1.Distinguish Multithreading and Multitasking.

S.NO	Multitasking	Multithreading
1.	In multitasking, users are allowed to perform many tasks by CPU.(or) Multitasking is a process of executing multiple tasks simultaneously.	In multithreading, multiple threads are created from a process.(or) Multithreading is a process of executing multiple threads simultaneously.
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated the same memory.
4.	In multitasking, the CPU is provided in order to execute many tasks at a time.	While in multithreading also, a CPU is provided in order to execute many threads from a process at a time.
5.	In multitasking, processes don’t share the same resources, each process is allocated separate resources.	While in multithreading, each process shares the same resources.
6.	Multitasking is slow compared to multithreading.	While multithreading is faster.
7.	In multitasking, termination of a process takes more time.	While in multithreading, termination of thread takes less time.
8.	Involves running multiple independent processes or tasks	Involves dividing a single process into multiple threads that can execute concurrently
9.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process
10.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process
11.	Examples: running multiple applications on a computer, running multiple servers on a network	Examples: splitting a video encoding task into multiple threads, implementing a responsive user interface in an application

2.Demonstrate the usage of inter-thread communication in Java with a suitable example.

Inter-thread Communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

1. **public final void** notify()

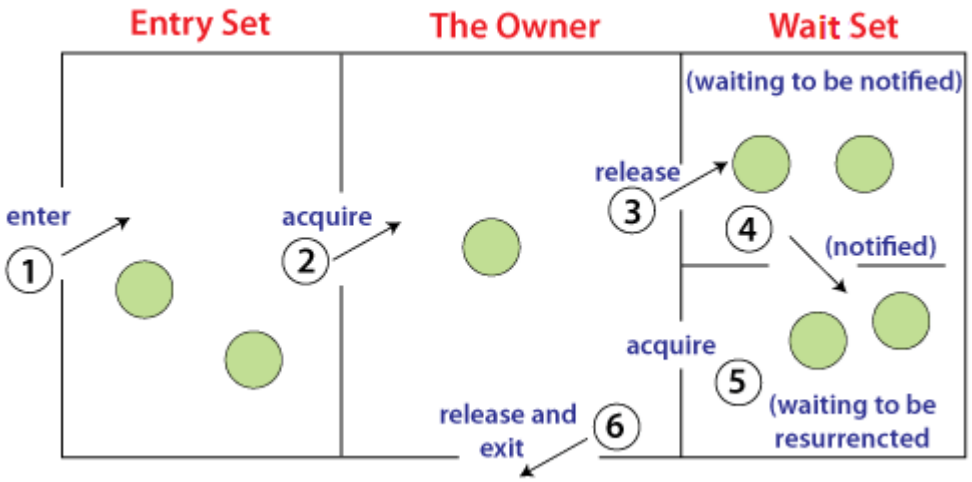
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

1. **public final void** notifyAll()

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.java

```
1. class Customer{
2. int account=5000;
3.
4. synchronized void withdraw(int amount){
5. System.out.println("going to withdraw...");
6.
7. if(account<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. try{wait();}catch(Exception e){}
10. }
11. account-=amount;
12. System.out.println("withdraw completed...");
13. }
```

```
14.
15. synchronized void deposit(int amount){
16. System.out.println("going to deposit...");
17. account+=amount;
18. System.out.println("deposit completed... ");
19. notify();
20. }
21. }
22.
23. class Test{
24. public static void main(String args[]){
25.  Customer c=new Customer();
26. new Thread(){
27. public void run(){c.withdraw(15000);}
28. }.start();
29. new Thread(){
30. public void run(){c.deposit(15000);}
31. }.start();
32.
33. }}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

3.Demonstrate the usage of thread synchronization in Java with a suitable example?
Synchronizing threads:-

Thread synchronization in java is a way of programming several threads to carry out independent tasks easily. It is capable of controlling access to multiple threads to a particular shared resource. The main reason for using thread synchronization are as follows:

- To prevent interference between threads.
- To prevent the problem of consistency.

Types of Thread Synchronization

In Java, there are two types of thread synchronization:

1. Process synchronization
2. Thread synchronization

Process synchronization- The process means a program in execution and runs independently isolated from other processes. CPU time, memory, etc resources are allocated by the operating system.

Thread synchronization- It refers to the concept where only one thread is executed at a time while other threads are in the waiting state. This process is called thread synchronization. **It is used because it avoids interference of thread and the problem of inconsistency.**

In java, thread synchronization is further divided into two types:

- Mutual exclusive- it will keep the threads from interfering with each other while sharing any resources.
- Inter-thread communication- It is a mechanism in java in which a thread running in the critical section is paused and another thread is allowed to enter or lock the same critical section that is executed.

Mutual Exclusive

It is used for preventing threads from interfering with each other and to keep distance between the threads. Mutual exclusive is achieved using the following:

- Synchronized Method
- Synchronized Block
- Static Synchronization

Lock Concept in Java

- Synchronization Mechanism developed by using the synchronized keyword in java language. It is built on top of the locking mechanism, this locking mechanism is taken care of by Java Virtual Machine (JVM). The synchronized keyword is only applicable for methods and blocks, it can't apply to classes and variables. Synchronized keyword in java creates a block of code is known as a critical section. To enter into the critical section thread needs to obtain the corresponding object's lock.

Java Synchronized Method

If we use the Synchronized keywords in any method then that method is Synchronized Method.

- It is used to lock an object for any shared resources.
- The object gets the lock when the synchronized method is called.
- The lock won't be released until the thread completes its function.

Syntax:

```
Acess_modifiers synchronized return_type method_name (Method_Parameters) {

// Code of the Method.

}
```

Example program:-

```
public class SynchronizedMethodEx extends Thread {

   public synchronized void run(){//synchronized method

      for(int i=1;i<=3;i++)

      {
```

```
            System.out.println(Thread.currentThread().getName());
        }
    }
    public static void main(String args[])
    {
        SynchronizedMethodEx s1=new SynchronizedMethodEx();
        SynchronizedMethodEx s2=new SynchronizedMethodEx();
        SynchronizedMethodEx s3=new SynchronizedMethodEx();
        s1.start();//thread0
        s2.start();//thread1
        s3.start();//thread2
    }
}
```

output:-

```
Thread-0
Thread-0
Thread-0
Thread-1
Thread-1
Thread-1
Thread-2
Thread-2
Thread-2
```

Synchronized Block

- Suppose you don't want to synchronize the entire method, you want to synchronize few lines of code in the method, then a synchronized block helps to synchronize those few lines of code. It will take the object as a parameter. It will work the same as Synchronized Method. In the case of synchronized method lock accessed is on the method but in the case of synchronized block lock accessed is on the object.

Syntax:

```
synchronized (object) {
//code of the block.
}
```

Program to understand the Synchronized Block:

Example:-

```
public class SynchronizedMethodEx extends Thread {
    public  void run(){
        synchronized(this){//synchronized block
        for(int i=1;i<=3;i++)
        {
            System.out.println(Thread.currentThread().getName());
        }
        }
    }
    public static void main(String args[])
    {
        SynchronizedMethodEx s1=new SynchronizedMethodEx();
        SynchronizedMethodEx s2=new SynchronizedMethodEx();
        SynchronizedMethodEx s3=new SynchronizedMethodEx();
        s1.start();//thread0
        s2.start();//thread1
        s3.start();//thread2
    }
}
```

output:-

```
Thread-0
Thread-0
```

Thread-0

Thread-1

Thread-1

Thread-1

Thread-2

Thread-2

Thread-2

Static Synchronization

- In java, every object has a single lock (monitor) associated with it. The thread which is entering into synchronized method or synchronized block will get that lock, all other threads which are remaining to use the shared resources have to wait for the completion of the first thread and release of the lock.
- Suppose in the case of where we have more than one object, in this case, two separate threads will acquire the locks and enter into a synchronized block or synchronized method with a separate lock for each object at the same time. To avoid this, we will use static synchronization.
- In this, we will place synchronized keywords before the static method. In static synchronization, lock access is on the class not on object and Method.

Syntax:

```
synchronized static return_type method_name (Parameters) {

//code

}
```

Example Program:-

```
public class SynchronizedMethodEx implements Runnable {
   public static synchronized void display(){

      for(int i=1;i<=3;i++)
      {
         System.out.println(Thread.currentThread().getName());
      }

   }
   public void run(){
      this.display();
   }

   public static void main(String args[])
   {
      SynchronizedMethodEx s1=new SynchronizedMethodEx();
      Thread t1=new Thread(s1);
      SynchronizedMethodEx s2=new SynchronizedMethodEx();
      Thread t2=new Thread(s2);
```

```
        t1.start();//thread0
        t2.start();//thread1

    }
}
```

Output:-

```
Thread-0
Thread-0
Thread-0
Thread-1
Thread-1
Thread-1
```

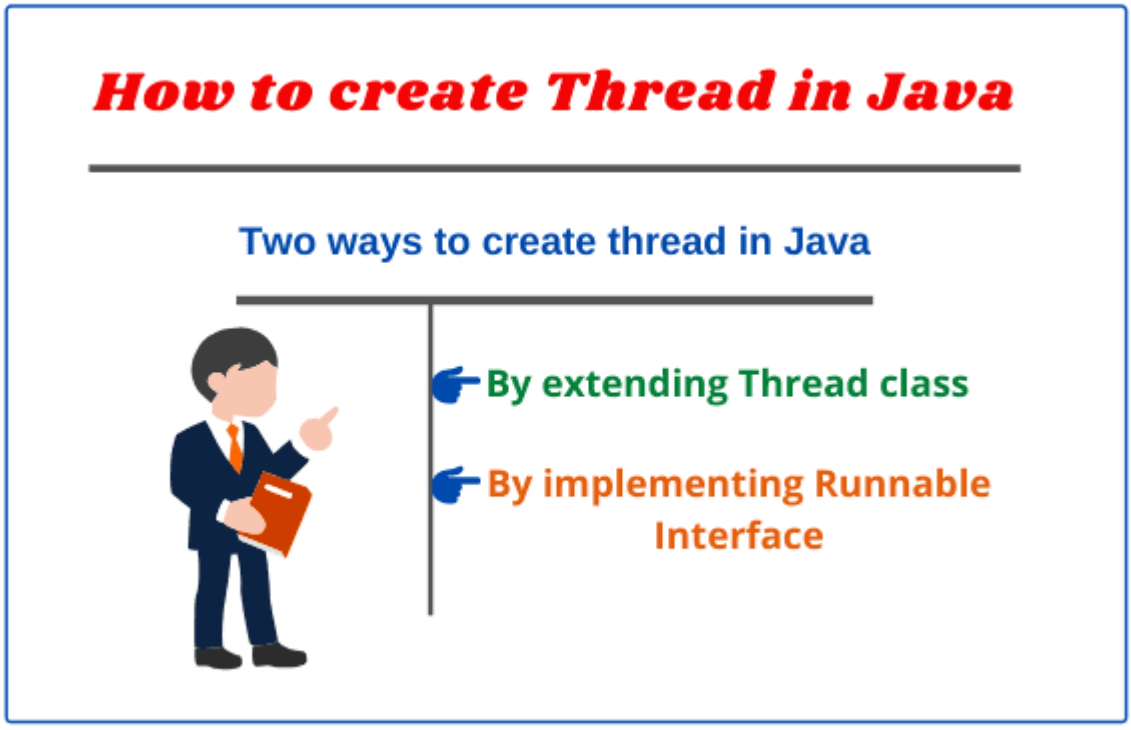
4.How many types of ways are there for creating threads? Explain each with an example.

Creating threads

Creating threads in Java | We know that every Java program has at least one thread called main thread. When a program starts, main thread starts running immediately. Apart from this main thread, we can also create our own threads in a program that is called child thread. Every child threads create from its main thread known as parent thread.

There are two ways to create a new thread in Java. They are as follows:

1. One is by extending java.lang.Thread class
2. Another is by implementing java.lang.Runnable interface



1.By Extending Thread class

Extending Thread class is the easiest way to create a new thread in Java. The following steps can be followed to create your own thread in Java.

1. Create a class that extends the Thread class. In order to extend a thread, we will use a keyword extends. The Thread class is found in java.lang package.

The syntax for creating a new class that extends Thread class is as follows:

```
Class MyClass extends Thread
```

2. Now in this newly created class, define a method run(). Here, run() method acts as entry point of the new thread. The run() method contains the actual task that thread will perform. Thus, we override run() method of Thread class.

```
public void run()

{

 // statements to be executed.

}
```

3. Create an object of newly created class so that run() method is available for execution. The syntax to create an object of Myclass is as follows:

```
MyClass obj = new Myclass();
```

4. Now create an object of Thread class and pass the object reference variable created to the constructor of Thread class.

```
Thread t = new Thread(obj);

or,

 Thread t = new Thread(obj, "threadname");
```

5. Run the thread. For this, we need to call to start() method of Thread class because we cannot call run() method directly. The syntax to call start() method is as follows:

```
t.start();
```

Now, the thread will start execution on the object of Myclass, and statements inside run() method will be executed while calling it. By following all the above steps, you can create a Thread in Java.

Example Program 1:

```
class MyClass extends Thread{
    @Override
```

```
    public void run()
    {
       for(int i=1;i<=2;i++)
       {
          System.out.println("Child Thread");
       }
    }
    public static void main(String args[])
    {
    MyClass obj=new MyClass();
    Thread t=new Thread(obj);
    t.start();
    }
}
```

Out Put:-

Child Thread
Child Thread

2. By Implementing Runnable Interface

1. Threads can also be created by implementing Runnable interface of java.land package. Creating a thread by implementing Runnable interface is very similar to creating a thread by extending Thread class.
2. All steps are similar for creating a thread by implementing Runnable interface except the first step. The first step is as follows:
3. 1. To create a new thread using this method, create a class that implements Runnable interface of java.lang package.
4. The syntax for creating a new class that implements Runnable interface is as follows:
5. class Myclass implements Runnable

Example Program:-

```
class MyThread implements Runnable
{
  public void run()
  {
    for(int i=1;i<=5;i++)
    {
       System.out.println("My Child Thread");
    }

  }
```

```
    public static void main(String args[])
    {
        // Create an object of MyThread class.
        MyThread r=new MyThread();
        // Create an object of Thread class and pass reference variable r to
Thread class constructor.
        Thread t=new Thread(r);
        t.start();
// This thread will execute statements inside run() method of MyThread
object.

    }
}
```

Out Put:-

```
My Child Thread
My Child Thread
My Child Thread
My Child Thread
My Child Thread
```

1.How do you handle the mouse events using java AWT. Explain.

Handling mouse events

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 to handle mouse events.

Methods of MouseListener interface

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

1. **public abstract void** mouseClicked(MouseEvent e);
2. **public abstract void** mouseEntered(MouseEvent e);
3. **public abstract void** mouseExited(MouseEvent e);
4. **public abstract void** mousePressed(MouseEvent e);
5. **public abstract void** mouseReleased(MouseEvent e);

The signature of 2 methods found in MouseMotionListener interface are given below:

1.public abstract void mouseMoved(MouseEvent e);

2.public abstract void mouseDragged(MouseEvent e);

Java MouseListener Example

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class MouseListenerExample extends Frame implements MouseListener,MouseMotion
   Listener{
4.     Label l;
5.     MouseListenerExample(){
6.         addMouseListener(this);
7.
8.         l=new Label();
9.         l.setBounds(20,50,100,20);
10.        add(l);
11.        setSize(300,300);
12.        setLayout(null);
13.        setVisible(true);
14.    }
15.    public void mouseClicked(MouseEvent e) {
16.        l.setText("Mouse Clicked");
17.    }
18.    public void mouseEntered(MouseEvent e) {
19.        l.setText("Mouse Entered");
20.    }
21.    public void mouseExited(MouseEvent e) {
22.        l.setText("Mouse Exited");
23.    }
24.    public void mousePressed(MouseEvent e) {
25.        l.setText("Mouse Pressed");
26.    }
27.    public void mouseReleased(MouseEvent e) {
28.        l.setText("Mouse Released");
29.    }
30.public abstract void mouseMoved(MouseEvent e){
31.        l.setText("");
32.}
33.public abstract void mouseDragged(MouseEvent e){
34.        l.setText("Mouse Dragged");
```

```
35.}
36.public static void main(String[] args) {
37.    new MouseListenerExample();
38.}
39.}
```

Output:

2.Demonstrate the usage of adapter classes with a suitable example.

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.

For example,

MouseListener	MouseAdapter
void mouseClicked(MouseEvent me)	*void mouseClicked(MouseEvent me){ }*
void mouseEntered(MouseEvent me)	*void mouseEntered(MouseEvent me) { }*
void mouseExited(MouseEvent me)	*void mouseExited(MouseEvent me) { }*
void mousePressed(MouseEvent me)	*void mousePressed(MouseEvent me) { }*
void mouseReleased(MouseEvent me)	*void mouseReleased(MouseEvent me) { }*

Table: Commonly used Listener Interfaces implemented by Adapter Classes

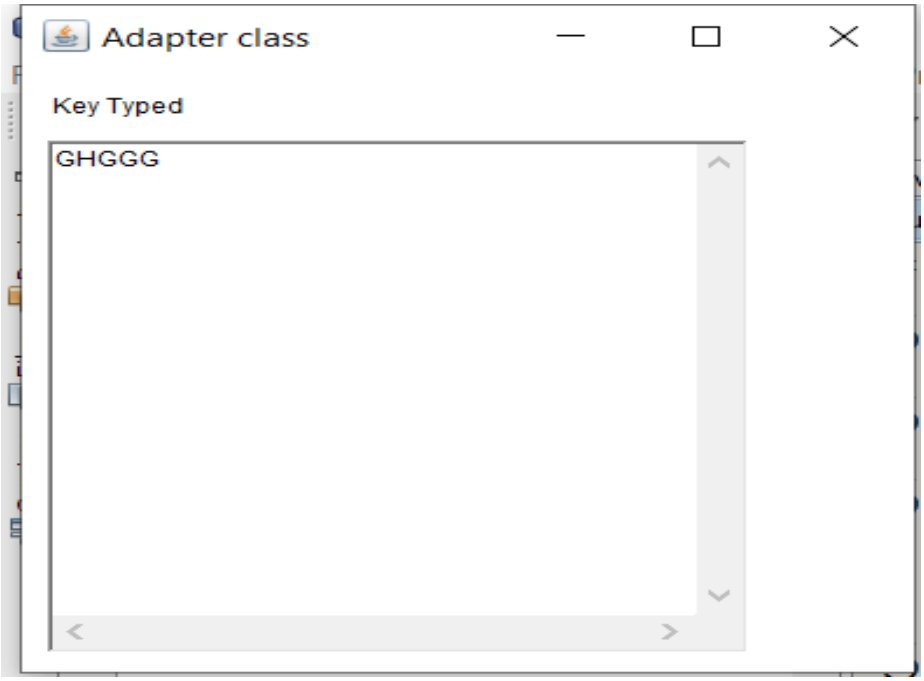
Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

The adapter classes are found in **java.awt.event package.**

Example Program:-

KeyAdapterExample.java

```
import java.awt.*;
   import java.awt.event.*;
   public class KeyAdapterExample extends KeyAdapter{
      Frame f;
      Label l;
      TextArea area;
      KeyListenerExample(){
         f=new Frame("Adapter class");
         l=new Label();
         l.setBounds(20,50,100,20);
         area=new TextArea();
         area.setBounds(20,80,300, 300);
         area.addKeyListener(this);
         f.add(l);
 f.add(area);
         f.setSize(400,400);
         f.setLayout(null);
         f.setVisible(true);
      }

      public void keyTyped(KeyEvent e) {
         l.setText("Key Typed");
      }

      public static void main(String[] args) {
         new KeyAdapterExample();
      }
   }
```



3.Explain about Event classes, Event sources, Event listeners and the relationship among them.

Event sources

A source is an object that generates an event. Generally sources are components. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

Syntax:-

public void addTypeListener (TypeListener el)

Here, Type is the name of the event, and el is a reference to the event listener.
For example, the method that registers a keyboard event listener is called **addKeyListener( ).**
A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeTypeListener(TypeListener el)

Event Listeners and Event Classes

A listener is an object that is notified when an event occurs. It has two major requirements.

1.It must have been registered with one or more sources to receive notifications about specific types of events.

2. It must implement methods to receive and process these notifications.The methods that receive and process events are defined in a set of interfaces found in java.awt.event package.

An event listener registers with an event source to receive notifications about the events of a particular type. Various event listener interfaces and event classes defined in the java.awt.event

Event Classes	Listener Interfaces	Occurs
ActionEvent	ActionListener	Button click Menu item click
MouseEvent	MouseListener	Mouse state changed
	MouseMotionListener	Mouse moved or dragged
MouseWheelEvent	MouseWheelListener	Mouse wheel rotated
KeyEvent	KeyListener	Keyboard key state changed
ItemEvent	ItemListener	Checkbox clicked or list/choice item selected
TextEvent	TextListener	Text field/Text area state changed
AdjustmentEvent	AdjustmentListener	Adjustment value changed for scrollpane
WindowEvent	WindowListener	Window state changed
ComponentEvent	ComponentListener	Component state changed
ContainerEvent	ContainerListener	Component added/removed to/from container
FocusEvent	FocusListener	Component gains/losses keyboard focus

Example Program:-

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

//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);//button source register

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

}
OutPut:-

4.What is a Layout manager? Explain the different types of Layout managers in detail.

Layout Managers:-

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

LayoutManager

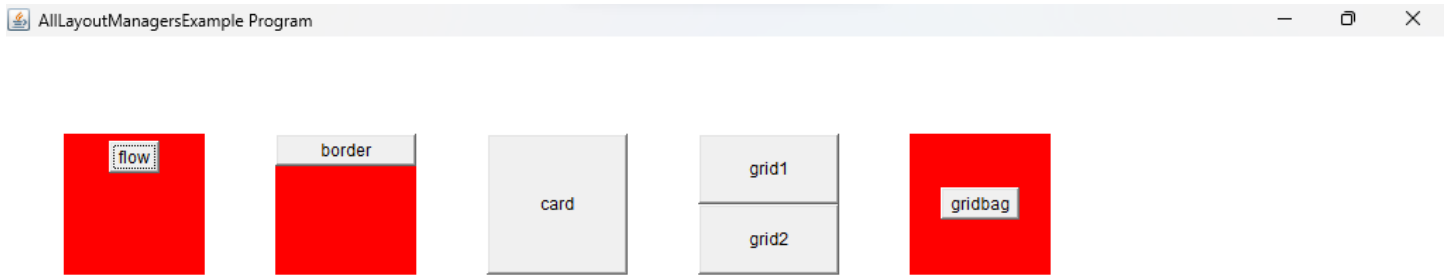
- **FlowLayout**: It arranges the components in a container like the words on a page. It fills the top line from **left to right and top to bottom**. The components are arranged in the order as they are added i.e. first components appears at top left, if the container is not wide enough to display all the components, it is wrapped around the line. Vertical and horizontal gap between components can be controlled. The components can be **left, center or right aligned.**
- **BorderLayout**: It arranges all the components along the edges or the middle of the container i.e. **top, bottom, right and left** edges of the area. The components added to the top or bottom gets its preferred height, but its width will be the width of the container and also the components added to the left or right gets its preferred width, but its height will be the remaining height of the container. The components added to the center gets neither its preferred height or width. It covers the remaining area of the container.
- **GridLayout**: It arranges all the components in a grid of **equally sized cells**, adding them from the **left to right** and **top to bottom**. Only one component can be placed in a cell and each region of the grid will have the same size. When the container is resized, all cells are automatically resized. The order of placing the components in a cell is determined as they were added.

- **GridBagLayout**: It is a powerful layout which arranges all the components in a grid of cells and maintains the aspect ration of the object whenever the container is resized. In this layout, cells may be different in size. It assigns a consistent horizontal and vertical gap among components. It allows us to specify a default alignment for components within the columns or rows.
- **CardLayout**: It arranges two or more components having the same size. The components are **arranged in a deck**, where all the cards of the same size and the **only top card are visible at any time**. The first component added in the container will be kept at the top of the deck. The default gap at the left, right, top and bottom edges are zero and the card components are displayed either **horizontally or vertically.**

Example Program:-

```
import java.awt.*;
import java.awt.event.*;
public class AllLayoutManagersExample extends Frame {
    public AllLayoutManagersExample()
    {
        Button b1=new Button("flow");
        Button b2=new Button("border");
        Button b3=new Button("card");
        Button b4=new Button("grid1");
        Button b5=new Button("grid2");
        Button b6=new Button("gridbag");

        //Flow Layout
        Panel p1=new Panel();
        p1.setBounds(50,100,100,100);
        p1.setBackground(Color.red);
        p1.setLayout(new FlowLayout());
        p1.add(b1);
        add(p1);

        //Border Layout
        Panel p2=new Panel();
        p2.setBounds(200,100,100,100);
        p2.setBackground(Color.red);
        p2.setLayout(new BorderLayout());
        p2.add(b2,BorderLayout.NORTH);
        add(p2);

        //Card Layout
        Panel p3=new Panel();
        p3.setBounds(350,100,100,100);
        p3.setBackground(Color.red);
        p3.setLayout(new CardLayout());
        p3.add(b3);
        add(p3);

        //GridLayout
        Panel p4=new Panel();
        p4.setBounds(500,100,100,100);
        p4.setBackground(Color.red);
        p4.setLayout(new GridLayout(2,1));
```

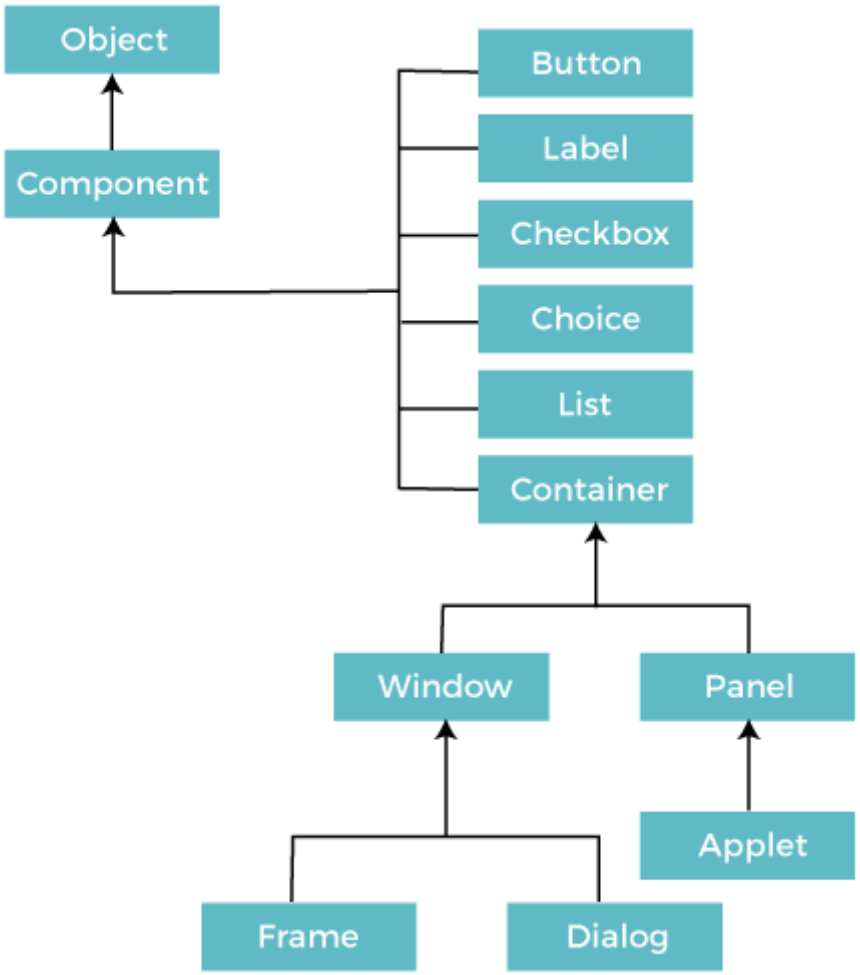
```
        p4.add(b4);
        p4.add(b5);
        add(p4);

        //GridBagLayout
        Panel p5=new Panel();
        p5.setBounds(650,100,100,100);
        p5.setBackground(Color.red);
        p5.setLayout(new GridBagLayout());
        p5.add(b6);
        add(p5);


        setSize(300,300);
        setTitle("AllLayoutManagersExample Program");
        setLayout(null);
        setVisible(true);

//Frame window closing
        this.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent w){
                dispose();
            }
        });

    }
    public static void main(String args[])
    {
        new AllLayoutManagersExample();
    }
}
```



5.Sketch the AWT hierarchy with a neat diagram and explain some of the user interface components.

AWT hierarchy

The hierarchy of Java AWT classes are given below.



User Interface Components:-

1.Labels

The object of the Label class is a component for placing text in a container. It is used to display a single line of **read only text**. The text can be changed by a programmer but a user cannot edit it directly.

It is called a passive control as it does not create any event when it is accessed. To create a label, we need to create the object of **Label** class.

AWT Label Fields

The java.awt.Component class has following fields:

1. **static int LEFT:** It specifies that the label should be left justified.
2. **static int RIGHT:** It specifies that the label should be right justified.
3. **static int CENTER:** It specifies that the label should be placed in center.

Label class Constructors

Sr. no.	Constructor	Description
1.	Label()	It constructs an empty label.
2.	Label(String text)	It constructs a label with the given string (left justified by default).
3.	Label(String text, int alignement)	It constructs a label with the specified string and the specified alignment.

Label Class Methods

Specified

Sr. no.	Method name	Description
1.	void setText(String text)	It sets the texts for label with the specified text.
2.	Void setAlignment(int alignment)	It sets the alignment for label with the specified alignment.
3.	String getText()	It gets the text of the label
4.	int getAlignment()	It gets the current alignment of the label.

Java AWT Label Example

In the following example, we are creating two labels l1 and l2 using the Label(String text) constructor and adding them into the frame.

LabelExample.java

```
1. import java.awt.*;
2.
3. public class LabelExample {
4. public static void main(String args[]){
5.
6.    // creating the object of Frame class and Label class
7.    Frame f = new Frame ("Label example");
8.    Label l1, l2;
9.
10.   // initializing the labels
11.   l1 = new Label ("First Label.");
```

```
12.     l2 = new Label ("Second Label.");
13.
14.     // set the location of label
15.     l1.setBounds(50, 100, 100, 30);
16.     l2.setBounds(50, 150, 100, 30);
17.
18.     // adding labels to the frame
19.     f.add(l1);
20.     f.add(l2);
21.
22.     // setting size, layout and visibility of frame
23.     f.setSize(400,400);
24.     f.setLayout(null);
25.     f.setVisible(true);
26. }
27. }
```

Output:

b)Buttons

Java AWT Button

A button is basically a control component with a label that generates an event when pushed. The **Button** class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

When we press a button and release it, AWT sends an instance of **ActionEvent** to that button by calling **processEvent** on the button. The **processEvent** method of the button receives the all the events, then it passes an action event by calling its own method **processActionEvent**. This method passes the action event on to action listeners that are interested in the action events generated by the button.

To perform an action on a button being pressed and released, the **ActionListener** interface needs to be implemented. The registered new listener can receive events from the button by calling **addActionListener** method of the button.

AWT Button Class Declaration

1. **public class** Button **extends** Component **implements** Accessible

Button Class Constructors

Following table shows the types of Button class constructors

Sr. no.	Constructor	Description
1.	Button( )	It constructs a new button with an empty string i.e. it has no label.
2.	Button (String text)	It constructs a new button with given string as its label.

Button Class Methods

Sr. no.	Method	Description
1.	void setText (String text)	It sets the string message on the button
2.	String getText()	It fetches the String message on the button.
3.	void setLabel (String label)	It sets the label of button with the specified string.
4.	String getLabel()	It fetches the label of the button.
7.	void addActionListener(ActionListener l)	It adds the specified action listener to get the action events from the button.
8.	String getActionCommand()	It returns the command name of the action event fired by the button.
9.	void removeActionListener (ActionListener l)	It removes the specified action listener so that it no longer receives action events from the button.
10.	void setActionCommand(String command)	It sets the command name for the action event given by the button.

Note: The Button class inherits methods from java.awt.Component and java.lang.Object classes.

Java AWT Button Example

Example 1:

ButtonExample.java

```
1. import java.awt.*;
2. public class ButtonExample {
3. public static void main (String[] args) {
4.
5.     // create instance of frame with the label
6.     Frame f = new Frame("Button Example");
7.
8.     // create instance of button with label
9.     Button b = new Button("Click Here");
10.
11.    // set the position for the button in frame
12.    b.setBounds(50,100,80,30);
13.
14.    // add button to the frame
15.    f.add(b);
16.    // set size, layout and visibility of frame
17.    f.setSize(400,400);
18.    f.setLayout(null);
19.    f.setVisible(true);
20. }
21. }
```

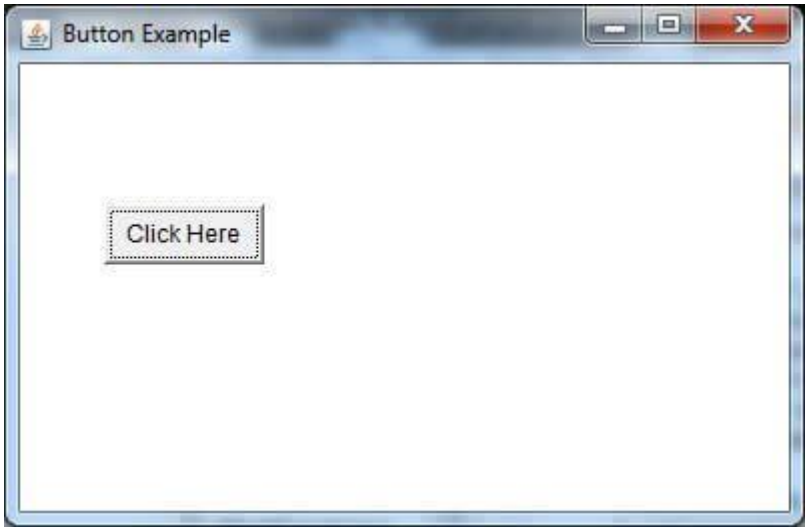
To compile the program using command prompt type the following commands

```
1. C:\Users\Anurati\Desktop\abcDemo>javac ButtonExample.java
```

If there's no error, we can execute the code using:

```
1. C:\Users\Anurati\Desktop\abcDemo>java ButtonExample
```

Output:



c)Scrollbars

The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

It can be added to top-level container like Frame or a component like Panel. The Scrollbar class extends the **Component** class.

AWT Scrollbar Class Declaration

1. **public class** Scrollbar **extends** Component **implements** Adjustable, Accessible

Scrollbar Class Fields

The fields of java.awt.Image class are as follows:

- **static int HORIZONTAL** - It is a constant to indicate a horizontal scroll bar.
- **static int VERTICAL** - It is a constant to indicate a vertical scroll bar.

Scrollbar Class Constructors

Sr. no.	Constructor	Description
1	Scrollbar()	Constructs a new vertical scroll bar.
2	Scrollbar(int orientation)	Constructs a new scroll bar with the specified orientation.
3	Scrollbar(int orientation, int value, int visible, int minimum, int maximum)	Constructs a new scroll bar with the specified orientation, initial value, visible amount, and minimum and maximum values.

Where the parameters,

- **orientation**: specifiey whether the scrollbar will be horizontal or vertical.
- **Value**: specify the starting position of the knob of Scrollbar on its track.
- **Minimum**: specify the minimum width of track on which scrollbar is moving.
- **Maximum**: specify the maximum width of track on which scrollbar is moving.

Scrollbar Class Methods

Sr. no.	Method name	Description
1.	int getMaximum()	It gets the maximum value of the scroll bar.
2.	int getMinimum()	It gets the minimum value of the scroll bar.
3.	int getOrientation()	It returns the orientation of scroll bar.
4.	int getUnitIncrement()	It fetches the unit increment of the scroll bar.
5.	int getValue()	It fetches the current value of scroll bar.
6.	int getVisibleAmount()	It fetches the visible amount of scroll bar.
7.	void setBlockIncrement(int v)	It sets the block increment from scroll bar.
8.	void setMaximum (int newMaximum)	It sets the maximum value of the scroll bar.
9.	void setMinimum (int newMinimum)	It sets the minimum value of the scroll bar.
10.	void setOrientation (int orientation)	It sets the orientation for the scroll bar.
11.	void setUnitIncrement(int v)	It sets the unit increment for the scroll bar.
12.	void setValue (int newValue)	It sets the value of scroll bar with the given argument value.
13.	void setValues (int value, int visible, int minimum, int maximum)	It sets the values of four properties for scroll bar: value, visible amount, minimum and maximum.
14.	void setVisibleAmount (int newAmount)	It sets the visible amount of the scroll bar.

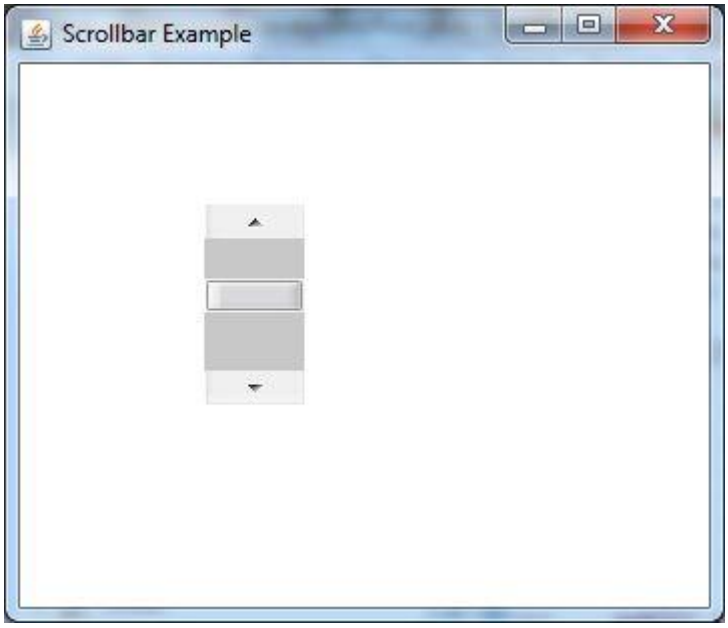
Java AWT Scrollbar Example

In the following example, we are creating a scrollbar using the Scrollbar() and adding it into the Frame.

ScrollbarExample1.java

```
1. // importing awt package
2. import java.awt.*;
3.
4. public class ScrollbarExample1 {
5.
6. // class constructor
7. ScrollbarExample1() {
8.
9.       // creating a frame
10.          Frame f = new Frame("Scrollbar Example");
11.      // creating a scroll bar
12.          Scrollbar s = new Scrollbar();
13.
14.      // setting the position of scroll bar
15.          s.setBounds (100, 100, 50, 100);
16.
17.      // adding scroll bar to the frame
18.          f.add(s);
19.
20.      // setting size, layout and visibility of frame
21.          f.setSize(400, 400);
22.          f.setLayout(null);
23.          f.setVisible(true);
24. }
25.
26. // main method
27. public static void main(String args[]) {
28.      new ScrollbarExample1();
29. }
30. }
```

Output:



6.Distinguish Event Listeners from Event Adapters.

Event Listeners:-

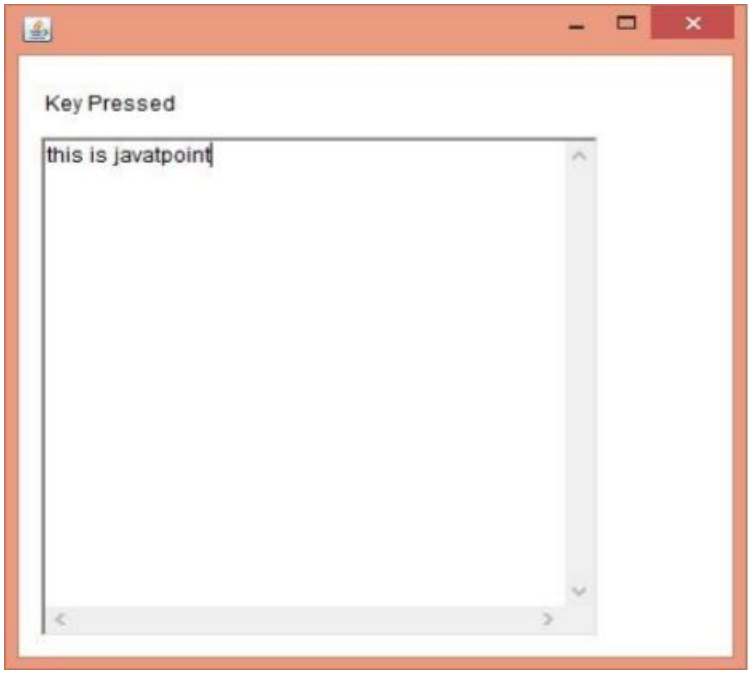
When we implement a listener interface in any class then we must have to implement all the methods declared in that interface because all the methods in an interface are abstract and must be override in class which implement it. For example consider the following program which demonstrates handling of key events by implementing listener interface.

KeyListenerExample.java

```
import java.awt.*;
  import java.awt.event.*;
  public class KeyListenerExample extends Frame implements KeyListener{
    Label l;
    TextArea area;
    KeyListenerExample(){
      l=new Label();
      l.setBounds(20,50,100,20);
      area=new TextArea();
      area.setBounds(20,80,300, 300);
      area.addKeyListener(this);
      add(l);
     add(area);
      setSize(400,400);
      setLayout(null);
      setVisible(true);
    }
    public void keyPressed(KeyEvent e) {
```

```
        l.setText("Key Pressed");
    }
    public void keyReleased(KeyEvent e) {
        l.setText("Key Released");
    }
    public void keyTyped(KeyEvent e) {
        l.setText("Key Typed");
    }

    public static void main(String[] args) {
        new KeyListenerExample();
    }
}
```

OUTPUT:



Our above example for handling key events implements KeyListener interface so the class KeyEvent has to implement all the three methods listed below.

1. public void keyTyped(KeyEvent e)
2. public void keyPressed(KeyEvent e)
3. public void keyReleased(KeyEvent e)

This can be inconvenient because if we want to use only one or two methods in your program then? It is not suitable solution to implement all the methods every time even we don't need them.

Adapter class makes it easy to deal with this situation. An adapter class provides empty implementations of all methods defined by that interface.

Adapter classs are very useful if you want to override only some of the methods defined by that interface. Here the names of Listener interface and corresponding interface are given which are in java.awt.event package.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListner
ContainerAdapter	ContainerListner
FocusAdapter	FocusListner
KeyAdapter	KeyListner
MouseAdapter	MouseListner
MouseMotionAdapter	MouseMotionListner
WindowAdapter	WindowListner

Now consider a situation in which we want to perform any action only when key Typed then we should have to override keyTyped() method. In this case if we use the listener interface then we must have to implements all the above three methods, the adapter class KeyAdapter will minimize programmer's work. Following example demonstrate the use of Adapter class in place of Listener interface.

KeyAdapterExample.java

```
import java.awt.*;
   import java.awt.event.*;
   public class KeyAdapterExample extends KeyAdapter{
      Frame f;
      Label l;
      TextArea area;
      KeyAdapterExample(){
         f=new Frame("Adapter class");
         l=new Label();
         l.setBounds(20,50,100,20);
         area=new TextArea();
         area.setBounds(20,80,300, 300);
         area.addKeyListener(this);
         f.add(l);
 f.add(area);
         f.setSize(400,400);
         f.setLayout(null);
```

```
        f.setVisible(true);
    }

    public void keyTyped(KeyEvent e) {
        l.setText("Key Typed");
    }

    public static void main(String[] args) {
        new KeyAdapterExample();
    }
}
```

7.Delegation Event Model is essential in Event handling. Justify the statement.

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.
Steps to perform Event Handling Following steps are required to perform event handling:
1. Register the component with the Listener
2. Implement the concerned interface

Delegation event model

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.

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.

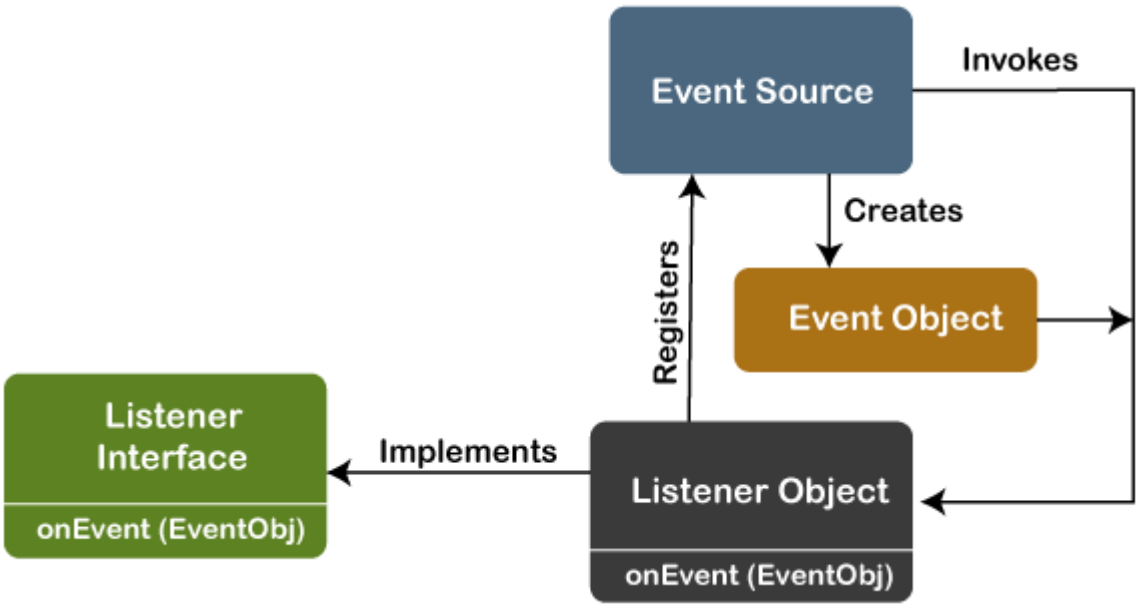


Fig.Delegation Event Model

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.

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:

1. **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.TooManyListenersException

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.

8.How do you handle the Key events using java AWT. Explain.

Handling keyboard events

KeyboardEvent objects describe a user interaction with the keyboard; each event describes a single interaction between the user and a key (or combination of a key with modifier keys) on the keyboard. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package. It has three methods to handle mouse events.

Methods of KeyListener interface

The signature of 3 methods found in keyListener interface are given below:

void keyPressed(KeyEvent ke)

void keyReleased(KeyEvent ke)

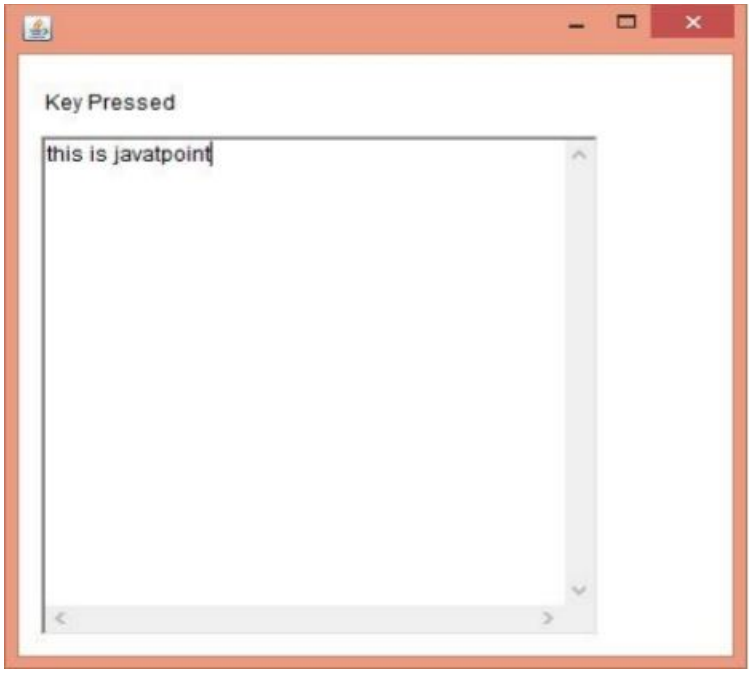
void keyTyped(KeyEvent ke)

Example program:-

```
import java.awt.*;
  import java.awt.event.*;
  public class KeyListenerExample extends Frame implements KeyListener{
    Label l;
    TextArea area;
    KeyListenerExample(){
      l=new Label();
      l.setBounds(20,50,100,20);
      area=new TextArea();
      area.setBounds(20,80,300, 300);
      area.addKeyListener(this);
      add(l);
     add(area);
      setSize(400,400);
      setLayout(null);
      setVisible(true);
    }
```

```
    public void keyPressed(KeyEvent e) {
       l.setText("Key Pressed");
    }
    public void keyReleased(KeyEvent e) {
       l.setText("Key Released");
    }
    public void keyTyped(KeyEvent e) {
       l.setText("Key Typed");
    }

    public static void main(String[] args) {
       new KeyListenerExample();
    }
}
```

OUTPUT:



1.Differentiate applets from application programs.

Difference between applets and applications

Parameters	Java Application	Java Applet
Definition	Applications are just like a Java program that can be executed independently without using the web browser.	Applets are small Java programs that are designed to be included with the HTML web document. They require a

Parameters	Java Application	Java Applet
		Java-enabled web browser for execution.
main () method	The application program requires a main() method for its execution.	The applet does not require the main() method for its execution instead init() method is required.
Compilation	The “javac” command is used to compile application programs, which are then executed using the “java” command.	Applet programs are compiled with the “javac” command and run using either the “appletviewer” command or the web browser.
File access	Java application programs have full access to the local file system and network.	Applets don’t have local disk and network access.
Access level	Applications can access all kinds of resources available on the system.	Applets can only access browser-specific services. They don’t have access to the local system.
Installation	First and foremost, the installation of a Java application on the local computer is required.	The Java applet does not need to be installed beforehand.
Execution	Applications can execute the programs from the local system.	Applets cannot execute programs from the local machine.
Program	An application program is needed to perform some tasks directly for the user.	An applet program is needed to perform small tasks or part of them.
Run	It cannot run on its own; it needs JRE to execute.	It cannot start on its own, but it can be executed using a Java-enabled web browser.
Connection with servers	Connectivity with other servers is possible.	It is unable to connect to other servers.
Read and Write Operation	It supports the reading and writing of files on the local computer.	It does not support the reading and writing of files on the local computer.

Parameters	Java Application	Java Applet
Security	Application can access the system's data and resources without any security limitations.	Executed in a more restricted environment with tighter security. They can only use services that are exclusive to their browser.
Restrictions	Java applications are self-contained and require no additional security because they are trusted.	Applet programs cannot run on their own, necessitating the maximum level of security.

2.Sketch the MVC architecture and also explain each and every component in it.

MVC Architecture

The Model-View-Controller Architecture

Swing uses the *model-view-controller architecture* (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves.

Model

The model encompasses the state data for each component. There are different models for different types of components. For example, the model of a scrollbar component might contain information about the current position of its adjustable "thumb," its minimum and maximum values, and the thumb's width (relative to the range of values). A menu, on the other hand, may simply contain a list of the menu items the user can select from. This information remains the same no matter how the component is painted on the screen; model data is always independent of the component's visual representation.

View

The view refers to how you see the component on the screen. For a good example of how views can differ, look at an application window on two different GUI platforms. Almost all window frames have a title bar spanning the top of the window. However, the title bar may have a close box on the left side (like the Mac OS platform), or it may have the close box on the right side (as in the Windows platform). These are examples of different types of views for the same window object.

Controller

The controller is the portion of the user interface that dictates how the component interacts with events. Events come in many forms — e.g., a mouse click, gaining or losing focus, a keyboard event that triggers a specific menu command, or even a directive to repaint part of the screen. The controller decides how each component reacts to the event—if it reacts at all.

Figure 1-6 shows how the model, view, and controller work together to create a scrollbar component. The scrollbar uses the information in the model to determine how far into the scrollbar to render the thumb and how wide the thumb should be. Note that the model specifies this information relative to the minimum and the maximum. It does not give the position or width of the thumb in screen pixels—the view calculates that. The view determines exactly where and how to draw the scrollbar, given the proportions offered by the model. The view knows whether it is a horizontal or vertical scrollbar, and it knows exactly how to shadow the end buttons and the thumb. Finally, the controller is responsible for handling mouse events on the component. The controller knows, for example, that dragging the thumb is a legitimate action for a scrollbar, within the limits defined by the endpoints, and that pushing on the end buttons is acceptable as well. The result is a fully functional MVC scrollbar.

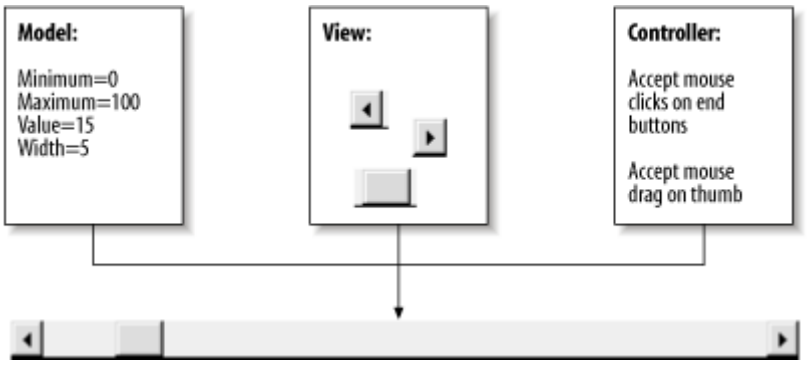


Figure 1-6. The three elements of a model-view-controller architecture

MVC Interaction

With MVC, each of the three elements—the model, the view, and the controller—requires the services of another element to keep itself continually updated. Let’s continue discussing the scrollbar component.

We already know that the view cannot render the scrollbar correctly without obtaining information from the model first. In this case, the scrollbar does not know where to draw its “thumb” unless it can obtain its current position and width relative to the minimum and maximum. Likewise, the view determines if the component is the recipient of user events, such as mouse clicks. (For example, the view knows the exact width of the thumb; it can tell whether a click occurred over the thumb or just outside of it.) The view passes these events on to the

controller, which decides how to handle them. Based on the controller's decisions, the values in the model may need to be altered. If the user drags the scrollbar thumb, the controller reacts by incrementing the thumb's position in the model. At that point, the whole cycle repeats. The three elements, therefore, communicate their data as shown in Figure 1-7.

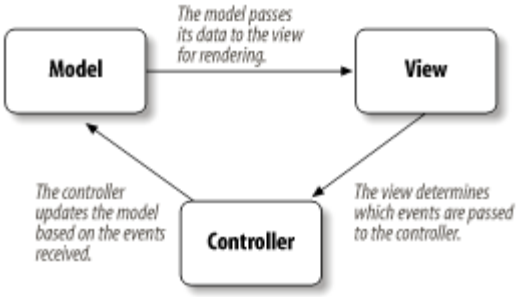


Figure 1-7. Communication through the model-view-controller architecture

MVC in Swing

Swing actually uses a simplified variant of the MVC design called the *model-delegate* . This design combines the view and the controller object into a single element, the *UI delegate* , which draws the component to the screen and handles GUI events. Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UI delegate then becomes a two-way street, as shown in Figure 1-8.

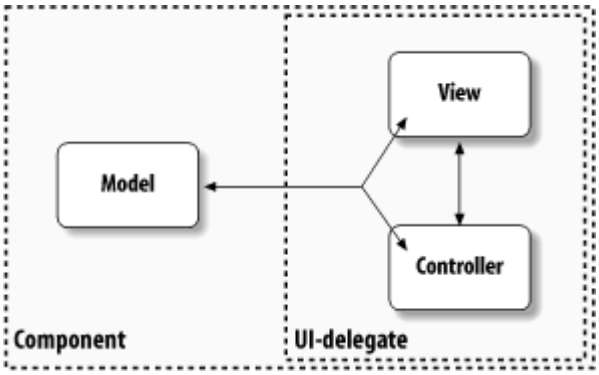


Figure 1-8. With Swing, the view and the controller are combined into a UI-delegate object

So let's review: each Swing component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate (in conjunction with AWT) reacts to various events that propagate through the component.

Note that the separation of the model and the UI delegate in the MVC design is extremely advantageous. One unique aspect of the MVC architecture is the ability to tie multiple views to a single model. For example, if you want to display the same data in a pie chart and in a table,

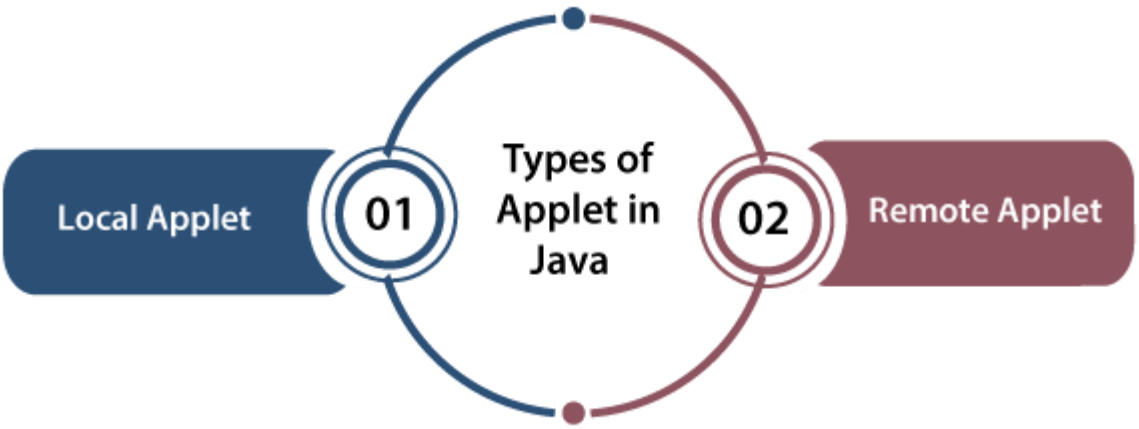
you can base the views of two components on a single data model. That way, if the data needs to be changed, you can do so in only one place—the views update themselves accordingly In the same manner, separating the delegate from the model gives the user the added benefit of choosing what a component looks like without affecting any of its data. By using this approach, in conjunction with the lightweight design, Swing can provide each component with its own pluggable Look&Feel.

3.What are the different types of applets and explain .

Types of Applets

A special type of Java program that runs in a Web browser is referred to as **Applet**. It has less response time because it works on the client-side. It is much secured executed by the browser under any of the platforms such as Windows, Linux and Mac OS etc. There are two types of applets that a web page can contain.

1. **Local Applet**
2. **Remote Applet**



Let's understand both types of Applet one by one:

Local Applet

Local Applet is written on our own, and then we will embed it into web pages. Local Applet is developed locally and stored in the local system. A web page doesn't need the get the information from the internet when it finds the local Applet in the system. It is specified or defined by the file name or pathname. There are two attributes used in defining an applet, i.e., the **codebase** that specifies the path name and **code** that defined the name of the file that contains Applet's code.

Specifying Local applet

```
<applet
   codebase = "G:\anu"
   code = "FaceApplet.class"
```

```
    width = 120
    height = 120>
</applet>
```

First, we will create a Local Applet for embedding in a web page.

After that, we will add that Local Applet to the web page.

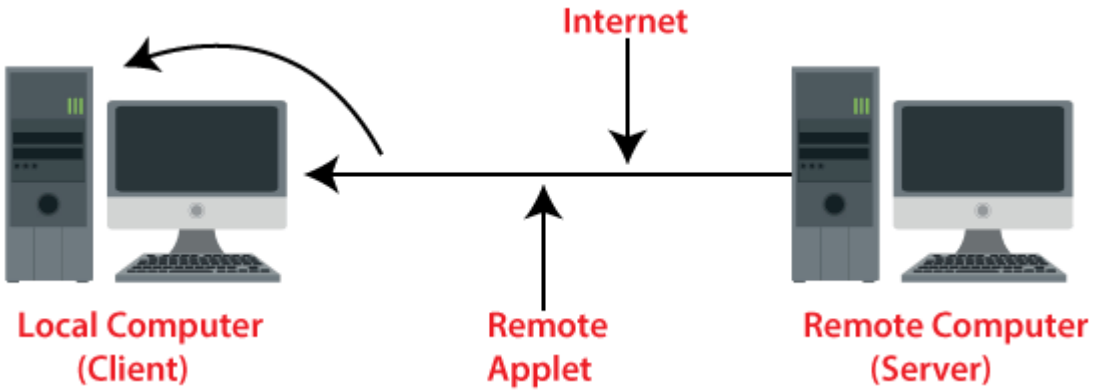
FaceApplet.java

```
//Import packages and classes
import java.applet.*;
import java.awt.*;
//Creating FaceApplet class that extends Applet
public class FaceApplet extends Applet
{
    //paint() method starts
    public void paint(Graphics g){
        //Creating graphical object
        g.setColor(Color.red);
        g.drawString("Welcome", 50, 50);

    }
}
```

Execute the above code by using the following commands:

Remote Applet

A remote applet is designed and developed by another developer. It is located or available on a remote computer that is connected to the internet. In order to run the applet stored in the remote computer, our system is connected to the internet then we can download run it. In order to locate and load a remote applet, we must know the applet's address on the web that is referred to as Uniform Recourse Locator(URL).



Specifying Remote applet

```
1. <applet
2.    codebase = "http://www.myconnect.com/applets/"
3.    code = "FaceApplet.class"
4.    width = 120
5.    height =120>
6. </applet>
```

4.Explain any 4 Swing components along with its methods.

1.ImageIcon

The class **ImageIcon** is an implementation of the Icon interface that paints Icons from Images.
Class Declaration

Following is the declaration for **javax.swing.ImageIcon** class −

public class ImageIcon
  extends Object
    implements Icon, Serializable, Accessible

An implementation of the Icon interface that paints Icons from Images. Images that are created from a URL, filename or byte array are preloaded using MediaTracker to monitor the loaded state of the image. For further information and examples of using image icons, see How to Use Icons in The Java Tutorial.
Class Constructors

Sr.No.	Constructor & Description
1	**ImageIcon()** Creates an uninitialized image icon.
2	**ImageIcon(byte[] imageData)**

	Creates an ImageIcon from an array of bytes which were read from an image file containing a supported image format, such as GIF, JPEG, or (as of 1.3) PNG.
3	**ImageIcon(byte[] imageData, String description)** Creates an ImageIcon from an array of bytes which were read from an image file containing a supported image format, such as GIF, JPEG, or (as of 1.3) PNG.
4	**ImageIcon(Image image)** Creates an ImageIcon from an image object.
5	**ImageIcon(Image image, String description)** Creates an ImageIcon from the image.
6	**ImageIcon(String filename)** Creates an ImageIcon from the specified file.
7	**ImageIcon(String filename, String description)** Creates an ImageIcon from the specified file.
8	**ImageIcon(URL location)** Creates an ImageIcon from the specified URL.
9	**ImageIcon(URL location, String description)** Creates an ImageIcon from the specified URL.

Class Methods

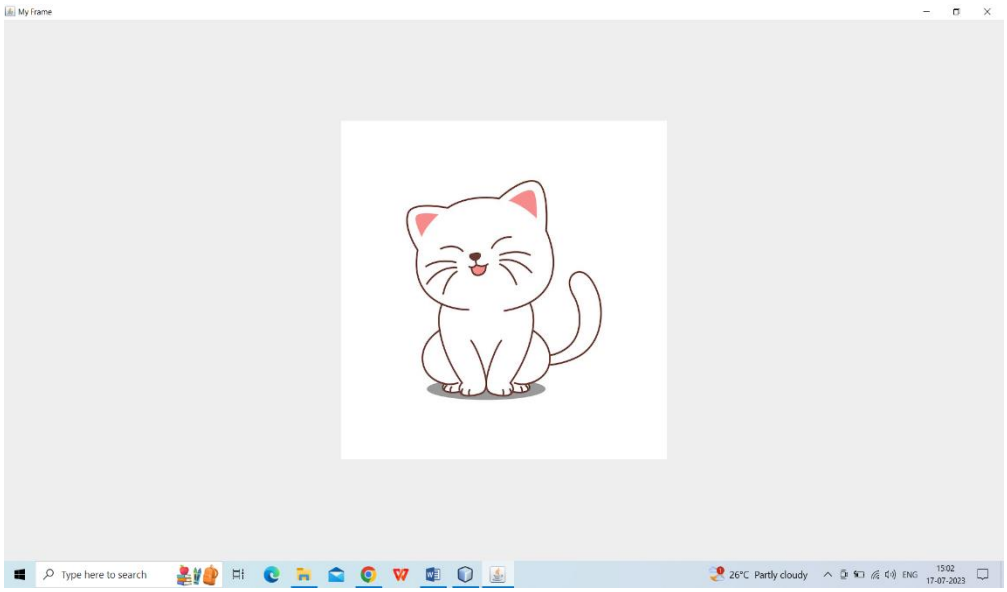
Sr.No.	Method & Description
1	**String getDescription()** Gets the description of the image.
2	**int getIconHeight()** Gets the height of the icon.
3	**int getIconWidth()** Gets the width of the icon.

4	**Image getImage()** Returns this icon's Image.
5	**void paintIcon(Component c, Graphics g, int x, int y)** Paints the icon.
6	**void setDescription(String description)** Sets the description of the image.
7	**void setImage(Image image)** Sets the image displayed by this icon.

Example program:-

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="ImageEx" width=250 height=300>
</applet>
*/
public class ImageEx extends JApplet
{

public void init()
{
ImageIcon icon = new ImageIcon("C:\\Users\\Admin\\Pictures\\pictures\\cat.jpg","Anusha");

        JLabel j = new JLabel(icon);
        add(j);

setLayout(new FlowLayout());
}
}
```

Output:-



2. JLabel

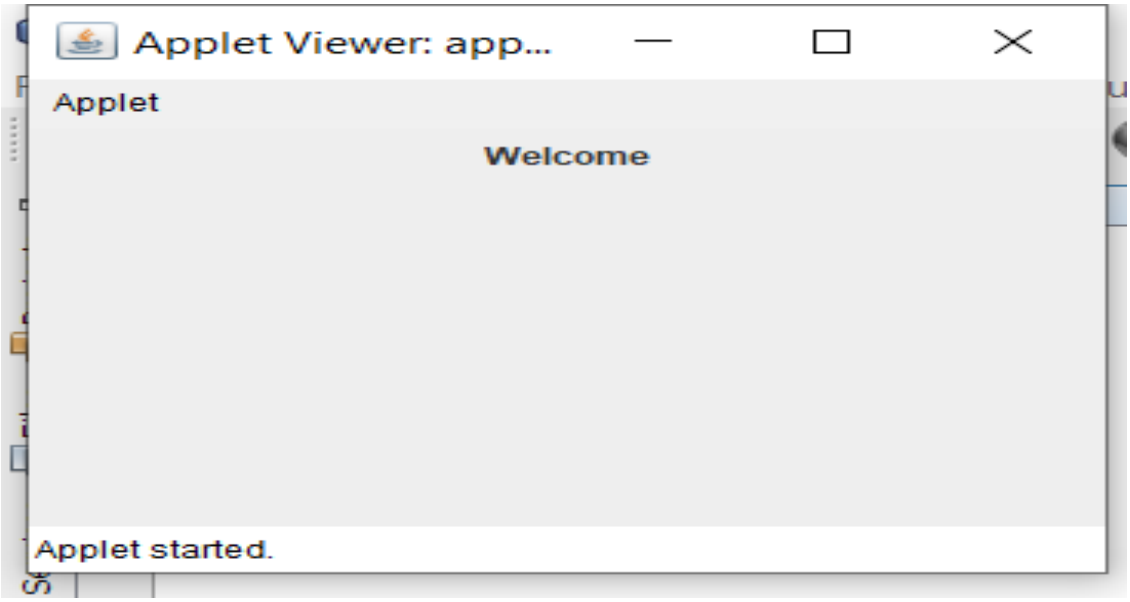
JLabel is a class of java Swing . JLabel is used to display a short string or an image icon. JLabel can display text, image or both . JLabel is only a display of text or image and it cannot get focus. JLabel is inactive to input events such a mouse focus or keyboard focus. By default labels are vertically centered but the user can change the alignment of label.
Fields:-
Public static final int LEFT
Public static final int RIGHT
Public static final int CENTER
Public static final int LEADING
Public static final int TRAILING

Constructor of the class are :
JLabel() : creates a blank label with no text or image in it.
JLabel(String s) : creates a new label with the string specified.
JLabel(Icon i) : creates a new label with a image on it.
JLabel(String s, Icon i, int align) : creates a new label with a string, an image and a specified horizontal alignment
Commonly used methods of the class are :
getIcon() : returns the image that the label displays
setIcon(Icon i) : sets the icon that the label will display to image i
getText() : returns the text that the label will display
setText(String s) : sets the text that the label will display to string s
Example program:-

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code=" LabelEx " width=250 height=300>
</applet>
*/
public class LabelEx extends JApplet
{
```

```
public void init()
{
JLabel b=new JLabel("Welcome");
add(b);
setLayout(new FlowLayout());
}
}
```

Output:-



3.JTextfield

The Swing text field is encapsulated by the JTextComponent class, which extends JComponent. It provides functionality that is common to Swing text components. One of its subclasses is JTextField, which allows you to edit one line of text. Some of its **constructors** are shown here:

JTextField( )
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)

Here, s is the string to be presented, and cols is the number of columns in the text field.

The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a JTextField object is created and is added to the content pane.

TextField 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.

Example:

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet
{
public void init()
{
setLayout(new FlowLayout());
JTextField jtf = new JTextField(10);
add(jtf);
}
}
```

Output:-

4.JButton Class

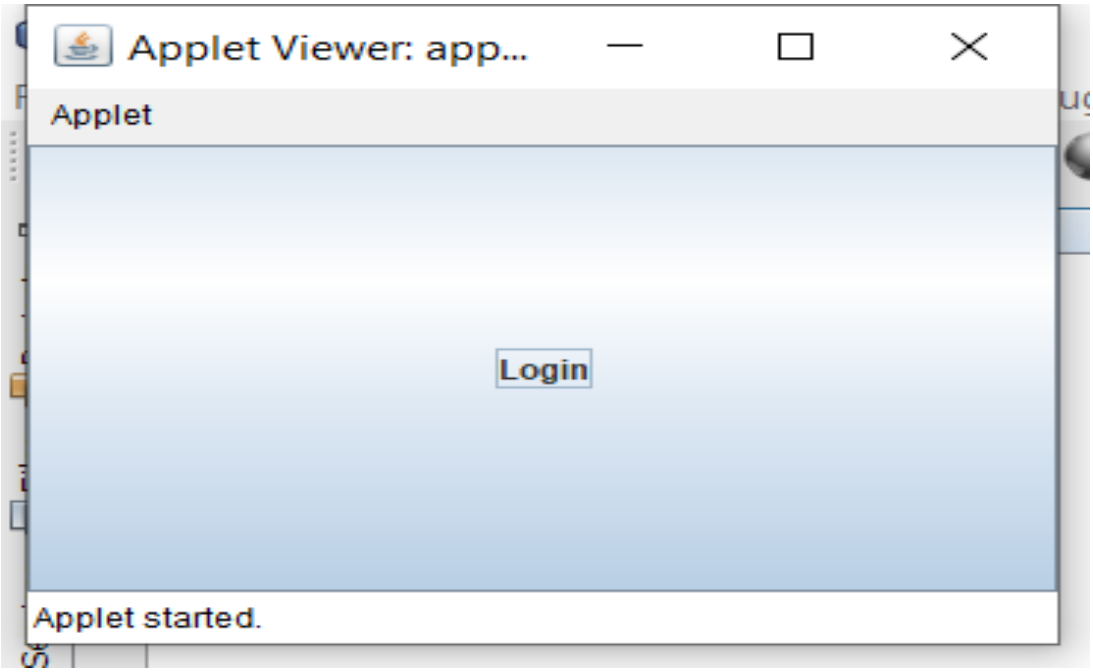
The JButton class provides the functionality of a push button. JButton allows an icon, a string, or both to be associated with the push button.
Some of its **constructors** are shown here:
JButton(Icon i)
JButton(String s)
JButton(String s, Icon i)
Here, s and i are the string and icon used for the button.
Button class Methods:

Methods	Description
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo2" width=250 height=300>
</applet>
*/
public class JButtonDemo2 extends JApplet
{

public void init()
{
JButton b=new JButton("Login");
add(b);
}
}
```

Output:-



5.Illustrate the life cycle methods of an applet with a suitable program.

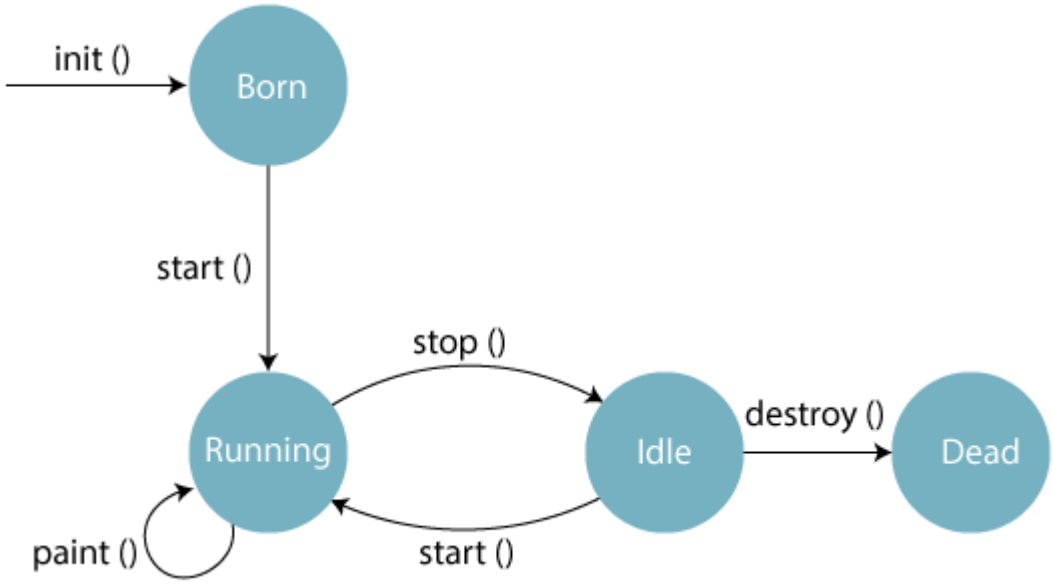
Life cycle of an applet

In Java, an applet is a special type of program embedded in the web page to generate dynamic content. Applet is a class in Java.

The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has five core methods namely init(), start(), stop(), paint() and destroy().These methods are invoked by the browser to execute.

Along with the browser, the applet also works on the client side, thus having less processing time.

Methods of Applet Life Cycle

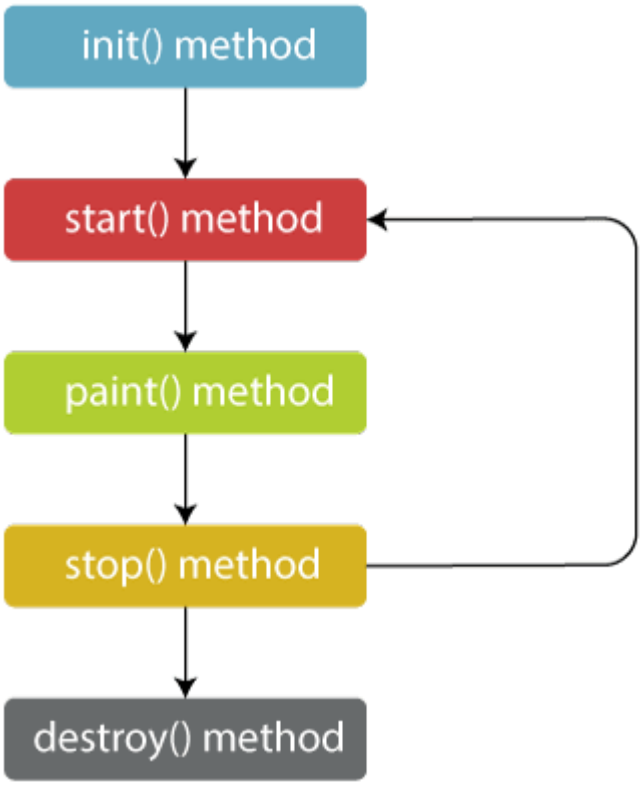


There are five methods of an applet life cycle, and they are:

- **init():** The init() method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the init() method within the applet.
- **start():** The start() method contains the actual code of the applet and starts the applet. It is invoked immediately after the init() method is invoked. Every time the browser is loaded or refreshed, the start() method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the init() method is invoked.
- **stop():** The stop() method stops the execution of the applet. The stop () method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the stop() method is invoked. When we go back to that page, the start() method is invoked again.
- **destroy():** The destroy() method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.
- **paint():** The paint() method belongs to the Graphics class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the start() method and when the browser or applet windows are resized.

Flow of Applet Life Cycle:

These methods are invoked by the browser automatically. There is no need to call them explicitly.



Syntax of entire Applet Life Cycle in Java

```
1. class TestAppletLifeCycle extends Applet {
2. public void init() {
3. // initialized objects
4. }
5. public void start() {
6. // code to start the applet
7. }
8. public void paint(Graphics graphics) {
9. // draw the shapes
10.}
11.public void stop() {
12.// code to stop the applet
13.}
14.public void destroy() {
15.// code to destroy the applet
16.}
17.}
```

6.Illustrate the ways of passing parameters to applets.

Passing parameters to applets

The APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the getParameter( ) method. It returns the value of the specified parameter in the form of a String object.

Param Tag

The <param> tag is a sub tag of the <applet> tag. The <param> tag contains two attributes: *name* and *value* which are used to specify the name of the parameter and the value of the parameter respectively. For example, the param tags for passing name and age parameters looks as shown below:

<param name=”name” value=”Ramesh” />
<param name=”age” value=”25″ />

Now, these two parameters can be accessed in the applet program using the *getParameter()* method of the *Applet* class.

getParameter() method:-

The *getParameter()* method of the *Applet* class can be used to retrieve the parameters passed from the HTML page. The syntax of *getParameter()* method is as follows:

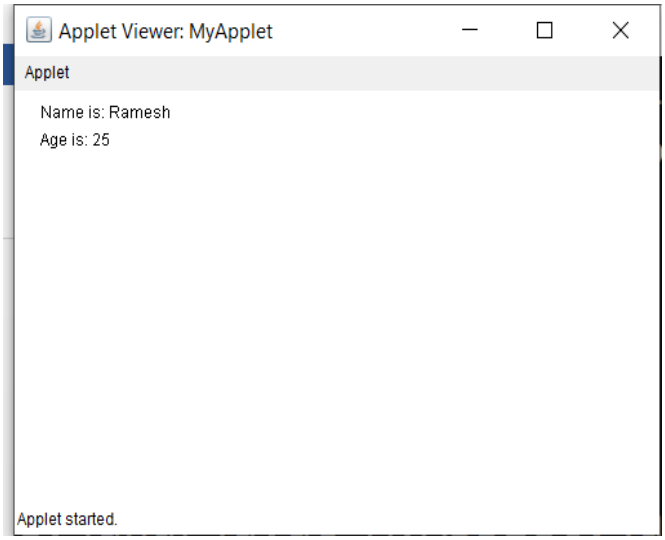
String getParameter(String param-name)

Let’s look at a sample program which demonstrates the <param> HTML tag and the *getParameter()* method:

```
import java.awt.*;
import java.applet.*;
public class MyApplet extends Applet
{
	String n;
	String a;
	public void init()
	{
		n = getParameter("name");
		a = getParameter("age");
	}
	public void paint(Graphics g)
	{
		g.drawString("Name is: " + n, 20, 20);
		g.drawString("Age is: " + a, 20, 40);
	}
}
```

```
/*
	<applet code="MyApplet" height="300" width="500">
		<param name="name" value="Ramesh" />
		<param name="age" value="25" />
	</applet>
*/
```

OutPut:-



7.Compare and contrast any 4 AWT and Swing components.

8.Swing provides platform-independent and lightweight components. Justify.