Module 4

Multi Threaded Programming and Event Handling

Multithreaded Programming :

- Multithreading is a conceptual programming paradigm where a program is divided into two or more subprogram, Which can be implemented at the same time in parallel.
- This is something similar to dividing a task into subtask and assigning them to processor for execution independently and simultaneously.
- Multithreading is a specialized form of multitasking. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- In a *thread-based* multitasking environment, the thread is the smallest unit of dispatch able code. This means that a single program can perform two or more tasks simultaneously.

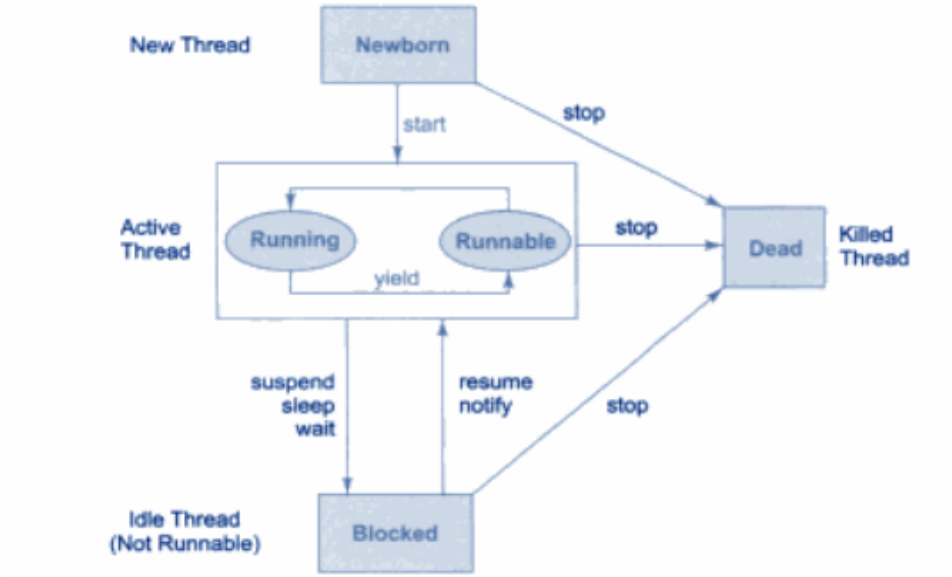
Thread:

- The small unit of program or sub module is called as thread.
- Each thread defines a separate path of execution
- Threads that do things like memory management and signal handling but from the application programmer's point of view, you start with just one thread, called the main thread.
- Main thread has the ability to create additional threads.

Life cycle of thread:

Thread can enter into different state during life of thread, different stages in thread are as follows:

- New born
- Runnable
- Running
- Dead
- Blocked



New Born state:

- When we create the thread class object, then thread is said to be born and move to new born state. But still thread is not under running state.
- Thread can be moved to either one of two state as follows:
- Thread can move from born state to dead state when we invoke stop() method.
- Thread can move from born state to runnable state when we invoke start() method.

Runnable state:

- Runnable state means thread is ready for execution and waiting for the processor to free.
- All thread are joined in queue and waiting for execution.
- If all the threads have equal priority,Then they are given atime slot for execution in round robin fashion ie. FCFS fashion.

Running state:

Running state means, Thread is under execution. Thread will run until its relinquish control on its own or its is preempted by the higher priority thread.

Blocked state:

- Thread is said to be blocked when it is preempted from entering into runnable state and subsequently the running state.
- This can be happen when thread is suspend, wait, sleep in order to satisfy certain requirements.

Dead State:

- Thread can be killing as soon as its born state by calling stop() method.
- Thread will automatically kill, as soon as its completed the operation

3.Creating thread:

Thread can be created in two ways:

- By creating a thread class(**Extending Thread class**)
- By converting a class to a Thread Class(**Implementing Runnable interface**)

1.By Creating a Thread class(Extending Thread class):

- Declare a class by extending Thread Class
- implement the run() method,Where run() method is the override method in order to implement the actual code to be executed by Thread.
- Create a thread object and call a start method to initiate the thread execution

Program:

```
class Thread1 extends Thread
{
        public void run()
        {
                for (int i=0;i<5;i++)
                {
                        System.out.println("Child thread");
                }
        }
}
```

```
class ThreadDemo
{
	public static void main(String[] args)
	{
		Thread1 t=new Thread1();
		t.start();

		for (int i=0;i<5;i++)
		{
			System.out.println("Main thread");
		}

	}
}
```

2.By converting a class to a Threadable Class(Implementing Runnable interface)

- Runnable interface declare the run() method that is required for implementing thread in our program. The following are the steps are taken to implement the runnable interface.
- Declare the class by implementing Runnable interface
- Implement the run() method.
- Crteate a thred by defining an object that is instantiated from this Runnabel class as the target of that thread
- Call the start() method to run the thread.

Program:

```
class Thread4 implements Runnable
{
	public void run()
	{
		for (int i=0;i<5;i++)
		{
			System.out.println("Runnable Child thread");
		}
	}
}

class RunnableDemo
{
	public static void main(String[] args)
	{
		Thread4 t4=new Thread4();
		Thread t=new Thread(t4);
```

```
                t.start();

                for (int i=0;i<5;i++)
                {
                        System.out.println("Main thread");
                }

        }
}
```

Threads methods:

The different threads methods are as follows:

Yield(),stop(),suspend(),resume(),wait(),notify(),notifyall().

1.yield()

Calling **yield()** will move the current thread from running to runnable, to give other threads a chance to execute. However the scheduler may still bring the same thread back to running when processor is free.

```
class Thread5 extends Thread
{
        public void run()
        {
                for (int i=0;i<5;i++)
                {

                        System.out.println("Child thread");
                        Thread.yield();
                }
                System.out.println("Status: " +isAlive());
        }
}

class ThreadYieldDemo
{
        public static void main(String[] args)
        {
                Thread5 t=new Thread5();
                t.start();

                for (int i=0;i<5;i++)
                {
```

```
                    System.out.println("Main thread yielded");
                }

            }
        }
```

Stop():

When stop() is called then processor will kill thread permentally.It means thread move to dead state.

```
class A extends Thread
{

public void run()
{
for(int i=0;i<10;i++)
{
if(i==2) stop();
}
}
}

class B
{
public static void main(String ar[])
{
A a1=new A();
a1.start();
}
}
```

Sleep():

- When sleep() is called then processor will stop the execution of thread for the specified amount of time from the execution.
- This static **sleep()** method causes the thread to suspend execution for a given time. The sleep method has two overloaded versions:
  - static void sleep (long milliseconds) throws InterruptedException
  - static void sleep (long milliseconds, int nanoseconds) throws InterruptedException

Multithreading Program using join() and sleep() methods

```
class Thread6 extends Thread
{
        public void run()
        {
                for (int i=0;i<5;i++)
                {
                        System.out.println("Child thread completes first");
                        try
                        {
                                Thread.sleep(2000);
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println(e);
                        }

                }
        }
}

class ThreadJoinDemo
{
        public static void main(String[] args)throws InterruptedException
        {
                Thread6 t=new Thread6();
                t.start();
                t.join();
                for (int i=0;i<5;i++)
                {System.out.println("Main thread");
                }

        }
}
```

Suspend():

- When suspend() is called then processor Sends the calling thread into block state.
- Using resume() method its bring the thread back from block state to running state.

Program for sleep and suspend resume methods:

```
class A extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
if(i==2) try { sleep(100);} catch(Exception e){ s.o.p(e) resume();}
}
}
}
class B extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
if(i==2) suspend();
}
}
}
class Mainclass
{
public static void main(String ar[])
{
A a1=new A();
B b1=new B();
a1.start();
b1.start();
}
}
```

Thread Priority:

- Each thread assigned a priority,which effects the order in which it is scheduled for running.
- Thread of same priority are given equal treatment by the java scheduler and there for they share the processor on FCFS basis
- Java permits us to set the priority of the thread using setPriority()methods.

Final void setPriority(int level)

- Where level specify the new priority setting for the calling thread. Level is the integer constant as follows**:**

  MAX_PRIORITY

  MIN_PRIORITY

  NORM_PRIORITY

- The MAX_priority value is 10,MIN_PRIORITY values is 1 And NORM_PRIORITY is the default priority whose value is 5.
- We can also obtain the current priority setting value by calling getPriority() method of thread.

  Final int getPriority()

Program:

```
class Thread1 extends Thread
{
        public void run()
        {
                for (int i=0;i<5;i++)
                {
                        System.out.println("Child thread");
                }
        }
}

class ThreadPri
{
        public static void main(String[] args)
        {
        //System.out.println(Thread.currentThread().getPriority());   // to get priority

                //Thread.currentThread().setPriority(15);  // error

                Thread.currentThread().setPriority(8);     // to set priority

                Thread1 t=new Thread1();
                t.start();

                System.out.println(t.getPriority());



        }
```

```
}
```

Synchronization:

- When two or more thread needs access to the shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization.
- Key to **synchronized** is the concepts of monitor or semaphores. A monitor is an object that is used as mutually exclusive lock or mutex. Only one thread can own a monitor at a given time. When one thread acquires a lock it is said to have entered the monitor.
- All other thread attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other thread are said to be waiting for monitor.
- This can be achieved by using keyword synchronized to method.

Syntex:

```
synchronized void method_name()
{  /* implementation or operation
}
```

Program:

```
class A
{
        synchronized void addNew(int i)
        {
                Thread t=Thread.currentThread();

                for(int n=1;n<5;n++)
                {
                        System.out.println(t.getName()+ ""+(i+n));
                }
        }
}

class B extends Thread
{
        A a1=new A();

        public void run()
        {
                a1.addNew(100);
```

```
        }
}


class SynchDemo
{
        public static void main(String args[])
        {
                B b=new B();

                Thread t1=new Thread(b);
                Thread t2=new Thread(b);

                t1.setName("T1: ");
                t2.setName("T2: ");

                t1.start();
                t2.start();
        }
}
```

Inter-thread communication:

- Inter-thread communication can be defined as exchange of message between two or more threads. The transfer of message takes place before or after changes of state of thread.
- The inter-thread communication can be achieved with the help of three methods as follows:

  wait(), notify() notifyall()

  wait()- tells the calling thread to give up the monitor and go to sleep mode until some of other thread enters the same monitor and call the notify() methods

  notify()-wakes up a thread that called wait() method on the same object.

  notifyall()-wakes up all thread that called wait() methods on the same object.
- Inter-thread communication can be implemented by using the key word as synchronized to methods.
- Different types of inter-thread communication example are as follows:
  - Producer-consumer problem
  - Reader –writer problem
  - Bounded-Buffer problem(Also called as Producer-consumer problem)

Producer-consumer problem/bounded buffer problem:

- Producer thread goes on producing an item unless an until buffer is full.
- Producer thread check before producessing an item weather buffer is full or not, if buffer is full producer wait producing an item unless and until consumer thread consume a item.
- Consumer thread goes on consuming an item which is produced by the producer. The consumer thread check weather buffer is empty before consuming. If buffer is empty consumer thread as to wait until producer has to produce an item.

Program:

```
class Queue
{
        int i;
        boolean produced=false;

        synchronized void Put(int n)
        {
                if(produced)
                {
                        try{
                                wait();
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
                }
                i=n;
                produced=true;
                System.out.println("Put: " +i);
                notify();
        }

        synchronized int Get()
        {
                if(!produced)
                {
                        try{
                                wait();
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
```

```
                }
                System.out.println("Got: " +i);
                produced=false;
                notify();
                return i;
        }

}

class Producer implements Runnable
{
        Queue q;

        Producer(Queue q)
        {
                this.q=q;
                new Thread(this,"Producer").start();

        }
        public void run()
        {
                int i=0;
                while(true)
                {
                        q.Put(i++);
                }
        }
}

class Consumer implements Runnable
{
        Queue q;

        Consumer(Queue q)
        {
                this.q=q;
                new Thread(this,"Producer").start();

        }


        public void run()
        {
                while(true)
                {
                        q.Get();
```

```
                }
        }
}

class PCdemo
{
        public static void main(String args[])
        {
                Queue q=new Queue();

                new Producer(q);
                new Consumer(q);

        }
}
```

Reader-writer problem:

- Reader thread reading an item from the buffer, Where as writer thread writing an item to buffer.
- If reader is reading then writer has to wait unless and until reading is finish.
- While writing thread writing an content then no other thread read the content unless and until writing is over.
- This problem can be achieved using wait, notify and nitifyall method and using synchronized keyword to method.

Program:

```
class A
{
static Int count=0;
synchronized void read()
{
while(count<0)
{
try
{
wait();
```

```
}
catch(Exception e)
{
System.out.println(e);
}
count--;
notify();
}
synchronized void write()
{
while(count>0)
{
try{
wait();
}
catch(Exception e)
{
System.out.println(e);
}
count++;
notify();
}
}
class Reader extends Thread
{
public void run()
{
A a1=new A();
System.out.println("reader thread");
for(int i=0;i<10;i++)
a1.read();
```

```
}
 }
class Writer extends Thread
{
public void run()
{
A a1=new A();
System.out.println("Writer thread");
for(int i=0;i<10;i++)
a1.read();
}
 }
class MainThread
{
public static void main(String a[])
{
Reader r=new Reader();
Writer w=new Writer();
r.start();
w.start();
}
}
```

isAlive()

- The final **isAlive()** method returns true if the thread is still running or the Thread has not terminated.

Program:

```
class t1 extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
```

```
                {
                System.out.println("Thread t1 :" +isAlive());
                }
        }
}

class IsAlive
{
        public static void main(String[] args)
        {
                t1 t11=new t1();

                t11.start();


                //System.out.println("Main Thread");

        }
}
```

Creating multiple thread:

- More than one thread can be created using single object of thread class. Where all the thread can execute parallel.

Program:

```
class Thread2 extends Thread
{
        public void run()
        {
                for (int i=0;i<5;i++)
                {
                        System.out.println("Child1 thread");
                }
        }
}


class Thread3 extends Thread
{
        public void run()
        {
                for (int i=0;i<5;i++)
                {
```

```
                        System.out.println("Child2 thread");
                }
        }
}

class ThreadDemo1
{
        public static void main(String[] args)
        {
                Thread2 t1=new Thread2();
                Thread3 t2=new Thread3();
                t1.start();
                t2.start();

                for (int i=0;i<5;i++)
                {
                        System.out.println("Main thread");
                }

        }
}
```

Event Handling

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. Event describes the change of state of any object. **Example:** Pressing a button, entering a character in Textbox.

Delegation event model :

- It defines standard and consistent mechanisms to generate and process events. Here the source generates an event and sends it to one or more listeners.
- The listener simply waits until it receives an event. Once it is obtained, It processes this event and returns.
- Listeners should register themselves with a source in order to receive an even notification. Notifications are sent only to listeners that want to receive them.

Components of Event Handling

Event handling has three main components,

Events :

- An event is a change of state of an object. In the delegation model, an *event* is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

Events Source :

- Event source is an object that generates an event.
- This occurs when the interval state of that object changes in some way.Source event generate more than one type of events.

Listeners :

A listener is an object that listens to the event. A listener gets notified when an event occurs.It has two major requirement

- **It must have been registred with one or more source to receive notification about specific types of event**
- **It must implement methods to receive and process these notification**

Event class:

The classes that represent events are at the core of Java's event handling mechanism.

EventObject : It is at the root of the Java event class hierarchy in **java.util.** It is the superclass for all events.

Its one constructor is shown here: EventObject(Object *src)*

Here, *src* is the object that generates this event. EventObject contains two methods: getSource( ) and toString( ). The getSource( ) method returns the source of the event.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.

1.ActionEvent Class

- An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or
- a menu item is selected.
- The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK,CTRL_MASK, META_MASK,** and **SHIFT_MASK.**
- **ActionEvent** has these three constructors:

  ActionEvent(Object *src,* int *type,* String *cmd)*

  ActionEvent(Object *src,* int *type,* String *cmd,* int *modifiers)*

ActionEvent(Object *src,* int *type,* String *cmd,* long *when,* int *modifiers)*

- Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type,* and its command string is *cmd.* The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred

2 **The AdjustmentEvent Class**

An **AdjustmentEvent** is generated by a scroll bar There are five types of adjustment events each defines integer constants that can be used to identify them

BLOCK_DECREMENT-the user clicked inside the scroll bar to decrese its value

BLOCK_INCREMENT- The user clicked inside the scroll bar to increses its value

TRACK-the slider was dragged

UNIT_DECREMENT-The button at the end of scroll bar was clicked to decreses its value

UNIT_INCREMENT-The button at the end of scroll bar was clicked to increses its value

Here is one AdjustmentEvent constructor

AdjustmentEvent(Adjustable src,int id,int type,int data);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type and its associated data is data.*

*The **getAdjustable()** method returns the object that generated the event.*

***getAdjustmentType()** method returns one of the constant defined by the AdjustmentEvent.*

3.ComponentEvents class:

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events .There are four integer constanst

COMPONENT_HIDDEN-the component was hidden

COMPONENT_MOVED-the component was moved

COMPONENT_RESIZED-the component was resized

COMPONENT_SHOWN-the component was shown.

There is one constructor

ComponentEvent(Component src,int type);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.*

getComponent() *method returns the component that was generated the event.*

4.ContainerEvent class:

A **ContainerEvent** is generated when a component is added to or removed from a container

Its has two integer constant:

COPONENT_ADDED-the component has been added to.

COMPONENT_REMOVED-The component has been removed out.

There is one constructor:

ContainerEvent(Component src,int type, component comp);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type and comp is the argument that indicates that component is added.*

getContainer() method generates the event.

getChild() method returns a reference to the component that was added or removed from the container.

5.ItemEvent classs:

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.there are two ineger constants:

DESELECTED-the user deselected an item

SELECTED-user selected an item

One constructor:

ItemEvent(itemSelectable src ,int type,object entry,int state);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.*

getItem()*method can be used to obtain a reference to the item that generated an event.*

getItemSelectable()-*method can be used to obtain a reference to the itemSelectable object that generated an event.*

getStateChange()-*method returns the state change for the event.*

6.KeyEvent class:

A **KeyEvent** is generated when keyboard input occurs.

There is one constructor:

KeyEvent(Component src,int type,long when,int modifier,int code,char ch);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.the system time at which key pressed,modifier argument indicates which modifier were pressed when key event generated.*

getChar() methods returns CHAR_UNDEFINED when a KEY_TYPED event occurs.

getKeyCode() method returns VK_UNDEFINED.

7.MouseEVENT class:

There are eight types of mouse event:

MOUSE_CLICKED-the user clicked the mouse

MOUSE_DRAGGED-the user dragged the mouse

MOUSE_ENTERED-the user entered the mouse

MOUSE_EXITED-the user exit the mouse

MOUSE_MOVED-the user moved the mouse

MOUSE_PRESSED-the user pressed the mouse

MOUSE_RELEASED-the user released the mouse

There is one constructor:

MouseEvent(Component src,int type,long when,int modifier,int x,int y,int click,boolen triggersPopup)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.the system time at which key pressed,modifier argument indicates which modifier were pressed when key event generated.*

8.TextEvent class:

The TextEvent Class Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.

There is one constructor:

TextEvent(Object src,int type);

Source Event:

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

Event Source	Description
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listeners Interface:

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus. (Added by Java 2, version 1.4)
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

1.ActionListener interface:

This interface define the actionPerformed() method that is invoked when an action event occurs.

void actionPerformed(ActionEvent ae);

2.AdjustmentListener interface:

This interface define the adjustmentValueChanged() method that is invoked when an adjustment event ocuurs.

void adjustmentValueChanged(AdjustmentEvent ae);

3.ComponentListener inetface:

This inetface define four methods that are invoked when a component is resized,moved,shown etc.

void componentResized(ComponentEvent ce);

void componentMoved(ComponentEvent ce);

void componentShown(ComponentEvent ce);

void componentHidden(ComponentEvent ce);

4.ItemListener inetface:

This interface define the itemStateChanged() method that is invoked when the state of an item changed.

void itemStateChanged(ItemEvent ie);

5.KeyListener interface:

This interface define three method,when key is ressed,released.

void keyPressed(KeyEvent ke);

void keyRelesed(KeyEvent ke);

void keyTyped(KeyEvent ke);

5.MouseListener interface:

This inetface define five methds

void mouseClicked(MouseEvent me);

void mouseEneterd(MouseEvent me);

void mouseExited(MouseEvent me);

void mousePressed(MouseEvent me);

void mouseReleased(MouseEvent me);

Program for handling keyboard events

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

public class Test extends Applet implements KeyListener
{
String msg="";
public void init()
{
addKeyListener(this);
}
```

```
public void keyPressed(KeyEvent k)
{
showStatus("KeyPressed");
}
public void keyReleased(KeyEvent k)
{
System.out.println("KeyRealesed");
}
public void keyTyped(KeyEvent k)
{
msg = msg+k.getKeyChar();
repaint();
}
public void paint(Graphics g)
{
g.drawString(msg, 20, 40);
}
}
```

Adapter class:

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

AWT Adapters:

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

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

Program :

```
class A extends Applet
{
public void init()
{
addMoseListener(new B(this));
}
}
class B extends MouseAdater
{
A  a1;
B( A a1)
{
this.a1=a1;
}
public void mouseClicked(MouseEvent me)
{
```

a1.showStatus(“mouse clicked”);

} }

Inner class/nested class:

- Class within other class is called nested class or inner class.
- Inner class is the member of outer class
- Outer class can access all the member of inner class, where as inner class cannot access the outer class member.

Program:

```
class A extends Applet
{
        public void inti()
        {
                addMoseListener(new B(this));
        }
        class B extends MouseAdater
        {
                public void mouseClicked(MouseEvent me)
                {
                        a1.showStatus(“mouse clicked”);
                }
        }
}
```