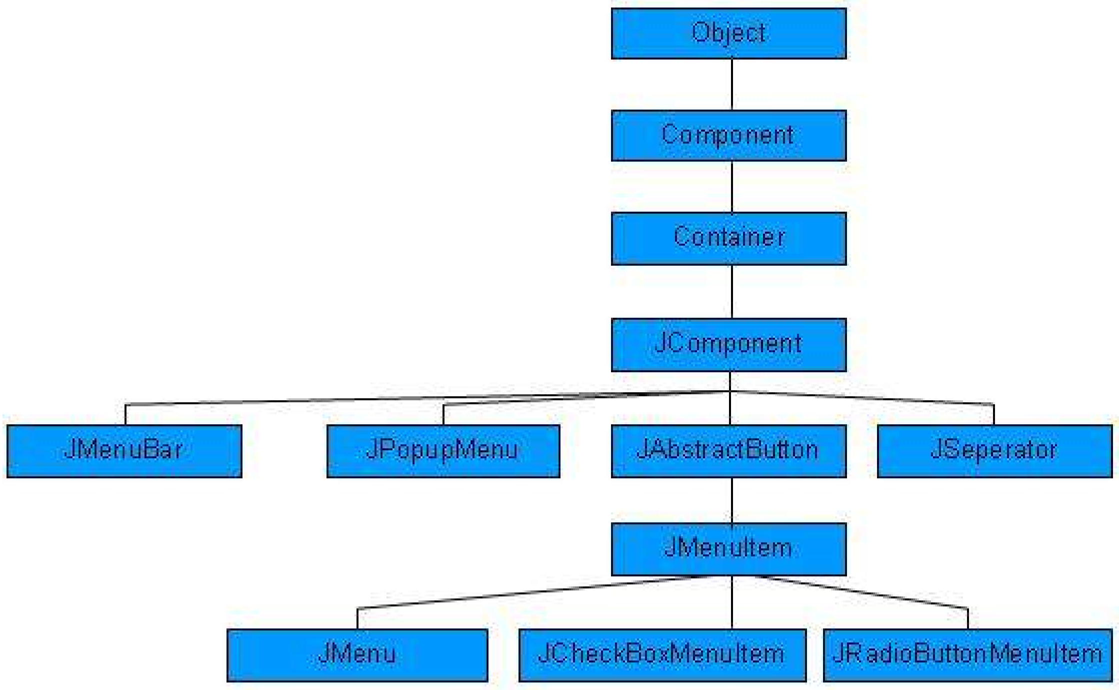
Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	LayoutManager & Description
1	BorderLayout The borderlayout arranges the components to fit in the five regions: east, west, north, south and center.
2	CardLayout The CardLayout object treats each component in the container as a card. Only one card is visible at a time.
3	FlowLayout The FlowLayout is the default layout.It layouts the components in a directional flow.
4	GridLayout The GridLayout manages the components in form of a rectangular grid.
5	GridBagLayout This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically,horizontally or along their baseline without requiring the components of same size.
6	GroupLayout The GroupLayout hierarchically groups components in order to position them in a Container.
7	SpringLayout A SpringLayout positions the children of its associated container according to a set of constraints.

SWING - MENU

As we know that every top-level window has a menu bar associated with it. This menu bar consist of various menu choices available to the end user. Further each choice contains list of options which is called drop down menus. Menu and MenuItem controls are subclass of MenuComponent class.

Menu Hiearchy



Menu Controls

Sr. No.	Control & Description
1	JMenuBar The JMenuBar object is associated with the top-level window.
2	JMenuItem The items in the menu must belong to the JMenuItem or any of its subclass.
3	JMenu The JMenu object is a pull-down menu component which is displayed from the menu bar.
4	JCheckboxMenuItem JCheckboxMenuItem is subclass of JMenuItem.
5	JRadioButtonMenuItem JRadioButtonMenuItem is subclass of JMenuItem.

6	JPopupMenu JPopupMenu can be dynamically popped up at a specified position within a component.

SWING - CONTAINERS

Introduction

The class **Container** is the super class for the containers of AWT. Container object can contain other AWT components.

Class declaration

Following is the declaration for **java.awt.Container** class:

```
public class Container
   extends Component
```

Class constructors

S.N.	Constructor & Description
1	**Container** This creates a new Container.

Class methods

S.N.	Method & Description
1	**Component add***Componentcomp* Appends the specified component to the end of this container.
2	**Component add***Componentcomp, intindex* Adds the specified component to this container at the given position.
3	**void add***Componentcomp, Objectconstraints* Adds the specified component to the end of this container.
4	**void add***Componentcomp, Objectconstraints, intindex* Adds the specified component to this container with the specified constraints at the specified index.
5	**Component add***Stringname, Componentcomp* Adds the specified component to this container.
6	**void addContainerListener***ContainerListenerl*

Adds the specified container listener to receive container events from this container.

7	**protected void addImpl***Componentcomp, Objectconstraints, intindex*
	Adds the specified component to this container at the specified index.
8	**void addNotify**
	Makes this Container displayable by connecting it to a native screen resource.
9	**void addPropertyChangeListener***PropertyChangeListenerlistener*
	Adds a PropertyChangeListener to the listener list.
10	**void addPropertyChangeListener***StringpropertyName, PropertyChangeListenerlistener*
	Adds a PropertyChangeListener to the listener list for a specific property.
11	**void applyComponentOrientation***ComponentOrientationo*
	Sets the ComponentOrientation property of this container and all components contained within it.
12	**boolean areFocusTraversalKeysSet***intid*
	Returns whether the Set of focus traversal keys for the given focus traversal operation has been explicitly defined for this Container.
13	**int countComponents**
	Deprecated. As of JDK version 1.1, replaced by getComponentCount.
14	**void deliverEvent***Evente*
	Deprecated. As of JDK version 1.1, replaced by dispatchEvent*AWTEvente*
15	**void doLayout**
	Causes this container to lay out its components.
16	**Component findComponentAt***intx, inty*
	Locates the visible child component that contains the specified position.
17	**Component findComponentAt***Pointp*
	Locates the visible child component that contains the specified point.
18	**float getAlignmentX**
	Returns the alignment along the x axis.
19	**float getAlignmentY**
	Returns the alignment along the y axis.

20	**Component getComponent*intn*** Gets the nth component in this container.
21	**Component getComponentAt*intx, inty*** Locates the component that contains the x,y position.
22	**Component getComponentAt*Pointp*** Gets the component that contains the specified point.
23	**int getComponentCount** Gets the number of components in this panel.
24	**Component[] getComponents** Gets all the components in this container.
25	**int getComponentZOrder*Componentcomp*** Returns the z-order index of the component inside the container.
26	**ContainerListener[] getContainerListeners** Returns an array of all the container listeners registered on this container.
27	**Set<AWTKeyStroke> getFocusTraversalKeys*intid*** Returns the Set of focus traversal keys for a given traversal operation for this Container.
28	**FocusTraversalPolicy getFocusTraversalPolicy** Returns the focus traversal policy that will manage keyboard traversal of this Container's children, or null if this Container is not a focus cycle root.
29	**Insets getInsets** Determines the insets of this container, which indicate the size of the container's border.
30	**LayoutManager getLayout** Gets the layout manager for this container.
31	**<T extends EventListener> T[] getListeners*Class < T > listenerType*** Returns an array of all the objects currently registered as FooListeners upon this Container.
32	**Dimension getMaximumSize** Returns the maximum size of this container.
33	**Dimension getMinimumSize** Returns the minimum size of this container.

34 **Point getMousePosition*booleanallowChildren***

Returns the position of the mouse pointer in this Container's coordinate space if the Container is under the mouse pointer, otherwise returns null.

35 **Dimension getPreferredSize**

Returns the preferred size of this container.

36 **Insets insets**

Deprecated. As of JDK version 1.1, replaced by getInsets.

37 **void invalidate**

Invalidates the container.

38 **boolean isAncestorOf*Componentc***

Checks if the component is contained in the component hierarchy of this container.

39 **boolean isFocusCycleRoot**

Returns whether this Container is the root of a focus traversal cycle.

40 **boolean isFocusCycleRoot*Containercontainer***

Returns whether the specified Container is the focus cycle root of this Container's focus traversal cycle.

41 **boolean isFocusTraversalPolicyProvider**

Returns whether this container provides focus traversal policy.

42 **boolean isFocusTraversalPolicySet**

Returns whether the focus traversal policy has been explicitly set for this Container.

43 **void layout**

Deprecated. As of JDK version 1.1, replaced by doLayout.

44 **void list*PrintStreamout, intindent***

Prints a listing of this container to the specified output stream.

45 **void list*PrintWriterout, intindent***

Prints out a list, starting at the specified indentation, to the specified print writer.

46 **Component locate*intx, inty***

Deprecated. As of JDK version 1.1, replaced by getComponentAt*int, int*.

47 **Dimension minimumSize**

Deprecated. As of JDK version 1.1, replaced by getMinimumSize.

48 **void paint***Graphicsg*

Paints the container.

49 **void paintComponents***Graphicsg*

Paints each of the components in this container.

50 **protected String paramString**

Returns a string representing the state of this Container.

51 **Dimension preferredSize**

Deprecated. As of JDK version 1.1, replaced by getPreferredSize.

52 **void print***Graphicsg*

Prints the container.

53 **void printComponents***Graphicsg*

Prints each of the components in this container.

54 **protected void processContainerEvent***ContainerEvente*

Processes container events occurring on this container by dispatching them to any registered ContainerListener objects.

55 **protected void processEvent***AWTEvente*

Processes events on this container.

56 **void remove***Componentcomp*

Removes the specified component from this container.

57 **void remove***intindex*

Removes the component, specified by index, from this container.

58 **void removeAll**

Removes all the components from this container.

59 **void removeContainerListener***ContainerListenerl*

Removes the specified container listener so it no longer receives container events from this container.

60 **void removeNotify**

Makes this Container undisplayable by removing its connection to its native screen resource.

61 **void setComponentZOrder***Componentcomp, intindex*

Moves the specified component to the specified z-order index in the container.

62 **void setFocusCycleRoot***booleanfocusCycleRoot*

Sets whether this Container is the root of a focus traversal cycle.

63 **void setFocusTraversalKeys***intid, Set < ?extendsAWTKeyStroke > keystrokes*

Sets the focus traversal keys for a given traversal operation for this Container.

64 **void setFocusTraversalPolicy***FocusTraversalPolicypolicy*

Sets the focus traversal policy that will manage keyboard traversal of this Container's children, if this Container is a focus cycle root.

65 **void setFocusTraversalPolicyProvider***booleanprovider*

Sets whether this container will be used to provide focus traversal policy.

66 **void setFont***Fontf*

Sets the font of this container.

67 **void setLayout***LayoutManagermgr*

Sets the layout manager for this container.

68 **void transferFocusBackward**

Transfers the focus to the previous component, as though this Component were the focus owner.

69 **void transferFocusDownCycle**

Transfers the focus down one focus traversal cycle.

70 **void update***Graphicsg*

Updates the container.

71 **void validate**

Validates this container and all of its subcomponents.

72 **protected void validateTree**

Recursively descends the container tree and recomputes the layout for any subtrees marked as needing it *thosemarkedasinvalid*.

Methods inherited

This class inherits methods from the following classes:

- java.awt.Component

- java.lang.Object

Loading [MathJax]/jax/output/HTML-CSS/jax.js