Multithreaded programming
UNIT 4

*) Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently.

*) Each part of such a program is called a thread, and each thread defines a separate path of execution.

*) There are two distinct types of multitasking: process based and thread-based.

*) A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently.

*) For example, process-based multitasking enables us to run the Java compiler at the same time that we are using a text editor.

*) 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 dispatchable code. This means that a single program can perform two or more tasks simultaneously.

*) For instance, a ~~text~~ text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Note: process - based

*) Each process have its own address in memory i.e each process allocates separate memory area.

*) process is heavy weight.

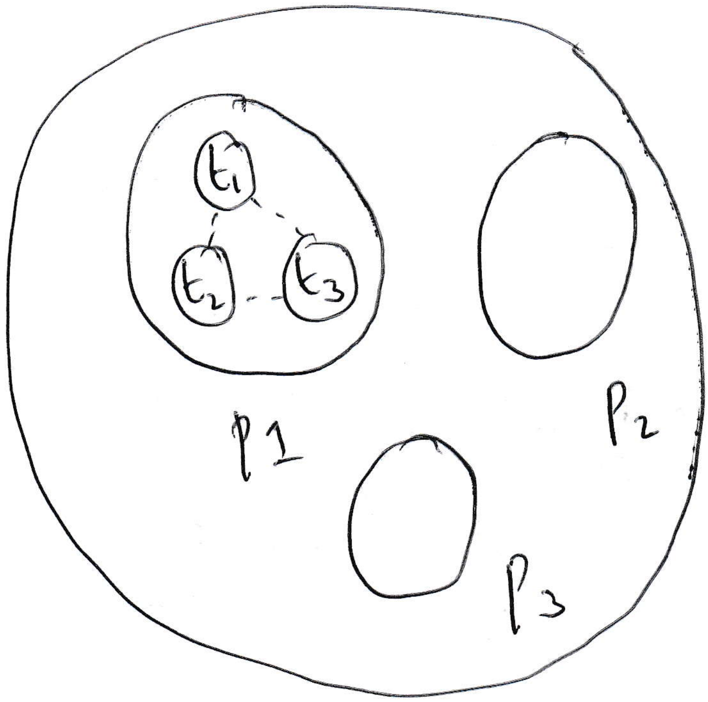
*) cost of communication between the process is high (switching from one process to another requires some time for saving & loading registers, memory maps, updating lists etc)

*) Thread - Based

*) Threads share the same address space.

*) Thread is lightweight

(threads within the process share the same address space, code section, data section, OS resources & communications between threads is low so) It is light weight



*) cost of communication b/w the thread is low, so we always prefer more multithreading than multiprocessing.

→ The Java Thread Model:

*) The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.

*) In fact, Java uses threads to enable the entire environment to be asynchronous.

*) Single threaded systems use an approach called an event loop with polling.

*) polling is a single event queue to decide what to do next. Once this polling mechanism returns with, say a file is ready to be read, then the event loop dispatches control to the event handler.

polling (single event queue) event loop $\xrightarrow[\text{control}]{\text{dispatches}}$ event handler.

*) until the event handler returns, nothing else can happen in the system. This wastes CPU time.

*) when a thread blocks because it is waiting for some resources, the entire program stops running.

*) The benefit of Java's multithreading is that the polling mechanism is eliminated.

*) one thread can pause without stopping other parts of our programs. for example, the idle time created when a thread reads data from a N/w or waits for user i/p can be utilized somewhere else.

Threads exist in several states:

*) A thread can be running.

*) A running thread can be suspended.

*) A suspended thread can then be resumed.

*) A thread can be blocked

*) A thread can be terminated

*) A thread can be resumed.

→ Thread Priorities

*) Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.

*) The thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

*) The rules that determine when a context switch takes place are simple:

– A thread can voluntarily claim control. This is done by explicitly yielding, sleeping or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the cpu.

*) A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted.

*) In cases where two threads with the same priority are competing for cpu cycles, the situation is a bit complicated. For operating systems such as windows, threads of equal priority are time-sliced automatically in round-robin fashion.

→ Synchronization

*) Because multithreading introduces an asynchronous behaviour to our programs, there must be a way to enforce synchronicity when you need it.

*) For example, if you want two threads to communicate and share a complicated data structure, such as linked list, you need some way to ensure that they don't conflict with eachother. i.e we must prevent one thread from writing data while another is in the middle of reading.

*) For this purpose, java implements an elegant twist on an age-old model of interprocess synchronization: the monitor

*) A monitor is a very small box that can hold only one thread. once a thread enters a monitor, all other threads must wait until that thread exits the monitor.

*) In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

→ Messaging

*) Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have.

*) Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

→ The Thread Class and the Runnable Interface.

*) Java multithreading system is built upon the Thread class, its methods, Runnable.

*) The Thread class defines several methods that help manage threads.

Method	Meaning.
getName	obtains a thread's name.
getPriority	obtains a thread's priority
isAlive	Determine if a thread is still running.
join	wait for a thread to terminate.
run	Entry point for the thread
sleep	suspend a thread for a period of time.
start	start a thread by calling its run method.

Thread Priorities:

*) Thread priorities are used by the thread schedular to decide when each thread should be allowed to run. In theory, higher priority threads get more cpu time than lower-priority threads.

*) In theory, threads of equal priority should get equal access to the cpu. But you need to be careful. Remember, Java is designed to work in a wide range of environment. Some of those environments implement multitasking differently than others.

*) To set a thread's priority, use the setpriority() method, which is a member of thread

general form: final void setPriority(int level)

*) The value of level must be within the range MIN_PRIORITY & MAX_PRIORITY. currently, these values are 1 and 10 respectively.

To return a thread to default priority, specify NORM_PRIORITY, which is currently 5.

These priorities are defined as final variables within Thread.

```
// Demonstrate thread priorities.

Class clicker implements Runnable
{
    int Click=0;
    Thread t;
    Private volatile boolean running = true;
    public clicker(int p)
    {
        t = new Thread(this);
        t.setpriority(p);
    }

public void run()
{
    while (running)
    {
        Click++;
    }
}

Public Void stop()
{
    running = false;
}
```

```
public void start()
{
  t.start();
}
}

class HiLoPri
{
 public static void main(string args[])
{
 Thread.currentThread().setpriority(Thread.MAX_PRIORITY);
 Clicker hi = new clicker(Thread.NORM_PRIORITY +2);
 Clicker lo = new clicker(Thread.NORM_PRIORITY -2);
 lo.start();
 hi.start();
try
{
 Thread.sleep(10000);
}
catch(InterruptedException e)
{
 System.out.println("Main thread interrupted");
}
lo.stop();
hi.stop();
// wait for child threads to terminate
try{
   hi.t.join();
   lo.t.join();
}
catch(InterruptException e)
{
S.o.p("Interrupted Exception");
}
S.o.p("low-priority: "
                 +lo.click);
S.o.p("high-priority: "
                 +hi.click);
}
}
```

o/p low.priority: 4408112
high.priority: 589626904

Thread priorities

```
class TestMultiPriority1 extends Thread
{
    public void run()
    {
        s.o.pln("running thread"+ Thread.currentThread().getName());

        s.o.pln("its priority is"+ Thread.currentThread().getPriority());
    }

    public static void main(String args[])
    {
        TestMultiPriority1 m1 = new TestMultiPriority1();
        TestMultiPriority1 m2 = new TestMultiPriority1();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

o/p running thread thread0

Its priority is 10

running thread thread1

Its priority is 10

→ Synchronization

*) When two or more threads need access to a shared resources, 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 synchronization is the concept of the monitor. A monitor is an object that is used as a mutually exclusive lock or mutex. Only one thread can own a monitor at a given time.

*) When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

```
// This program is not synchronized.
class callme
{
    void call(string msg)
    {
        system.out.print( "[" + msg);
        try {
```

```
Thread.sleep(1000);
}
catch(InterruptedException e)
{
  System.out.println("Interrupted");
}
  System.out.println("]");
}
}

Class Caller implements Runnable
{
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s)
  {
    target=targ;
    msg=s;
    t=new Thread(this);
    t.start();
  }
  public void run()
  {
    target.call(msg);
  }
}
```

```
public void run()
{
// to synchronize
synchronized(target)
{
  target.call(msg);
}
}
```

```
class synch
{
 public static void main(String args[])
 {
  callme target = new callme();
  caller ob1 = new caller(target, "Hello");
  caller ob2 = new caller(target, "synchronized");
  caller ob3 = new caller(target, "world");
  // wait for threads to end.
   try {
     ob1.t.join();
     ob2.t.join();        o/p [Hello [Synchronized [
     ob3.t.join();                     world]]]
   }
  catch (InterruptedException e)
   {
     S.o.p("Interrupted");
   }
 }
}
```

*) To fix the preceding program, we must serialize access to call(). That is, we must restrict its access to only one thread at a time.

*) To do this, we simply need to precede call() definition with the keyword synchronized as shown here:

```
class callme
{
  synchronized void call(string msg)
  {
      ...
  }
```

o/p [Hello]
[Synchronized]
[world]

→ Interthread Communication

*) polling is usually implemented by a loop that is used to check some condition repeatedly.

*) once the condition is true, appropriate action is taken. This wastes cpu time. for example, consider the classic queuing problem,

*) where one thread is producing some data and another is consuming it, To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.

*) In polling system, the consumer would waste many cpu cycles while it waited for the producer to produce.

```
class callme
{
  synchronized void call(string msg)
  {
     ...
  }
```

o/p [Hello]
[synchronized]
[world]

→ Interthread communication

*) polling is usually implemented by a loop that is used to check some condition repeatedly.

*) once the condition is true, appropriate action is taken. This wastes cpu time. for example, consider the classic queuing problem,

*) where one thread is producing some data and another is consuming it, To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.

*) In polling system, the consumer would waste many cpu cycles while it waited for the producer to produce.

once the producer was finished, it would start polling, wasting more cpu cycles waiting for the consumer to finish, so on.

*) To avoid polling, Java includes an interprocess communication mechanism via the wait(), notify() and notifyAll() methods.

1) wait(): tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().

calling thread
↓ to give up
monitor
↓ go to sleep
until some other thread enters monitor (notify())

2) notify(): wakes up the first thread that called wait() on same object.

3) notifyAll(): wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

// incorrect implementation of a producer & consumer.

```
class Q
{
 int n;
 synchronized int get()
 {
   System.out.println("got:" +n);
   return n;
 }
```

```
synchronized void put (int n)
 {
    this.n=n;
    system.out.println("put:"+n);
 }
}
```

```
class producer implements Runnable
{
  Q q;
  producer(Q q)
  {
    this.q =q;
    new Thread (this,"producer").start();
  }
  public void run()
  {
    int i=0;
    while (true)
    {
      q.put(i++);
    }
  }
}
```

```
class Consumer implements Runnable
{
  Q q;
  consumer (Q q)
  {
    this.q=q;
    new Thread (this, "consumer").start();
```

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

```
class PC
{
  public static void main(String args[])
  {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Press Control-C to stop");
  }
}
```


*) Although the put() & get() methods on Q are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue (values will vary with processor speed & load)

```
Put: 1        Put: 2
Got: 1          Put: 3
Got: 1            Put: 4
Got: 1            Put: 5
Got: 1            Put: 6
                  Put: 7
                  Got: 7
```

*)

1) The proper way to write the program in java is to use wait() and notify() to signal in both directions, as shown.

```
// A correct implementation of a producer & consumer.

Class Q
{
    int n;
    boolean valueset=false;

   Synchronized int get()
   {
      if(!Valueset)
      {
        try{
              wait();
           }
        catch (InterruptException e)
        {
          System.out.println("Interrupt caught");
        }
      System.out.println("Got:" +n);
      valueset=false;
      notify();  // wakes up the first thread that calls wait.
      return n;
      }
```

```
Synchronized void put(int n)
{
    if(valueset)
    {
        try{
            wait();
        }
        catch(InterruptedException e)
        {
            S.o.p("Interrupt caught");
        }
    }
    this.n = n;
    valueset = true;
    S.o.p("put:" +n);
    notify();
}
}
```

```
Class producer implements Runnable
{ Q q;
  producer(Q q)
  { this.q = q;
    new thread(this, "producer").start();
  }

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

```
class consumer implements Runnable
{
  Q q;
  consumer (Q q)
  {
    this.q = q;
    new Thread(this, "consumer").start();
  }
  public void run()
  {
    while(true)
    {
      q.get();
    }
  }
}

class PCFixed()
{
  public static void main (String args[])
  {
    Q q = new Q();
    new producer (q);
    new Consumer (q);
    S.o.p("press control c to stop");
  }
}
```

O/p

```
put : 1
got : 1
put : 2
got : 2
put : 3
got : 3
```

Event handling

*) Any program that uses a graphical user interface, such as java application written for windows, is event driven.

*) Events are supported by a number of packages, including java.util, java.awt and java.awt.event.

*) Most events to which your program will respond are generated when the user interacts with a GUI-based program.

*) There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar or check box.

*) This chapter begins with an overview of Java's event handling. It then examines the main event classes and interfaces used by AWT and demonstrates several fundamentals of event handling.

(ABSTRACT WINDOW TOOLKIT) → provides the interface to create GUI such as buttons.

→ Two Event Handling Mechanisms.

*) The way in which events are handled changed significantly between the original version of Java (1.0) and modern version of Java, beginning with Version 1.1

*) The 1.0 method of event handling is still supported, but it is not recommended for new programs. The modern approach is the way that events should be handled by all new programs and thus is the method employed by programs.

→ The Delegation Event Model

Note:
1) Register the Component with the Listner

Syntax: add~~Action~~TypeListner(~~Action~~TypeListner a);

2) If user performs any event like clicking the mouse button, scrolling etc, the object for ~~concerned~~ Concerned event class is created automatically and information about the source & the event get populated.

3) Event object is forwarded to the method of Registered Listener class.

4) method is now gets executed and returned.

*)

→ The Main Thread

```
Class CurrentThreadDemo
{
  public static void main(String args[])
  {
      Thread t = Thread.currentThread();
      System.out.println("current thread :" +t);
    // change the name of the thread
     t.setName("My Thread");
    System.out.println(" After name change " +t);
    try
    {
      for(int i=5; i>0; i--)
      {
         System.out.println(i);
         Thread.sleep(1000);
      }
      }
    Catch(InterruptedException e)
    {
      System.out.println("MAIN THREAD INTERRUPTED ");
    }
  }
}
```

*) when a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program. This is the one that is executed when your program begins.

A) The main thread ~~from~~ is important for two reasons:

1) It is the thread from which other "child" threads will be spawned.

2) Often, it must be the last thread to finish execution because it performs various shutdown actions.

*) main thread can be automatically created when your program is started, it can be controlled through a Thread object. To do so, we must obtain a reference to it by calling the method currentThread().

general form is: static Thread currentThread()

*) o/p of the program:

Current thread: Thread [main, 5, main]

After name change: Thread [my Thread, 5, main]

5
4
3
2
1

Notice the output produced when t is used as an argument to println(). This displays, in order, the name of the thread, its priority, and the name of its group.

*) By default, the name of the main thread is main. Its priority is 5, which is the default value, and main is also the name of the group of threads to which this thread belong.

*) The sleep() method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

general form: static void sleep(long milliseconds) throws InterruptedException.

→ <u>Creating a Thread</u>.

*) Java defines two ways in which this can be accomplished.

1) we can implement the Runnable interface.

2) we can extend the Thread class, itself.

Implementing Runnable:

*) The easiest way to create a thread is to create a class that implements the Runnable interface.

*) Runnable abstracts a unit of executable code. We can construct a thread on any objects that implements Runnable

*) To implement Runnable, a class need only implement a single method called run()

```
public void run()
```

*) Inside run(), we will define the code that constitutes the new thread.

*) After you create a class that implements Runnable, we will instantiate an object of type Thread from within that class.

*) Thread defines several constructors, The one that we will use is shown here:

```
Thread(Runnable threadob, string threadName)
```

*) In this constructor, threadob is an instance of a class that implements the Runnable interface.
This defines where execution of the thread will begin.
The name of the new thread is specified by threadName.

*) After the new thread is created, it will not start running until you call its ~~Star~~ start() method.

```
void start()
```

*) Example that creates a new thread and starts its running:


```
class NewThread implements Runnable
{
  Thread t;
  NewThread()
  {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("child thread" + t);
    t.start(); // start the thread
  }
```

```
//This is the entry point for the second thread.
  public void run()
   {
     try
       {
        for(int i=5; i>0; i--)
         {
           system.out.println("child thread: "+i);
           Thread.sleep(500);
         }
       }
     catch(InterruptedException e)
      {
       system.out.println("child Interrupted.");
      }
     system.out.println("Exiting child thread");
   }
}

Class ThreadDemo
 {
   public static void main(string args[])
    {
    new NewThread(); // create a new thread
    (or)
    NewThread R = new NewThread();
```

```
try
{
  for(int i=5; i>0; i--)
  {
   System.out.println("main thread: " + i);
   Thread.sleep(1000);
  }
 }
 catch(InterruptedException e)
 {
 System.out.println("main thread interrupted");
 }
 System.out.println("main thread exiting");
}
}
```

Note: Inside NewThread's Constructor, a new Thread object is Created by t = new Thread(this, "Demo Thread");

O/p : Child thread: Thread [Demo thread, 5, main]

main thread : 5	Main thread : 3
Child thread : 5	Child thread : 1
child thread : 4	Exiting child thread
Main thread : 4	Main thread : 2
Child thread : 3	: 1
" : 2	Main thread exiting

→ Extending Thread

*) The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

*) The extending class must override the run() method which is the entry point for the new thread. It must also call start() to begin execution of the new thread.

```
// Create a second thread by extending Thread.
class NewThread extends Thread
{
  NewThread()              // Create a new, second thread.
  {
    super("Demo Thread");
    system.out.println("child thread " + this);
    Start();
  }
  // This is the entry point for the second thread.
  public void run()
  {
    try
    {
      for (int i=5; i>0; i--)
      {
```

```
try
{
  for(int i=5; i>0; i--)
  {
    system.out.println("Child Thread"+i);
    Thread.sleep(500);
  }
} catch(InterruptedException e)
{
  system.out.println("child interrupted");
}
system.out.println("Exiting child thread");
}
}
```



```
class ExtendThread
{
  public static void main (string args[])
  {
    new NewThread(); //Creates a new thread
try
{
    for(int i=5; i>0; i--)
    {
      system.out.println("mainThread : "+i);
      Thread.sleep(1000);
    }
} 
    catch(InterruptedException e)
    {
     system.out.println("main thread Interrupted");
    }
    System.out.println("Main thread exiting");
```

Choosing an Approach

*) which approach is better. The Thread Class defines several methods that can be overridden by a derived Class, of these method the only one that must be overridden is run().

*) The same method required when you implement Runnable. many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. so if you will not be overriding any of threads other methods.

→ Creating multiple Threads

*) So far, we have been using only two threads: the main thread and one child thread. However your program can spawn as many threads as it needs.

*) The following program creates three child threads:

```
// create multiple threads
class NewThread implements Runnable
{
    string name; // name of thread
    Thread t;
  NewThread(string threadname)
  {
    name = threadname;
    t = new Thread(this, name);
    system.out.println("New thread" + t);
    t.start(); // start the thread.
  }
// This is the entry point for thread.
public void run()
{
    try{
        for(int i=5; i>0; i--)
        {
          system.out.println(name + ":" + i);
          Thread.sleep(1000);
        }
    }
        catch(InterruptedException e)
        {
          system.out.println(name + "Interrupted");
        }
      system.out.println(name + "exiting");
} }
```

```
Class MultiThreadDemo
{
  public static void main (String args[])
  {
    new NewThread ("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");
  try
  {
    //wait for other threads to end
    Thread.sleep(10000);
  }
  catch(InterruptedException e)
  {
    System.out.println("Main thread exiting");
  }
}
```


O/p:

```
New thread: Thread(One,5,main)
New thread: Thread(Two,5,main)
New thread: Thread(Three,5,main)
One : 5
Two : 5
Three : 5
One : 4
Two : 4
Three : 4
One : 3
Two : 3
Three : 3
One : 2
Two : 2
Three : 2
One : 1
Two : 1
Three : 1
One exiting
Two exiting
Three exiting
Main thread exiting.
```

→ Using isAlive() and join()

*) Two ways exist to determine whether a thread has finished. first, we can call isAlive() on the thread. This method is defined by Thread.

general form: final boolean isAlive();

*) The isAlive() method returns true if the thread upon which it is called is still running. It returns false otherwise.

*) while isAlive() is occasionally useful, the method that we will more commonly use to wait for a thread to finish is called join().

final void join() throws InterruptedException.

*) This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it.

[Join - waits for the thread to terminate]

//using join() to wait for threads to finish.

```
class NewThread implements Runnable
{
    String name;
    Thread t;

    NewThread (String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New Thread :" +t);
        t.start(); //start the thread.
    }

    public void run()
    {
        try
        {
            for(int i=5; i>0; i--)
            {
            System.out.println(name + ":" + i);
            Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
        System.out.println(name + "interrupted");
        }
```

```
System.out.println(name + "exiting");
}
}

class DemoJoin
{
  public static void main(String args[])
  {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");
System.out.println("Thread one is alive " + ob1.t.isAlive());
System.out.println("Thread two is alive " + ob2.t.isAlive());
System.out.println("Thread three is alive " + ob3.t.isAlive());
try   // wait for thread to finish.
  {
    System.out.println("waiting for thread to finish");
    ob1.t.join();
    ob2.t.join();
    ob3.t.join();
  }
catch (InterruptedException e)
{
  System.out.println("Main thread Interrupted");
}}
```

Interface → Abstract class

Int class Example

Interface {

void compute(); → Abstract method

void read()
{
}

}

System.out.println("Thread one is alive: " + ob1.t.isAlive());
System.out.println("Thread two is alive " + ob2.t.isAlive());
System.out.println("Thread three is alive " + ob3.t.isAlive());

System.out.println("Main thread exiting");

}
}

o/p

New thread: Thread(one,5,main)
New thread: Thread (Two,5,main)
New thread: Thread (Three,5,main)

Thread one is alive: true
" two is alive : true
" three is alive : true

waiting for threads to finish

One: 5
Two: 5
Three: 5

One: 4
Two: 4
Three: 4

⋮

One: 1
Two: 1
Three: 1

One exiting
Two exiting
Three exiting.

Thread one is alive : false

Thread two is alive : false

Thread three: false

Main thread exiting.

→ The ActionEvent class

*) An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.

*) ActionEvent class defines four integer constants that can be used to identify any modifiers associated with action event.

ALT_MASK, CTRL_MASK, META_MASK, SHIFT_MASK.

*) ActionEvent has these two constructors:

ActionEvent(Object src, int type, String cmd)

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

src → Reference to the object that generated this event.

Type → type of the event.

cmd → command string.

modifiers → indicates which modifier keys (ALT, CTRL, SHIFT) were pressed when an event was generated.

*) We can also obtain the command name for the invoking, ActionEvent object by using getActionCommand().

*) The getModifiers() method returns a value that indicates which modifier keys were pressed when an event has generated.

→ The AdjustmentEvent class.

*) An AdjustmentEvent is generated by a scroll bar. There are five types of adjustment events.

*) The AdjustmentEvent class defines integer constants that can be used to identify them.

*) The constants and their meanings are: (ID)

BLOCK_DECREMENT → The user clicked inside the scroll bar to decrement (decrease) its value.

BLOCK_INCREMENT → The user clicked inside the scroll bar to increase its value

TRACK → The slider was dragged.

UNIT_DECREMENT → The button at the end of the scroll bar was clicked to decrease its value

UNIT_INCREMENT → The button at the top of the scroll bar was clicked to increase its value.

*) AdjustmentEvent has this constructor:

AdjustmentEvent (Adjustable src, int id, int type, int data) → How much scroll down

→ adjustment event

Here src → is a reference to the object that generated this event.

Note: There is ADJUSTMENT_VALUE_CHANGED, that indicates that a change has occured, which is an integer constant.

id → id equals ADJUSTMENT_VALUE_CHANGED.

type → type of event & associated data.
& data

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

*) getAdjustmentType() → Type of the event is obtained.

*) getValue () → amount of Adjustment is obtained.

→ The ComponentEvent class

Component Event (superclass) → mouse, keyboard, window

*) ComponentEvent is generated when the size, position or visibility of a component is changed.

*) There are four types of component events.

COMPONENT_HIDDEN → The component was hidden.
COMPONENT_MOVED → The component was moved
COMPONENT_RESIZED → The component was resized
COMPONENT_SHOWN → The component became visible.

*) ComponentEvent Constructor is

ComponentEvent(Component src, int type)

*) ComponentEvent is the superclass of ContainerEvent, FocusEvent, KeyEvent, MouseEvent & WindowEvent.

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

Component getComponent()

→ The ContainerEvent class

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

*) There are two types of container events. The ContainerEvent class defines int constants that can be used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED.

*) Constructor defined as follows: → button / list panels → component add / remove → which contains ?

ContainerEvent (Component src, int type, Component comp)

comp → component that is added or removed from the container

src → reference to the container which generates this event.

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

*) we can obtain a reference to the container that generated this event by using getContainer() method.

Note :- components include lists, buttons, panels & windows. To use components, we need to place them in container. container is a component that holds & manages other components.

→ The FocusEvent class: (permanent / temporary focus)

*) A focus event is generated when a component gains or loses input focus.

*) These events are identified by the integer constants FOCUS_GAINED & FOCUS_LOST

*) FocusEvent is a subclass of ComponentEvent and has these constructors.

focus gain / focus lost

b/w default = true / false

permanent focus

FocusEvent(component src, int type)

FocusEvent(Component src, int type, boolean temporaryFlag)

FocusEvent(Component src, int type, boolean temporaryFlag, Component other)

*) The argument temporary flag ~~focus event~~ is set to true if the focus event is temporary. Otherwise, it is set to false (For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, focus is temporarily lost)

loses ← A → gains the focus

if B other as it is other will gain focus

*) getOppositeComponent() gives information about opposite components.

*) boolean isTemporary()

The method ~~reta~~ returns true if the change is temporary.

→ The InputEvent class

Component Event
↓
InputEvent
↓
Keyboard & mouse event.

*) InputEvent is a subclass of ComponentEvent

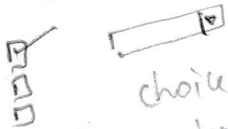
*) It is the superclass of component input event. Its subclasses are KeyEvent and MouseEvent.

*) InputEvent class defines the following eight integer constants that can be used to obtain information about any modifiers associated with this event:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

*) isAltDown(), isAltGraphDown(), isControlDown(), isMetaDown() & isShiftDown() methods test if these modifiers were pressed at the time this event was generated.

→ ItemEvent class:



check   choice or dropdown

*) 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 types of item events.

DESELECTED → The user deselected an item.

SELECTED. → The user selected an item.

*) ItemEvent defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state.

*) ItemEvent has this constructor.   select / Deselected   what have to select

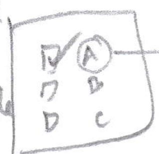
ItemEvent(ItemSelectable src, int type, Object entry, int state)   selectable & deselectable

*) src → reference to the component that generated this event. ex:- this might be a list or choice

name   element

type → Type of event.

getItemSelectable



getItem()

The specific item that generated the item event is passed in ~~entry~~ entry

state → current state of the item.

*) getItem () → used to obtain a refrence to the item that generated an event.

getItemSelectable () → can be used to obtain a reference to the Itemselectable object that generates an event.

getstateChange() → method returns the state change for the event (selected or not selected).

→ The KeyEvent Class.

*) A KeyEvent is genrated when keyboard input occurs.

*) There are three types of key events, which are identified by these integer constants.

Ctrl, Alt   A-Z   virtual key code

KEY_PRESSED, KEY_RELEASED & KEY_TYPED.

*) The first two events are generated when key is pressed and released.

*) The last event occurs only when a characte is generated.

*) There are many other integer constants that are defined by KeyEvent. For example VK_0 through VK_9 and

→ The MouseEvent class (cont)

int getButton()

It returns a value that represents the button that causes the event. The return value will be one of these constants defined by MouseEvent

NOBUTTON	BUTTON 1	BUTTON 2	BUTTON 3

*) NOBUTTON → value indicates that no button was pressed or released.

*) Java SE 6 added three methods to MouseEvent that obtain the coordinates of the mouse relative to the screen rather than the component. They are

point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()

*) The getLocationOnScreen() method returns a Point object that contains both the x and y coordinate.

→ The MouseWheelEvent class.

*) The MouseWheelEvent class, it is a subclass of MouseEvent. Not all mice have wheels. Mouse wheels are used for scrolling.

*) Two integer constants defined are:

WHEEL_BLOCK_SCROLL → A page-up or page down scroll event occurred.

WHEEL_UNIT_SCROLL → A line up or line-down scroll event occurred.

Constructor: MouseWheelEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup, int scrollHow, int amount, int count)

*) int getWheelRotation()

It returns the number of rotational units. If the value is positive, the wheel moved in Counterclockwise. Otherwise clockwise.

*) int getScrollType()

It returns either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL

→ The MouseEvent Class

*) There are seven types of mouse events.

MOUSE_CLICKED → The user clicked the mouse.
MOUSE_DRAGGED → The user dragged the mouse.
MOUSE_ENTERED → The mouse entered a component
MOUSE_EXITED → mouse exited from a component.
MOUSE_MOVED → mouse was moved
MOUSE_PRESSED → mouse was pressed
MOUSE_RELEASED → The mouse was released

*) Constructor: MouseEvent(Component src, int type, long when,
int modifiers, int x, int y, int clicks,
boolean triggersPopup)

*) Modifiers → The modifiers argument indicates which modifiers were pressed when a mouse event occurred.

*) The coordinates of the mouse are passed in x & y.

*) The click count is passed in clicks.

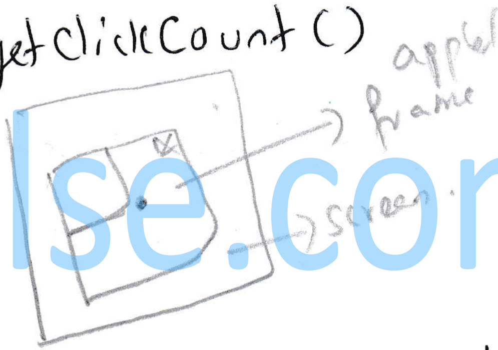
*) The triggerspopup flag indicates if this event causes a pop-up menu to appear on this platform.

*) getx() & gety() methods are used to return the x & y coordinates of the mouse when the event as occurred.

*) getpoint() - Alternatively, this method can be used to obtain the coordinates.

*) translatePoint() (int x, int y) method changes the location of the event.

*) getclick count() method obtains the number of mouse clicks for this event. i.e int getclickCount()



*) The TextEvent class

*) These are generated by text fields and text areas when characters are entered by a user or program.

*) constructor : TextEvent (object src, int type)

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

*) TextEvent defines the integer constant TEXT_VALUE_CHANGED.

→ The WindowEvent Class

*) There are seven types of window events. The constants and their meanings are shown here;

WINDOW_ACTIVATED → The windows was activated
WINDOW_CLOSED → window has been closed
WINDOW_CLOSING → user requested that the window be closed.
WINDOW_DEACTIVATED → window was deactivated.
WINDOW_ICONIFIED → The window was iconified.
WINDOW_OPENED → The window was opened

*) Constructor: WindowEvent(Window src, int type)

src → is a reference to the object that generated this event.
type → type of event

*) getWindow() method It returns the window object that generated the event.

→ Sources of Events

Event source	Description.
1) Button →	Generates action events when the button is pressed.
2) Checkbox →	Generates item events when the check box is selected or deselected.
3) Choice →	Generates item events when the choice is changed.
4) List →	Generates action events when an item is double-clicked, generates item events when item is selected or deselected.
5) Menu item →	Generates action events when a menu item is selected. generates item events when a checkable menu item is selected
6) Scrollbar →	Generates adjustment events when the scroll bar is manipulated.
7) Text components →	Generates text events when the user enters the character.
8) Window →	Generates window events when a window is activated, closed, deactivated etc.

→ Event Listener Interfaces

Interface	Description
1) ActionListener	→ Defines one method to receive action events.
2) Adjustment listener	→ Defines one method to receive adjustment events.
3) ComponentListener	→ Defines four methods to recognize when a component is hidden, moved, resized or shown.
4) containerListener	→ Defines two methods to recognize when a component is added to or removed from a container.
5) FocusListener	→ Defines two methods to recognize when a component gains or loses focus.
6) ItemListener	→ Defines one method to recognize when ~~a component~~ the state of an item changes
7) KeyListener	→ Define three methods to recognize when a key is pressed, released or typed.
8) Mouse listener	→ Define five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed or released.

9). MouseMotionListener → Defines two methods to recognize when the mouse is dragged or moved.

10) TextListener → Defines one method to recognize when a text value changes.

11) WindowListener → Defines seven methods to recognize when a window is activated, closed, deactivated, opened or quit.

Assignment: Event Listener Interfaces

→ Handling Mouse Events.

*) To handle mouse events, we must implement the MouseListener and the MouseMotionListener interface.

*) The following applet demonstrates the process. It displays the current coordinate of the mouse in the applet's status window with various mouse method().

// Demonstrate the mouse event handlers.

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

public class MouseEvents extends Applet
        implements MouseListener, MouseMotionListener
{
    string msg = " ";
    int x=0, y=0;
  public void init()
  {
    addMouseListener(this);
    addMouseMotionListener(this);
  }

// Mouse clicked
  public void mouseClicked(MouseEvent me)
  {
    x=0;
    y=10;
    msg= "Mouse clicked";
    repaint();
  }
```

```
// Mouse entered
public void mouseEntered(MouseEvent me)
{
   x = 10;
   y = 10;
   msg = "mouse entered";
   repaint();
}
// mouse exited
public void mouseExited(MouseEvent me)
{  x = 10;
   y = 10;
   msg = "Mouse exited";
   repaint();
}
// button pressed.
public void mousePressed(MouseEvent me)
{
   x = me.getX();
   y = me.getY();
   msg = "Down";
   repaint();
}
// button released
public void mouseReleased(MouseEvent me)
{  x = me.getX();
   y = me.getY();
   msg = "up";
}  repaint();
```

```
// mouse dragged.
public void mouseDragged
          (MouseEvent me)
{
  x = me.getX();
  y = me.getY();
  msg = "*";
  showStatus("Dragging mouse at
     " + me.getX() + "," + me.getY());
}
  // Display
public void paint(Graphics g)
{
  g.drawString(msg, x, y);
}
```

→ Handling Keyboard Events

*) To handle keyboard events, we use the same general architecture as shown in mouse event.

*) we have to implement a KeyListener interface.

*) Keyevents are generated when KEY_PRESSED, KEY_RELEASED & KEY_TYPED.

// Demonstrate the key event handlers

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class simplekey extends Applet
    implements KeyListener
{
  String msg = " ";
  int x=10, y=20;
 public void init()
 {
   addKeyListener(this);
 }
```

```
public void keypressed (KeyEvent ke)
{
  showStatus ("Key Down");
}
public void keyReleased (KeyEvent ke)
{
  showstatus ("Key up");
}
public void keyTyped (KeyEvent ke)
{
    msg = msg + ke.getKeychar ();
    repaint();
}
```

Display

```
public void paint (Graphics g)
{
    g.drawString (msg, x, y);
}
}
```