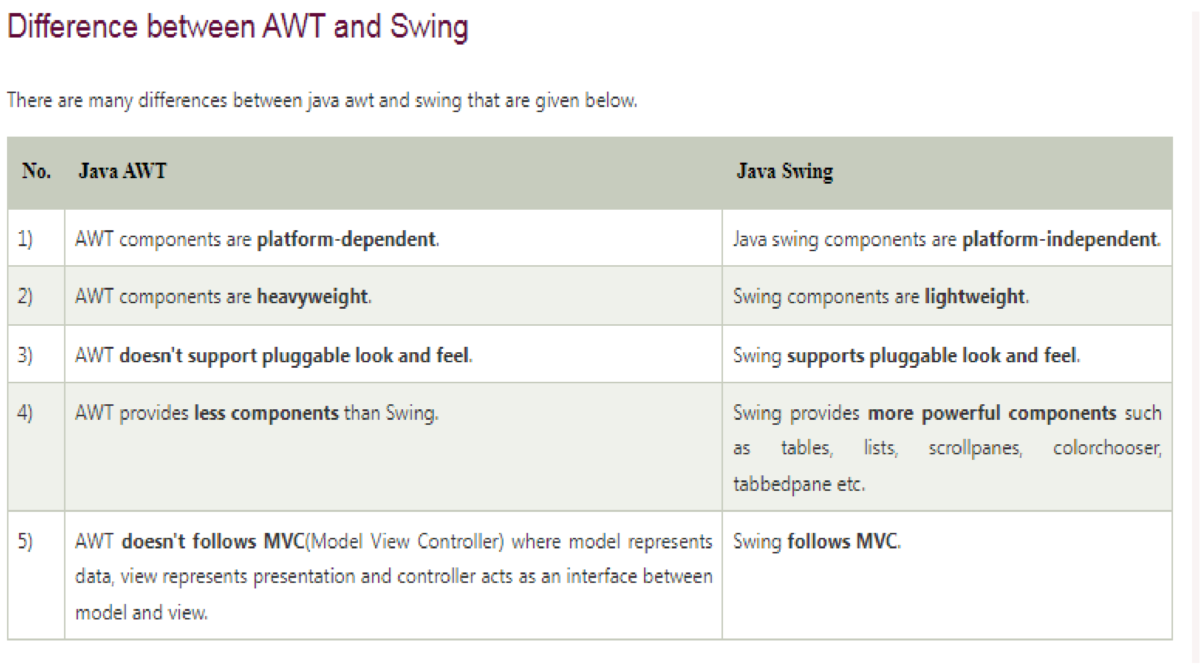
DR SNS RAJALAKSHI COLLEGE OF ARTS AND SCIENCE(AUTONOMOS), COIMBATORE -641017

16UCA502 - ADVANCED JAVA PROGRAMMING

UNIT I

Introducing Swing: swing is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.Unlike AWT, Java Swing provides platform-independent and lightweight components.The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.



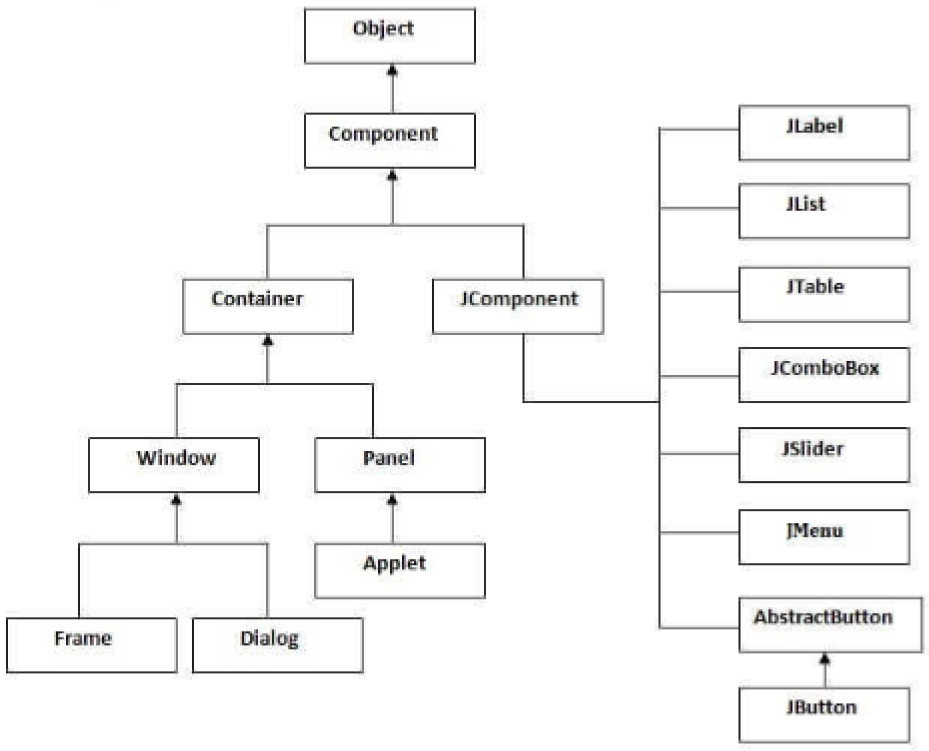
Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Commonly used Methods of Component class

The methods of Component class are widely used in java swing that are given below.

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Display different shapes

Demonstrates how to draw a line using draw() method of Graphics2D class with Line2D object as an argument.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.Line2D;
import javax.swing.JApplet;
import javax.swing.JFrame;

public class Main extends JApplet {
  public void init() {
    setBackground(Color.white);
    setForeground(Color.white);
  }
  public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setPaint(Color.gray);
    int x = 5;
    int y = 7;
    g2.draw(new Line2D.Double(x, y, 200, 200));
    g2.drawString("Line", x, 250);
  }
  public static void main(String s[]) {
    JFrame f = new JFrame("Line");
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    JApplet applet = new Main();
    f.getContentPane().add("Center", applet);
    applet.init();

    f.pack();
    f.setSize(new Dimension(300, 300));
    f.setVisible(true);
  }
}
```

Handling Events

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener

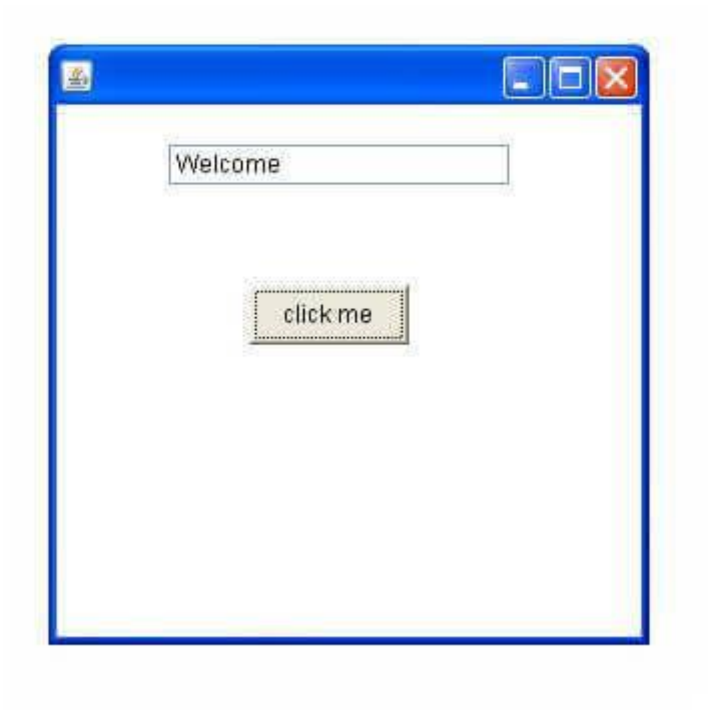
Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

```java
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){

//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);

//register listener
b.addActionListener(this);//passing current instance

//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}
}
```

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.



Delegation Event Model in Java

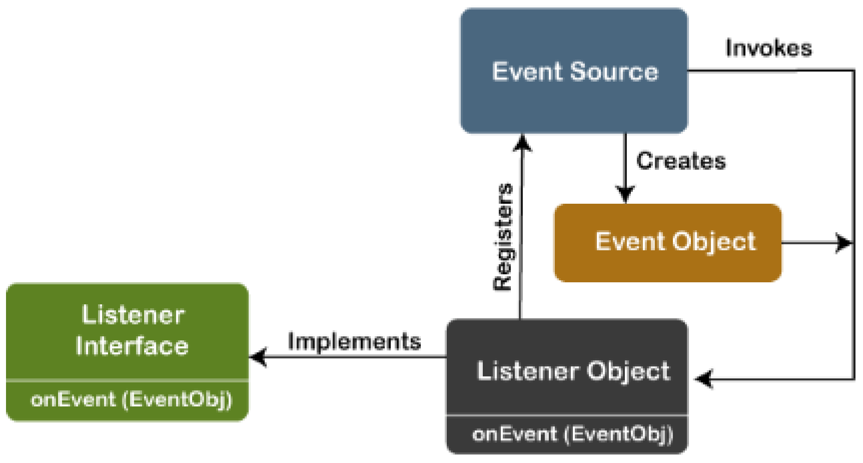
The Delegation Event model is defined to handle events in GUI programming languages. The GUI stands for Graphical User Interface, where a user graphically/visually interacts with the system.

The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

In this section, we will discuss event processing and how to implement the delegation event model in Java. We will also discuss the different components of an Event Model.

Event Processing in Java

Java support event processing since Java 1.0. It provides support for AWT (Abstract Window Toolkit), which is an API used to develop the Desktop application. In Java 1.0, the AWT was based on inheritance. To catch and process GUI events for a program, it should hold subclass GUI components and override action() or handleEvent() methods. The below image demonstrates the event processing.



But, the modern approach for event processing is based on the Delegation Model. It defines a standard and compatible mechanism to generate and process events. In this model, a source generates an event and forwards it to one or more listeners. The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it. The UI elements are able to delegate the processing of an event to a separate function.

The key advantage of the Delegation Event Model is that the application logic is completely separated from the interface logic.

In this model, the listener must be connected with a source to receive the event notifications. Thus, the events will only be received by the listeners who wish to receive them. So, this approach is more convenient than the inheritance-based event model (in Java 1.0).

In the older model, an event was propagated up the containment until a component was handled. This needed components to receive events that were not processed, and it took lots of time. The Delegation Event model overcame this issue.

Basically, an Event Model is based on the following three components:

- Events
- Events Sources
- Events Listeners

Events

The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. We can also consider many other user operations as events.

The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

Event Sources

A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:

```
public void addTypeListener (TypeListener e1)
```

From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener. For example, for a keyboard event listener, the method will be called as **addKeyListener()**. For the mouse event listener, the method will be called as **addMouseMotionListener()**. When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object. This process is known as event multicasting. In few cases, the event notification will only be sent to listeners that register to receive them.

Some listeners allow only one listener to register. Below is an example:

1. **public void** addTypeListener(TypeListener e2) **throws** java.util.TooManyListenersE xception

From the above syntax, the Type is the name of the event, and e2 is the event listener's reference. When the specified event occurs, it will be notified to the registered listener. This process is known as **unicasting** events.

A source should contain a method that unregisters a specific type of event from the listener if not needed. Below is an example of the method that will remove the event from the listener.

1. **public void** removeTypeListener(TypeListener e2?)

From the above syntax, the Type is an event name, and e2 is the reference of the listener. For example, to remove the keyboard listener, the **removeKeyListener()** method will be called.

The source provides the methods to add or remove listeners that generate the events. For example, the Component class contains the methods to operate on the different types of events, such as adding or removing them from the listener.

Event Listeners

An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

For example, the **MouseMotionListener** interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

Types of Events

The events are categories into the following two categories:

The Foreground Events:

The foreground events are those events that require direct interaction of the user. These types of events are generated as a result of user interaction with the GUI component. For example, clicking on a button, mouse movement, pressing a keyboard key, selecting an option from the list, etc.

The Background Events :

The Background events are those events that result from the interaction of the end-user. For example, an Operating system interrupts system failure (Hardware or Software).

To handle these events, we need an event handling mechanism that provides control over the events and responses.

The Delegation Model

The Delegation Model is available in Java since Java 1.1. it provides a new delegation-based event model using AWT to resolve the event problems. It provides a convenient mechanism to support complex Java programs.

Design Goals

The design goals of the event delegation model are as following:

- It is easy to learn and implement
- It supports a clean separation between application and GUI code.
- It provides robust event handling program code which is less error-prone (strong compile-time checking)
- It is Flexible, can enable different types of application models for event flow and propagation.
- It enables run-time discovery of both the component-generated events as well as observable events.
- It provides support for the backward binary compatibility with the previous model.

Let's implement it with an example:

Java Program to Implement the Event Delegation Model

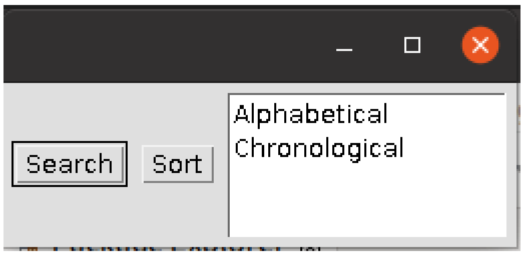
The below is a Java program to handle events implementing the event deligation model:

TestApp.java:

```java
import java.awt.*;
import java.awt.event.*;

public class TestApp {
   public void search() {
      // For searching
      System.out.println("Searching...");
   }
   public void sort() {
      // for sorting
      System.out.println("Sorting....");
   }

   static public void main(String args[]) {
     TestApp app = new TestApp();
     GUI gui = new GUI(app);
   }
}

class Command implements ActionListener  {
   static final int SEARCH = 0;
   static final int SORT = 1;
   int id;
   TestApp app;

   public Command(int id, TestApp app) {
      this.id = id;
      this.app = app;
   }

   public void actionPerformed(ActionEvent e) {
      switch(id) {
        case SEARCH:
          app.search();
          break;
        case SORT:
          app.sort();
          break;
      }
   }
```

```java
}

class GUI {

   public GUI(TestApp app) {
      Frame f = new Frame();
      f.setLayout(new FlowLayout());

      Command searchCmd = new Command(Command.SEARCH, app);
      Command sortCmd = new Command(Command.SORT, app);

      Button b;
      f.add(b = new Button("Search"));
      b.addActionListener(searchCmd);
      f.add(b = new Button("Sort"));
      b.addActionListener(sortCmd);

      List l;
      f.add(l = new List());
      l.add("Alphabetical");
      l.add("Chronological");
      l.addActionListener(sortCmd);
      f.pack();

      f.show();
   }
}
```

Output:



java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface

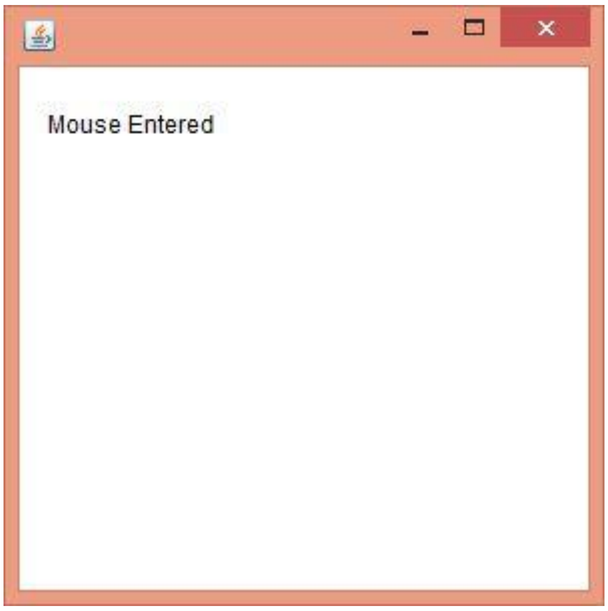
The signature of 5 methods found in MouseListener interface are given below:

```
public abstract void mouseClicked(MouseEvent e);
public abstract void mouseEntered(MouseEvent e);
public abstract void mouseExited(MouseEvent e);
public abstract void mousePressed(MouseEvent e);
public abstract void mouseReleased(MouseEvent e);
```

Java MouseListener Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener{
    Label l;
    MouseListenerExample(){
        addMouseListener(this);

        l=new Label();
        l.setBounds(20,50,100,20);
        add(l);
        setSize(300,300);
        setLayout(null);
        setVisible(true);
    }
    public void mouseClicked(MouseEvent e) {
        l.setText("Mouse Clicked");
    }
    public void mouseEntered(MouseEvent e) {
        l.setText("Mouse Entered");
    }
    public void mouseExited(MouseEvent e) {
        l.setText("Mouse Exited");
    }
    public void mousePressed(MouseEvent e) {
        l.setText("Mouse Pressed");
    }
    public void mouseReleased(MouseEvent e) {
        l.setText("Mouse Released");
    }
public static void main(String[] args) {
    new MouseListenerExample();
}
}
```

Output:


```
Mouse Entered
```

Java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. The **Java LayoutManagers** facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are the following classes that represent the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

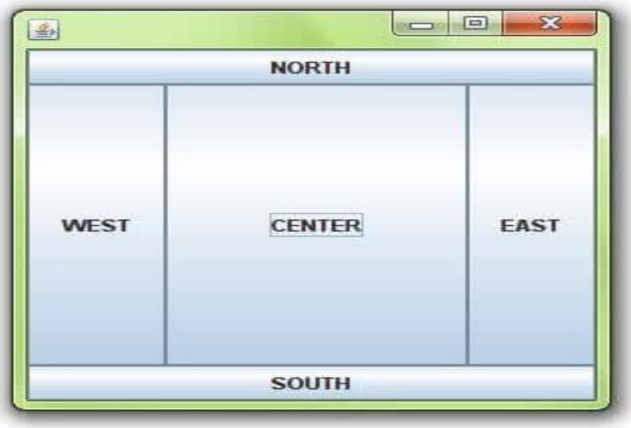
Example of BorderLayout class: Using BorderLayout() constructor

FileName: Border.java

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

public class Border
{
JFrame f;
Border()
{
    f = new JFrame();

```

```
     // creating buttons
    JButton b1 = new JButton("NORTH");; // the button will be labeled as NORTH
    JButton b2 = new JButton("SOUTH");; // the button will be labeled as SOUTH
    JButton b3 = new JButton("EAST");; // the button will be labeled as EAST
    JButton b4 = new JButton("WEST");; // the button will be labeled as WEST
    JButton b5 = new JButton("CENTER");; // the button will be labeled as CENTER

    f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction
    f.add(b2, BorderLayout.SOUTH);  // b2 will be placed in the South Direction
    f.add(b3, BorderLayout.EAST);  // b2 will be placed in the East Direction
    f.add(b4, BorderLayout.WEST);  // b2 will be placed in the West Direction
    f.add(b5, BorderLayout.CENTER);  // b2 will be placed in the Center

    f.setSize(300, 300);
    f.setVisible(true);
}
public static void main(String[] args) {
    new Border();
}
}
```

Output:



Example of BorderLayout class: Using BorderLayout(int hgap, int vgap) constructor

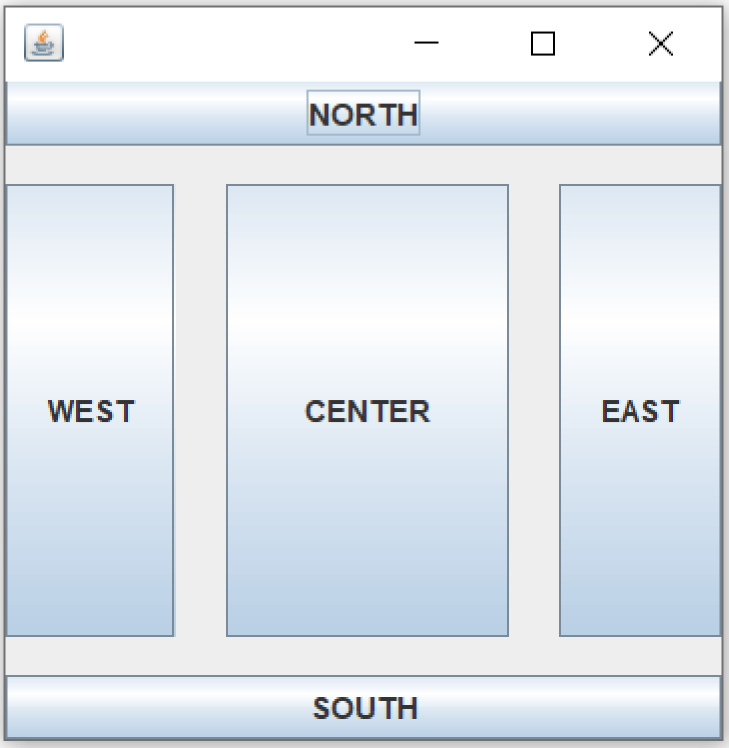
The following example inserts horizontal and vertical gaps between buttons using the parameterized constructor BorderLayout(int hgap, int gap)

FileName: BorderLayoutExample.java

```java
// import statement
import java.awt.*;
import javax.swing.*;
public class BorderLayoutExample
{
JFrame jframe;
// constructor
BorderLayoutExample()
{
   // creating a Frame
   jframe = new JFrame();
   // create buttons
   JButton btn1 = new JButton("NORTH");
   JButton btn2 = new JButton("SOUTH");
   JButton btn3 = new JButton("EAST");
   JButton btn4 = new JButton("WEST");
   JButton btn5 = new JButton("CENTER");
    // creating an object of the BorderLayout class using
    // the parameterized constructor where the horizontal gap is 20
    // and vertical gap is 15. The gap will be evident when buttons are placed
    // in the frame
   jframe.setLayout(new BorderLayout(20, 15));
   jframe.add(btn1, BorderLayout.NORTH);
   jframe.add(btn2, BorderLayout.SOUTH);
   jframe.add(btn3, BorderLayout.EAST);
```

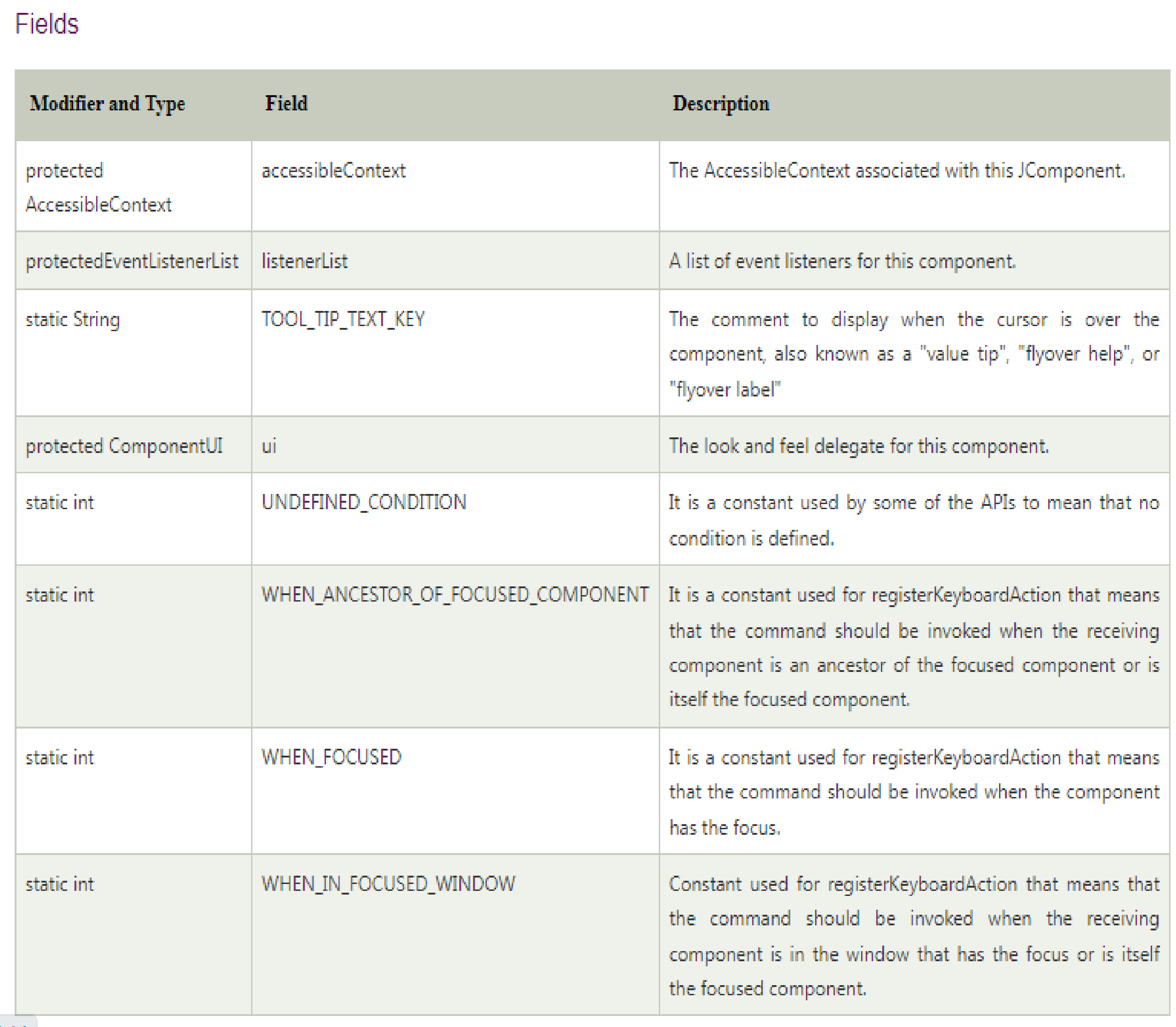
```
    jframe.add(btn4, BorderLayout.WEST);
    jframe.add(btn5, BorderLayout.CENTER);
    jframe.setSize(300,300);
    jframe.setVisible(true);
}
// main method
public static void main(String argvs[])
{
    new BorderLayoutExample();
}
}
```

Output:



Java JComponent

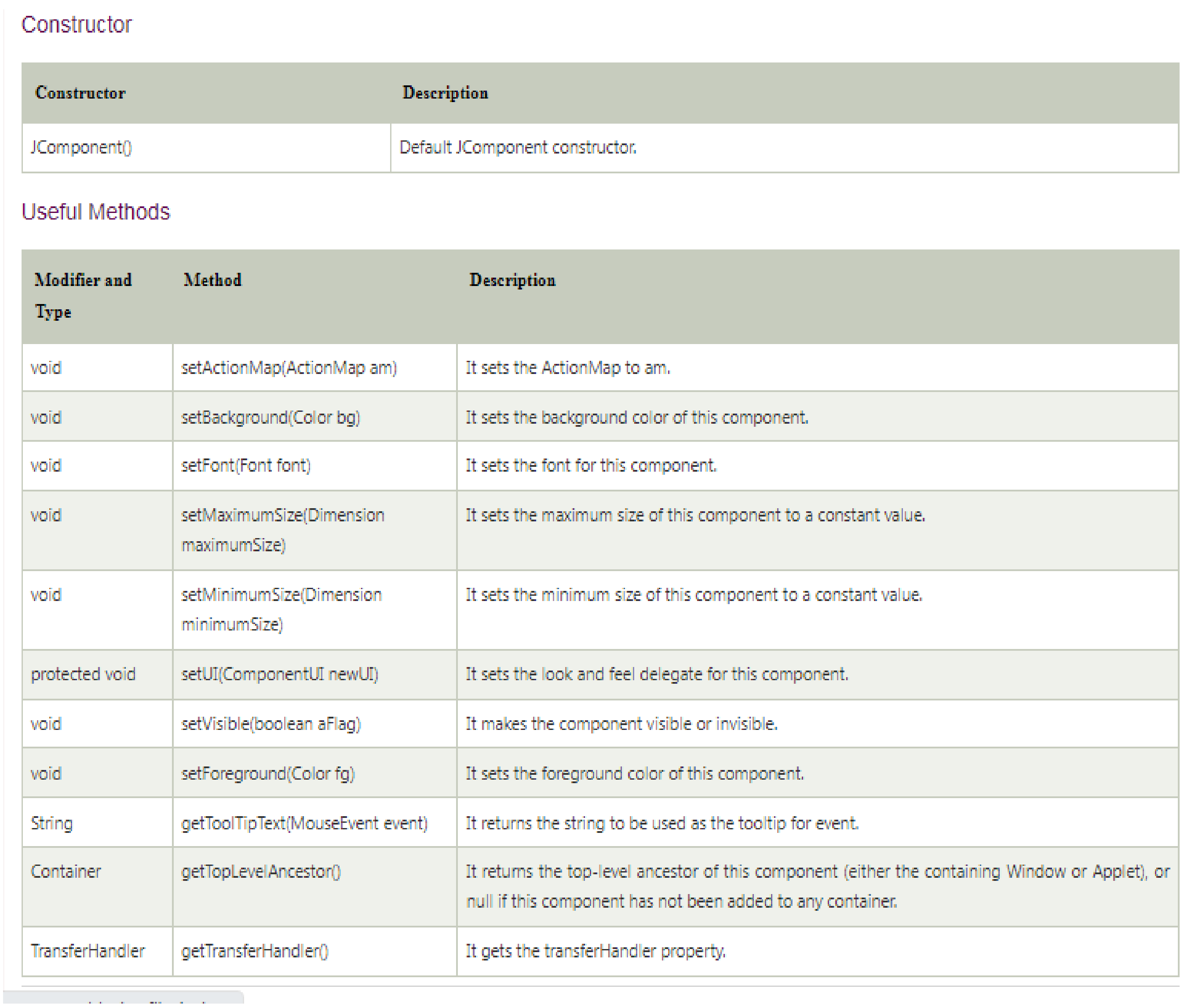
The JComponent class is the base class of all Swing components except top-level containers. Swing components whose names begin with "J" are descendants of the JComponent class. For example, JButton, JScrollPane, JPanel, JTable etc. But, JFrame and JDialog don't inherit JComponent class because they are the child of top-level containers.



Fields

Modifier and Type	Field	Description
protected AccessibleContext	accessibleContext	The AccessibleContext associated with this JComponent.
protectedEventListenerList	listenerList	A list of event listeners for this component.
static String	TOOL_TIP_TEXT_KEY	The comment to display when the cursor is over the component, also known as a "value tip", "flyover help", or "flyover label"
protected ComponentUI	ui	The look and feel delegate for this component.
static int	UNDEFINED_CONDITION	It is a constant used by some of the APIs to mean that no condition is defined.
static int	WHEN_ANCESTOR_OF_FOCUSED_COMPONENT	It is a constant used for registerKeyboardAction that means that the command should be invoked when the receiving component is an ancestor of the focused component or is itself the focused component.
static int	WHEN_FOCUSED	It is a constant used for registerKeyboardAction that means that the command should be invoked when the component has the focus.
static int	WHEN_IN_FOCUSED_WINDOW	Constant used for registerKeyboardAction that means that the command should be invoked when the receiving component is in the window that has the focus or is itself the focused component.

The JComponent class extends the Container class which itself extends Component. The Container class has support for adding components to the container.



Constructor

Constructor	Description
JComponent()	Default JComponent constructor.

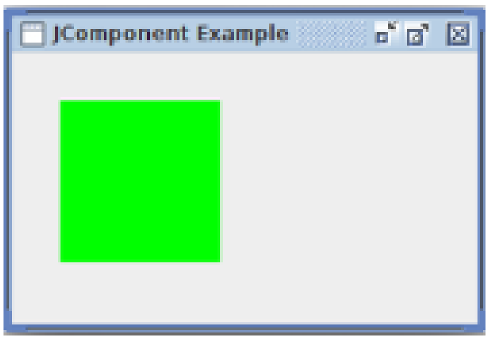
Useful Methods

Modifier and Type	Method	Description
void	setActionMap(ActionMap am)	It sets the ActionMap to am.
void	setBackground(Color bg)	It sets the background color of this component.
void	setFont(Font font)	It sets the font for this component.
void	setMaximumSize(Dimension maximumSize)	It sets the maximum size of this component to a constant value.
void	setMinimumSize(Dimension minimumSize)	It sets the minimum size of this component to a constant value.
protected void	setUI(ComponentUI newUI)	It sets the look and feel delegate for this component.
void	setVisible(boolean aFlag)	It makes the component visible or invisible.
void	setForeground(Color fg)	It sets the foreground color of this component.
String	getToolTipText(MouseEvent event)	It returns the string to be used as the tooltip for event.
Container	getTopLevelAncestor()	It returns the top-level ancestor of this component (either the containing Window or Applet), or null if this component has not been added to any container.
TransferHandler	getTransferHandler()	It gets the transferHandler property.

Java JComponent Example

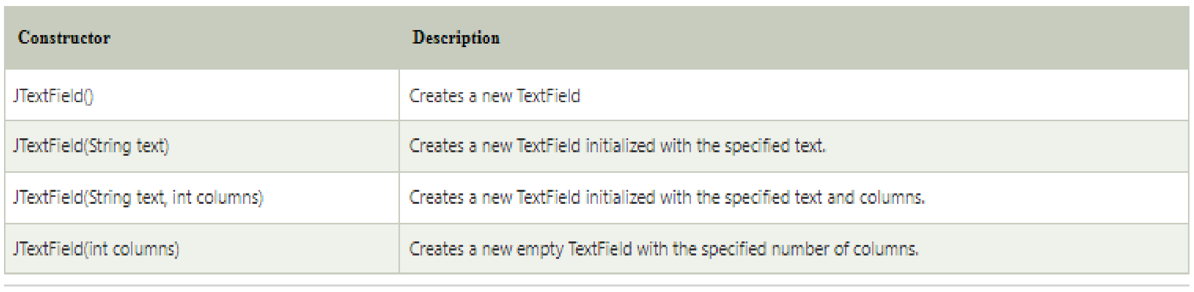
```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;
class MyJComponent extends JComponent {
    public void paint(Graphics g) {
      g.setColor(Color.green);
      g.fillRect(30, 30, 100, 100);
    }
}
public class JComponentExample {
    public static void main(String[] arguments) {
      MyJComponent com = new MyJComponent();
      // create a basic JFrame
      JFrame.setDefaultLookAndFeelDecorated(true);
      JFrame frame = new JFrame("JComponent Example");
      frame.setSize(300,200);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      // add the JComponent to main frame
      frame.add(com);
      frame.setVisible(true);
    }
}
```

Output:

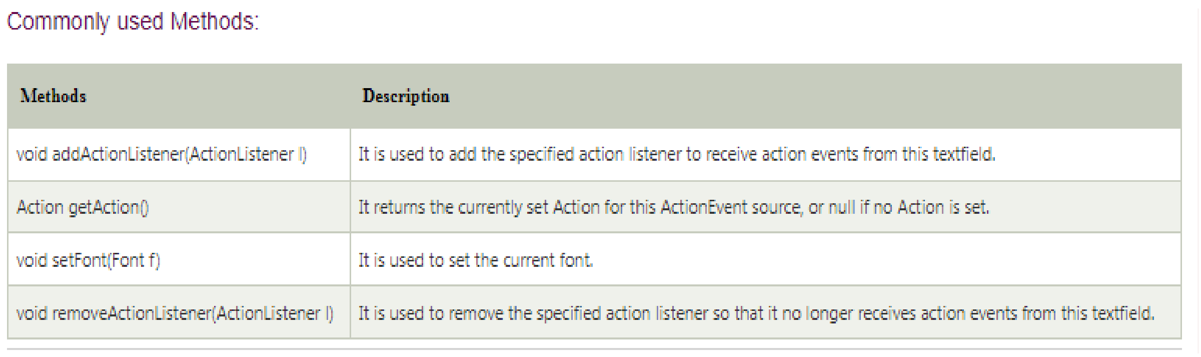


Text Components

The object of a TextField class is a text component that allows a user to enter a single line text and edit it. It inherits TextComponent class, which further inherits Component class. When we enter a key in the text field (like key pressed, key released or key typed), the event is sent to TextField..



Constructor	Description
JTextField()	Creates a new TextField
JTextField(String text)	Creates a new TextField initialized with the specified text.
JTextField(String text, int columns)	Creates a new TextField initialized with the specified text and columns.
JTextField(int columns)	Creates a new empty TextField with the specified number of columns.



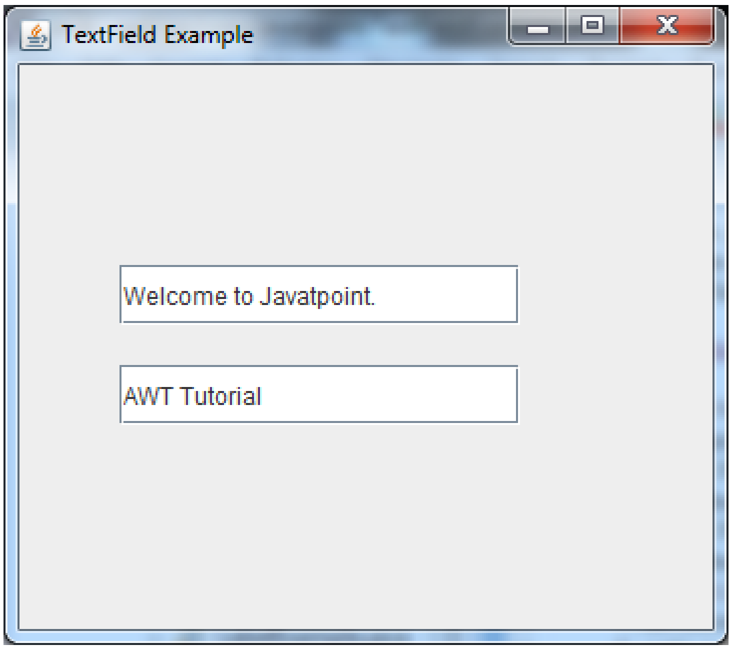
Commonly used Methods:

Methods	Description
void addActionListener(ActionListener l)	It is used to add the specified action listener to receive action events from this textfield.
Action getAction()	It returns the currently set Action for this ActionEvent source, or null if no Action is set.
void setFont(Font f)	It is used to set the current font.
void removeActionListener(ActionListener l)	It is used to remove the specified action listener so that it no longer receives action events from this textfield.

Java JTextField Example

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
    {
    JFrame f= new JFrame("TextField Example");
    JTextField t1,t2;
    t1=new JTextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new JTextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    }
    }
```

Output:



TextField Example

Welcome to Javatpoint.

AWT Tutorial

Menu

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass. The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

ava AWT MenuItem and Menu

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

AWT MenuItem class declaration

```
public class MenuItem extends MenuComponent implements Accessible
```

AWT Menu class declaration

```
public class Menu extends MenuItem implements MenuContainer, Accessible
```

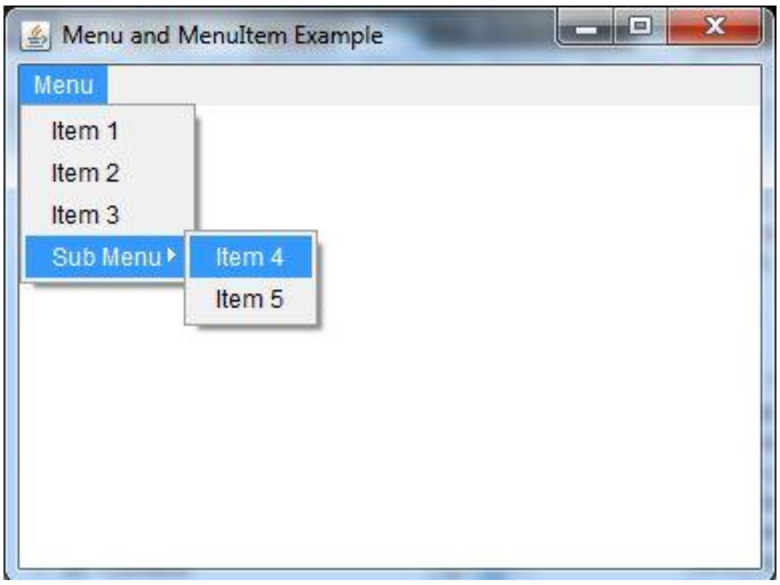
Java AWT MenuItem and Menu Example

```
import java.awt.*;
class MenuExample
{
    MenuExample(){
        Frame f= new Frame("Menu and MenuItem Example");
        MenuBar mb=new MenuBar();
        Menu menu=new Menu("Menu");
        Menu submenu=new Menu("Sub Menu");
        MenuItem i1=new MenuItem("Item 1");
        MenuItem i2=new MenuItem("Item 2");
        MenuItem i3=new MenuItem("Item 3");
        MenuItem i4=new MenuItem("Item 4");
        MenuItem i5=new MenuItem("Item 5");
        menu.add(i1);
        menu.add(i2);
        menu.add(i3);
        submenu.add(i4);
        submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}
}
```

Output:

Java Clock class

Java Clock class is used to provide an access to the current, date and time using a time zone. It inherits the Object class.



Because all date-time classes contain a now() function that uses the system clock in the default time zone, using the Clock class is not required. The aim of the Clock class is to allow you to plug in another clock whenever you need it. Instead of using a static method, applications utilise an object to get the current time. It simplifies the testing process. A method that requires a current instant can take a Clock as a parameter.

Java Clock Class Declaration

Let's see the declaration of java.time.Clock class.

```
public abstract class Clock extends Object
```

Methods of Java Clock Class

Java Clock class Example: getZone()

ClockExample1.java

```
import java.time.Clock;
public class ClockExample1 {
  public static void main(String[] args) {
    Clock c = Clock.systemDefaultZone();
    System.out.println(c.getZone());
  }
```

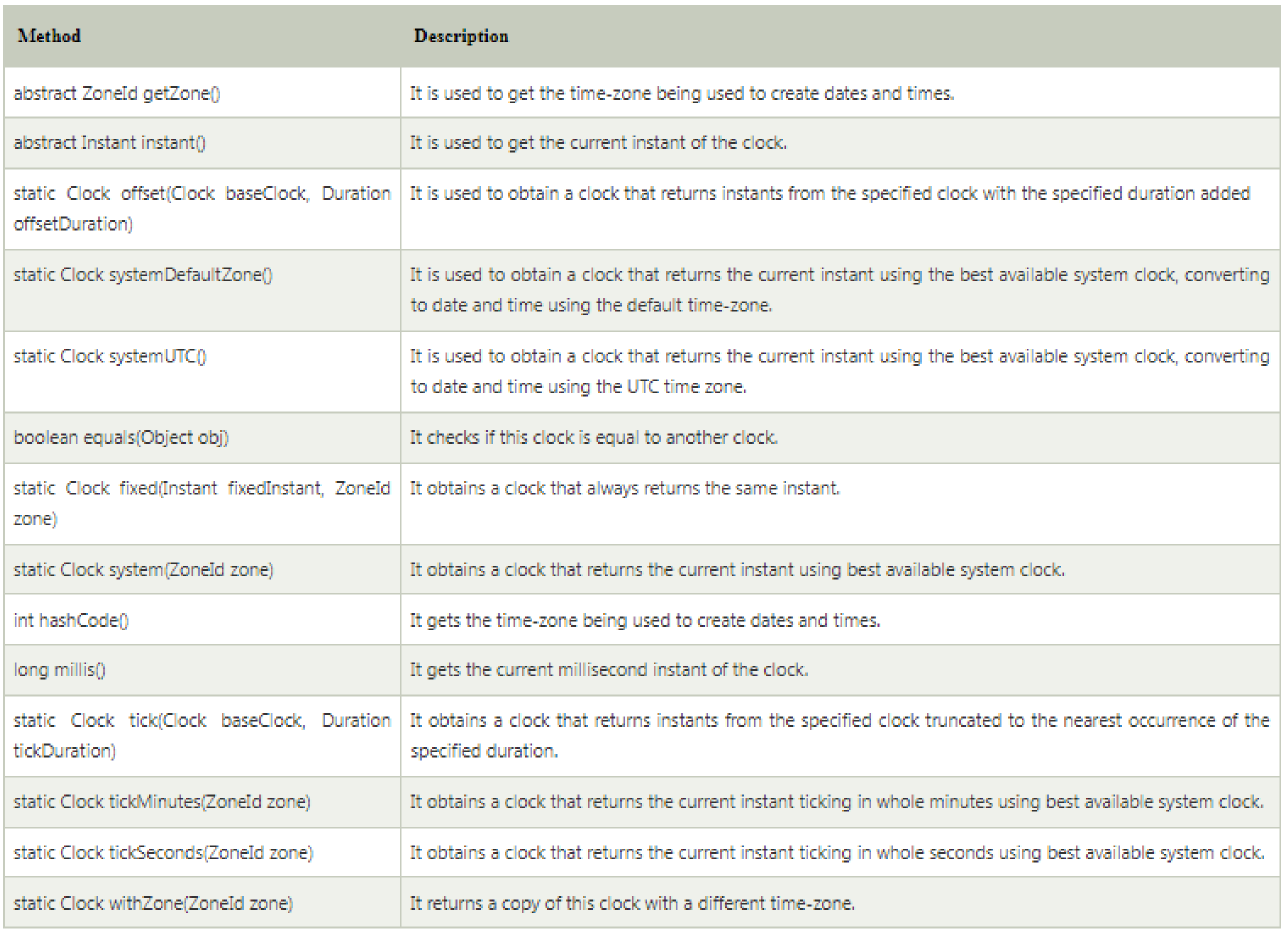
```java
}
```

Test it Now

Output:

```
Asia/Calcutta
```

Java Clock class Example: instant()



Method	Description
abstract ZoneId getZone()	It is used to get the time-zone being used to create dates and times.
abstract Instant instant()	It is used to get the current instant of the clock.
static Clock offset(Clock baseClock, Duration offsetDuration)	It is used to obtain a clock that returns instants from the specified clock with the specified duration added
static Clock systemDefaultZone()	It is used to obtain a clock that returns the current instant using the best available system clock, converting to date and time using the default time-zone.
static Clock systemUTC()	It is used to obtain a clock that returns the current instant using the best available system clock, converting to date and time using the UTC time zone.
boolean equals(Object obj)	It checks if this clock is equal to another clock.
static Clock fixed(Instant fixedInstant, ZoneId zone)	It obtains a clock that always returns the same instant.
static Clock system(ZoneId zone)	It obtains a clock that returns the current instant using best available system clock.
int hashCode()	It gets the time-zone being used to create dates and times.
long millis()	It gets the current millisecond instant of the clock.
static Clock tick(Clock baseClock, Duration tickDuration)	It obtains a clock that returns instants from the specified clock truncated to the nearest occurrence of the specified duration.
static Clock tickMinutes(ZoneId zone)	It obtains a clock that returns the current instant ticking in whole minutes using best available system clock.
static Clock tickSeconds(ZoneId zone)	It obtains a clock that returns the current instant ticking in whole seconds using best available system clock.
static Clock withZone(ZoneId zone)	It returns a copy of this clock with a different time-zone.

ClockExample2.java

```java
import java.time.Clock;
public class ClockExample2 {
  public static void main(String[] args) {
    Clock c = Clock.systemUTC();
    System.out.println(c.instant());
  }
}
```

Test it Now

Output:

```
2017-01-14T07:11:07.748Z
```

Java Clock class Example: systemUTC()

ClockExample3.java

```
import java.time.Clock;
public class ClockExample3 {
  public static void main(String[] args) {
    Clock c = Clock.systemUTC();
    System.out.println(c.instant());
  }
}
```

Test it Now

Output:

```
2017-01-14T07:11:07.748Z
```