```
class Demo
{
    public static void main (String args[])
    {
        try
        {
            ArithmeticException ae = new ArithmeticException
                                    ("can't divide by zero");
            throw ae;
        }
        catch ( ArithmeticException e)
        {
            System.out.println (e);
        }

        System.out.println(" After catch");
    }
}
```

enter another num

o/p :

```
java.lang.ArithmeticException : can't divide zero
After catch
```

II] To rethrow exception from a method

⇒ When we rethrow exception from a method, it becomes an uncaught exception of that method or constructor.

⇒ The programmer who wants to call that method/ constructor should call it with proper try-catch.

⇒ Though this increases the responsibility of a programmer, it facilitates him with a freedom to take corresponding action on that exception.

eg → : class file made by the 1st programer

```
class First
{
    void meth (int mm)
    {
        try
        {
            int k = 100/mm ;
            System.out.println ("k = "+k);
        }
        catch ( ArithmeticException e )
        {
            throw e ;
        }
}
```

→ Logic of 1st .class file has been implemented by us.

```
class Demo
{
    public static void main (String args[])
    {
        First f = new First();

        try
        {
            f.meth ( Integer.parseInt(args[0]));

        }
        catch ( ArithmeticException e )
        {
            S.o.p ("I'm handling exception
                    in my own way");
        }

        S.o.p ("After catch");
    }
}
```

'throws' keyword

⇒ throws keyword is used to intimate user about the uncaught exception in a method.

⇒ throws keyword is written in method name followed by names of Exception classes spparated by commas.

⇒ For some exceptions it is necessary to write 'throws' clause.

⇒ Such exceptions are known as checked exceptions.

⇒ And, all other exceptions are known as unchecked.

* we should not use 'throws' keyword in main method in case of not using the try catch for the uncaught exception. This is because, this uncaught exception will go to default exception handler of java which makes the program terminate after the notification of error.

⇒ we should make our exceptions as checked.

Eg.

```
class First
{
    void meth() throws IllegalAccessException
    {
        try
        {
            IllegalAccessException ie = new IllegalAccess
                                        Exception();
            throw ie;
        }
        catch (IllegalAccessException e)
        {
            throw e;
        }
    }
}

class Demo
{
    public static void main (String args[])
    {
        First f = new First();
        try
        {
            f.meth();
        }
        catch (IllegalAccessException e)
        {
            S.o.p. ("I'm handling exception in my
                                        own way");
        }
        S.o.p ("After catch");
    }
}
```

Creating our own Exception

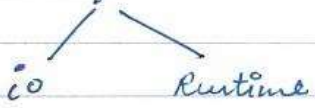
— Manual Exception

eg →

```
class MyException extends Exception
{
    private int i;

    MyException ( int num)
    {
        i = num ;
    }
    public String toString ( )
    {
      returns "MyException : value greater than
       100" ;
    }
}
class First
{
    void square (int num) throws MyException
    {
        if (num> 100)
        {
            MyException me = new MyException(num);
            throw me;
        }
        S.o.p ( "square = "num* num);
    }
}
```

```
class Demo
{
    public static void main (String args[])
    {
        First f = new First();

        try
        {
            f.square (Integer.parseInt (args[0]));
        }

        catch ( MyException me )
        {
            s.o.p (me);
        }

        System.out.println ("After catch");
    }
}
```

If any manual exception extends Exception class then the exception will become checked while if the exception ~~is~~ extends Runtime Exception or the exception classes which are in below hierarchy, then the exception becomes unchecked.



It's good to intimate the users hence, checked exceptions should be made, and therefore the manual exceptions should extends the 'Exception' class.

'try could be followed by either finally or catch block'

finally keyword

Eg →

```
try
{
        try
        {
            =
        }
        finally
        {
            =
        }
}
catch ( Arithmetic ae )
{
    =
    =
}
```

⇒ When an exception is raised in try block, the control of the program is passed to catch block.

⇒ Hence, the code in try block after exception is skipped.

⇒ But sometimes, this code may contain some important logic which has to be executed in any condition.

⇒ We put such code in a ~~try~~ finally block followed by try.

⇒ When an exception is raised in try block, before passing control to catch block, the control of the program is passed to finally block.

⇒ The finally block is executed first, & then catch block is executed.

eg

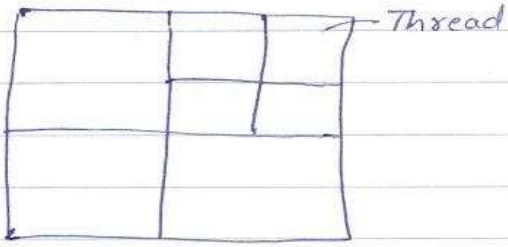
```
class First
{
    void meth1()
    {
        try
        {
            System.out.println("Inside meth1");
            int m=100, n=0;
            int k=m/n;
        }
        finally
        {
            System.out.println("Finally for meth1");
        }
    }
    void meth2()
    {
        try
        {
            System.out.println("Inside meth2");
            return;
        }
        finally
        {
            System.out.println("Finally for meth2");
        }
    }
}
```

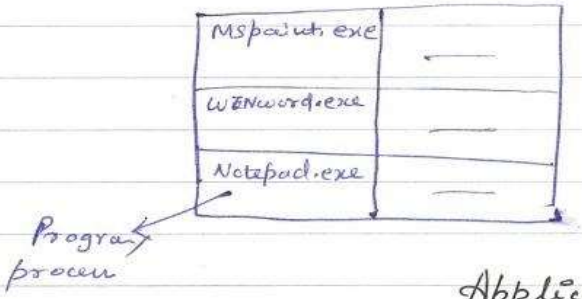
```
class Demo
{
    public static void main (String args[ ])
    {
        First f = new First();

        try
        {
            f.meth1( );
        }
        catch ( ArithmeticException e )
        {
            System.out.println ("Catch for meth1");
        }

        f.meth2( );
        System.out.println ("After catch");
    }
}
```

Multithreading

Multitasking – (i) Proces based
(ii) Thread based

Single-threaded system





Application

⇒ Thread is the smallest unit in case of Thread based multitasking

⇒ proces is the smalest unit in case of proces based multitasking.

⇒ Threads share the same memory allocated for that particular proces.

⇒ It means, it is the division of logic not the division of memory.

⇒ We are dividing the proces in different logics known as threads.

⇒ Single threaded System.

Multithreaded System

⇒ A thread represents an individual path of execution.

⇒ A thread is a power which takes on CPU time and makes CPU to execute specific logical task which has been delegated to it.

⇒ If running thread is suspended, any other thread can take on CPU time and starts working. This reduces wastage of CPU time.

Single threaded system

⇒ A single thread is in finite loop. It executes all the tasks sequentially & when all are done, the thread is dead. If thread suspends due to any reason, wastage of CPU time takes place.

* 3 (daemon) - inbuilt threads in java
* Main thread - daemon thread
* Garbage collector thread (Java checks periodically which is known as garbage collection where it checks the useless objects & removes it, according to a particular object)
* Applet thread
  ↳ (awt-event queue 1)

Whatever written under psvm, that becomes the logic of the main thread.

⇒ Peculanities ⇒

→ Every child thread is generated by main thread.

→ If main dies, then all the child threads are supposed to be dead.

⇒ Child thread → Thread made by the programmer

⇒ In java.lang package, there is a class 'Thread' alongwith constructors are ~~also~~ given which is used to make the objects which ultimately ~~form~~ results in child thread.

'Thread' class
↓
constructors
↓
object

We'll learn about the different methods of 'Thread class' & hence all the methods are applicable to the objects, i.e. 'Thread'(child thread).

⇒ Logic of child thread is written in

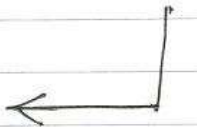
```
public void run()
{

}
```

Main thread (inbuilt thread given by java)

→ The execution of all console based applications start with 'main' method

→ This main method represents the logic of one inbuilt thread named 'main-thread'.

→ Two peculiarities



Thread class
main method — child thread

⇒ All the methods (variables/resources) of Thread class are applicable to child thread as well as 'main' since it's is also an object of Thread class.

Methods of Thread class

(i) currentThread() :-

Syntax → static Thread currentThread()
(↑ return type)

→ returns reference (object) of currently working thread.

(ii) setName() :-

Syntax → final void setName (String threadname)

→ accepts one string & sets it as a name of thread.

(iii) getName() :-

Syntax → final String getName()

→ this method returns the name of thread

(iv) sleep( ) :-

Syntax → static void sleep (long milliseconds)
throws InterruptedException

→ This method suspends thread for specified time period.

```
class Demo
{
    public static void main (String args[])
    {
        Thread t;
        t = Thread.currentThread();
        System.out.println(t);

        t.setName ("My Thread");
        System.out.println(t);

        String tname = t.getName();
        s.o.p(tname);

        for (int i=1; i<=4; i++)
        {
            System.out.println(i);

            try
            {
                Thread.sleep (1000);
            }
```

toString() method is implicitly called because it is in the Object class & can be overridden in the main also (implicitly).

since sleep() method is static & it throws exception, hence it should be called by class name & should be caught.

```
        catch (InterruptedException ie)
        {
            System.out.println(ie);
        }
    }
}
```

Thread group { in memory where thread is stored.

name ↓ priority ↓ group ↓

o/p- Thread [main, 5, main]
Thread [MyThread, 5, main]

My Thread
1
2
3
4

Two techniques to create our own child threads→

I] Implementing 'Runnable' interface

java.lang
↓
Runnable
↓
public abstract void run()

→ we have to create the logic before creating the thread.
Steps are following →

A class should implement the Runnable interface & override the run method, before creating the child thread.

Step 1 → class ThreadLogic implements Runnable
{
public void run()
{
=
Logic
=
}
}

Step 2 → Constructors of 'Thread' class :

↳ Thread (Runnable obj)
↳ Thread (Runnable obj, "String threadname")

eg. ThreadLogic TL = new ThreadLogic ();
Thread t = new Thread (TL);

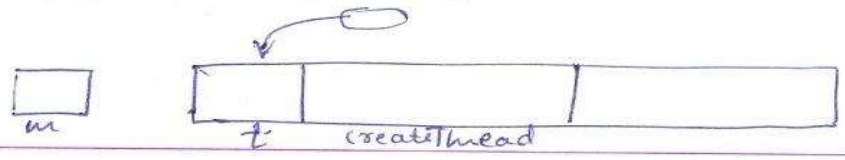
Since, we've ~~to~~ can't pass the object of an interface, hence we've to pass the object of such a class which implements that interface & overrides the method.

Step 3 → Starting 'Thread'

Method of 'Thread' class :-

public void start()
eg → t.start();

eg class ThreadLogic implements Runnable
{
public void run()
{
for (int i=1; i<=4; i++)
{
System.out.println(i);




```
            try
            {
                Thread.sleep (1000);
            }
            catch ( InterrupedException ie )
            {
                System.out.println (ie);
            }
        }
    }
class MyThread
{   Thread t;
    void createThread( )
    {
        ThreadLogic TL = new ThreadLogic( );
        t = new Thread(TL);
    }

    void beginThread( )
    {
        t.start( );
    }
}

class Demo
{
        p.s.v.m (String args[])
        {
            MyThread m = new MyThread( );
            m.createThread ( );
            m.beginThread( );
            // m.t.start ( );
        }
}
```

m.t → child thread

Creating multiple child threads

⇒
```
class MyThread implements Runnable
{
    Thread t;

    MyThread (String thname)
    {
        t = new Thread (this, thname)
        t.start();
    }
    Public void sum()
    {
        for(i=1; i<=4; i++)
        {
            s.o.p (t.getname() + " " + i);

            try
            {
                Thread.sleep(1000);
            }
            catch ( InterruptedException ie)
            {
                S.o.p ("Thread Interrupted");
            }
        }
    }
}
class Demo
{
    p.s.v.m (String ays[])
    {
        new MyThread ("One");
        new MyThread ("Two");
        new MyThread ("Three");
    }
}
```

* Threads work randomly

Extending 'Thread' class

Thread class itself implements the Runnable interface. The run method of Runnable is abstract and is overriden in the Thread class with a blank body.

→ Now, object of MyThread ~~is~~ can be treated as the child thread.
→ MyThread is as good as Thread class

→ All the methods like, start, sleep etc. come into MyThread and can be directly accessed. getname & setname can also be directly accessed, because they are directly inherited to the MyThread class from Thread class.

→ This method is not preferred over implementing the Runnable interface, because it supports multiple inheritance.

⇒ When any class extends Thread class, it becomes as good as Thread class.

⇒ Hence, it's object can be treated as a child Thread, since we know that the object of a 'Thread' class is said to be child Thread.

→ javap . lang. Thread → public void run()
→ javap . lang . Runnable → abstract void run()

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=1; i<=4; i++)
        {
            System.out.println(getName()+";"+i);
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException ie)
            {
            }
        }
    }
}
class Demo
{
    public static void main(String args[])
    {
        MyThread m = new MyThread();
        m.setName("One");
        m.start();
    }
}
```

Lifecycle of a thread

Multiple threads work concurrently not simultaneously.

States of Thread →

New

Runnable — t.start() — unpredictable time

Running — when thread gets CPU time

Blocked — sleeping position — After blocked state, thread shifts to Runnable state.

Dead

→ Sequence of Thread is not under the hands of programmer. We cannot control the flow / sequence of the working of Thread

1) → **New**
When we create thread, it is in new state.

2) → **Runnable**
When we start thread it is in runnable state.
A thread in runnable state is ready to run
→ if it gets CPU time & memory, it starts working.
After Runnable state, we can't predict its running state while runnable can be predicted.

3) → **Running**
When thread actually gets CPU time and starts working, It is in running state.

4) → **Blocked**
When thread in running state is suspended due to any reason, it is in blocked state.

* A thread in blocked state is neither working nor dead.
When the blocked state of thread is over, thread is in runnable state.

5) Dead →
When thread completes it's task and is terminated, it is in dead state.

eg
```
class MyThread implements Runable
{
    Thread t;
    MyThread()
    {
        t = new Thread(this);
        t.start();
    }

    public void run()
    {
        for(int i=1; i<=4; i++)
        {
            s.o.p("child"+i);
        }
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException ie)
        {
        }
    }
```

* System.exit(0) → command to make the ~~dead~~ main thread dead.

```
class Demo
{
    p.s.v.m( ~~static~~ String args[] )
    {
        new MyThread();

        for (int i=1; i<=4; i++)
        {
            s.o.p ("Main :" +i);

            try
            {
                Thread.sleep(1000)
            } }
            catch (                    )
            {

            }
        }
    }
}
```

logically in the o/p, child thread should execute first, because in the main, child thread is called by its constructor, but this is not so. Main thread executes first mostly because when the control goes from the constructor to the constructor defined in the class implementing Runnable, then the child thread is ~~get~~ made start but the thread will acquire the runnable state not running and it is totally unpredictable to guess the time for a thread to shift from 'runable' to 'running' state.

Methods of Thread class

isAlive() & join()

(i) **isAlive()**–

Syntax : final boolean isAlive()

⇒ If thread invoking this method is dead, this method returns false otherwise it returns true.

(ii) **join()**–

Syntax : final void join() throws InterruptedExcep

⇒ The thread in whose logic join method is called is suspended / blocked. It remains suspended till ~~any other thread entering~~

Eg.

```
t0                t1                t2
run()             run()             run()
{                 {                 {
  -                 -                 -
  - t2.join();      - t0.join         - t1.join  → This statement by no means affect the t1 thread. It has to do only with t2.
  -                 -                 -
  -                 -                 -
}                 }                 }
```

Deadlock cond^n

↳ Non-usable program

At design time, deadlock cond^n should be tackled.

the thread invoking join method does not complete its task & is terminated.

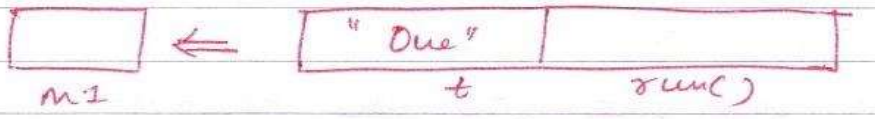
```
eg. class MyThread implements Runnable
{
    Thread t;

    MyThread ( )
    {
        t = new Thread (this);
        t.start ( );
    }

    Public void run( )
    {
        }
        }
        }
    }
class cDemo
{
    p. s. v. m (Strings args[ ])
    {
        MyThread m1 = new MyThread (this, "One");
        MyThread m2 = new MyThread ("Two");
        MyThread m3 = new MyThread ("Three");
    }
```

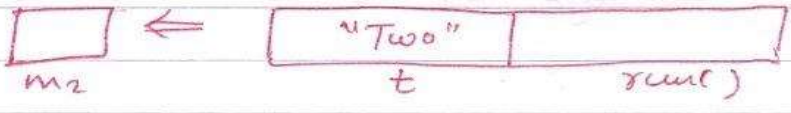
Rough es'

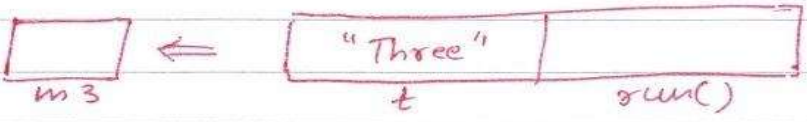
'Threads work ~~is~~ not simultaneously ~~not~~ concurrently.'

```
class MyThread implements Runnable
{
    Thread t;

    MyThread (String tname)
    {
        t = new Thread (this, tname);
        t.start();
    }
    public void run()
    {
        for (int i=1; i<=4; i++)
        {
            s.o.p (t.getName() + ": " + i);

            try
            {
                Thread.sleep (1000);
            }
            catch (InterruptedException ie)
            {
            }
        }
    }
}

class Demo
{
    p.s.v.m (String args[])
    {
        MyThread m1 = new MyThread ("One");
        MyThread m2 = new MyThread ("Two");
```

```
MyThread m3 = new MyThread("Three");

S.o.p (m1.t.isAlive());
S.o.p (m2.t.isAlive());
S.o.p (m3.t.isAlive());          } Status of thread

try
{
    m1.t.join();
    m2.t.join();
    m3.t.join();
}
catch (InterruptedException ie)
{
}

S.o.p.(m1.t.isAlive());
S.o.p.(m2.t.isAlive());          } gives False False False
S.o.p(m3.t.isAlive));
}
}
```







In one process,

⇒ More than one threads can be in runnable state ~~in~~ at one instant

⇒ But, only one thread will be in running state.

⇒ Thread works randomly.

⇒ ~~Wor~~ Time required to do a particular work by the thread ~~is~~ can vary and is not fixed (unpredictable).

⇒ In the entire library of java, there is no such method which can pass the control to any thread.

⇒ Hence as a programmer we cannot control the sequence of Threads.

⇒ But for this purpose, thre is a negative way given by java.

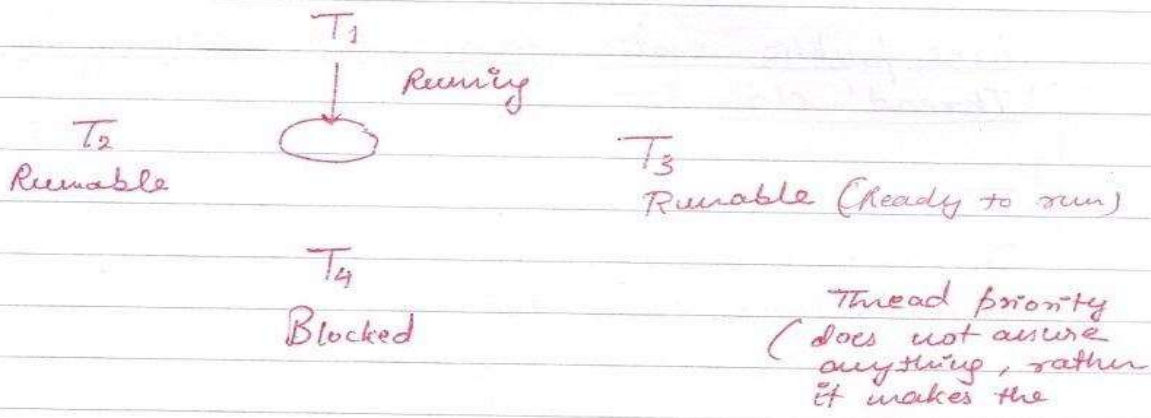
↳ We are not managing the control sequence rather we manage the sequence of sleeping of threads.

t_1
t_2 } making these two sleep will ultimately makes
t_3 } the t_1 faster.

Thread priority

Defines two things →

eg.



Thread priority (does not assure anything, rather it makes the

⇒ Thread priority defines two things →

1) If running thread of a process is suspended or dead then which thread among the Runnable state threads would take on CPU time first?

⇒ The thread with highest priority among the runnable state threads, will be having most of the chances.

2) If multiple threads of a process are working concurrently, for specified time period, then which thread will utilise how much CPU time?

⇒ The threads with highest priorities will be

By default, priority is 5.

utilising more CPU time.

Thread priority - 1 to 10 cannot be in dec
(1 = min, 10 = max)

Three public static final int variables of 'Thread' class :—

Thread.MIN_PRIORITY → 1
Thread.NORM_PRIORITY → 5
Thread.MAX_PRIORITY → 10

Methods of Thread class →

i) setPriority() :

final void setPriority(int P)

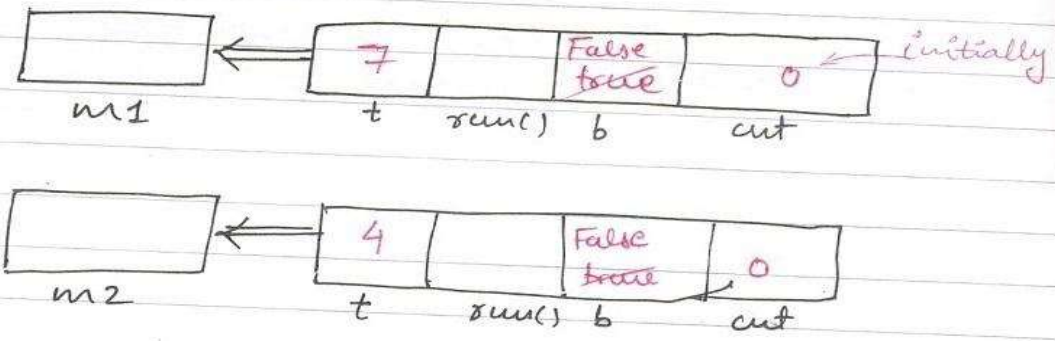
ii) getPriority() :

final int getPriority()

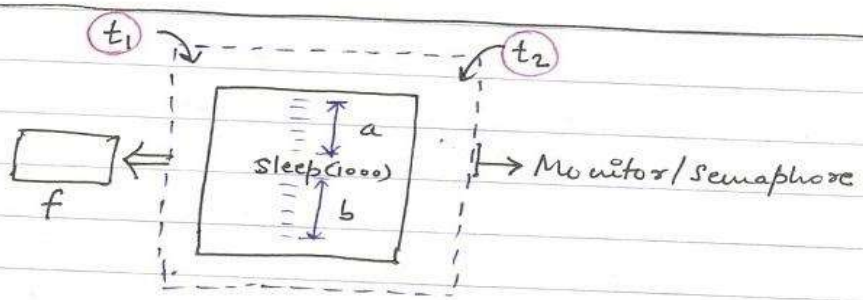
* final variables are written in capitals.

```
class MyThread implements Runnable
{
    Thread t;
    boolean b = true;
    long cnt = 0;

    MyThread ( int p)        → since we've to give priority in the constructor defined in main.
    {
        t = new Thread (this);
        t.setPriority (p);
        t.start();
    }

    public void run()
    {
        while (b)
        cnt++;
    }
}

class Demo
{
    public static void main (String args[])
    {
        MyThread m1 = new MyThread (7);
        MyThread m2 = new MyThread (5);

        try
        {
            Thread.sleep (10,000);
        }
        catch (InterruptedException ie)
        {
        }
```

```
        m1.b = False ;
        m2.b = False ;

        S.O.p (m1.cut);
        S.O.p (m2.cut);
    }
}
```



eg



Expected o/p:

a
b
a
b

Actual o/p:

a
a
b
b

Here, two threads t_1 & t_2 try to access the meth() method in their run() method. So, there could be logical problems because of the concurrency to access the method. Hence, monitor / semaphore is being introduced to overcome this logical problem. In this, method is protected by monitor when the first thread try to access the method.

In database, asynchronism is solved by the 'locks' method. The same type of problem is solved in java is with the help of monitors / semaphore.

Synchronization

Every object is covered by semaphore / monitor by default it is deactivated.

↳ Solution to stop concurrency to avoid asynchronous system.

⇒ When a common resource in memory is utilised by multiple clients, concurrently, some logical problems may raise due to concurrency.

⇒ Such a system is known as 'Asynchronous system'.

⇒ In multi-threading also when a single object in memory is being accessed concurrently by multiple threads, some logical problems may raise. or system may get asynchronized.

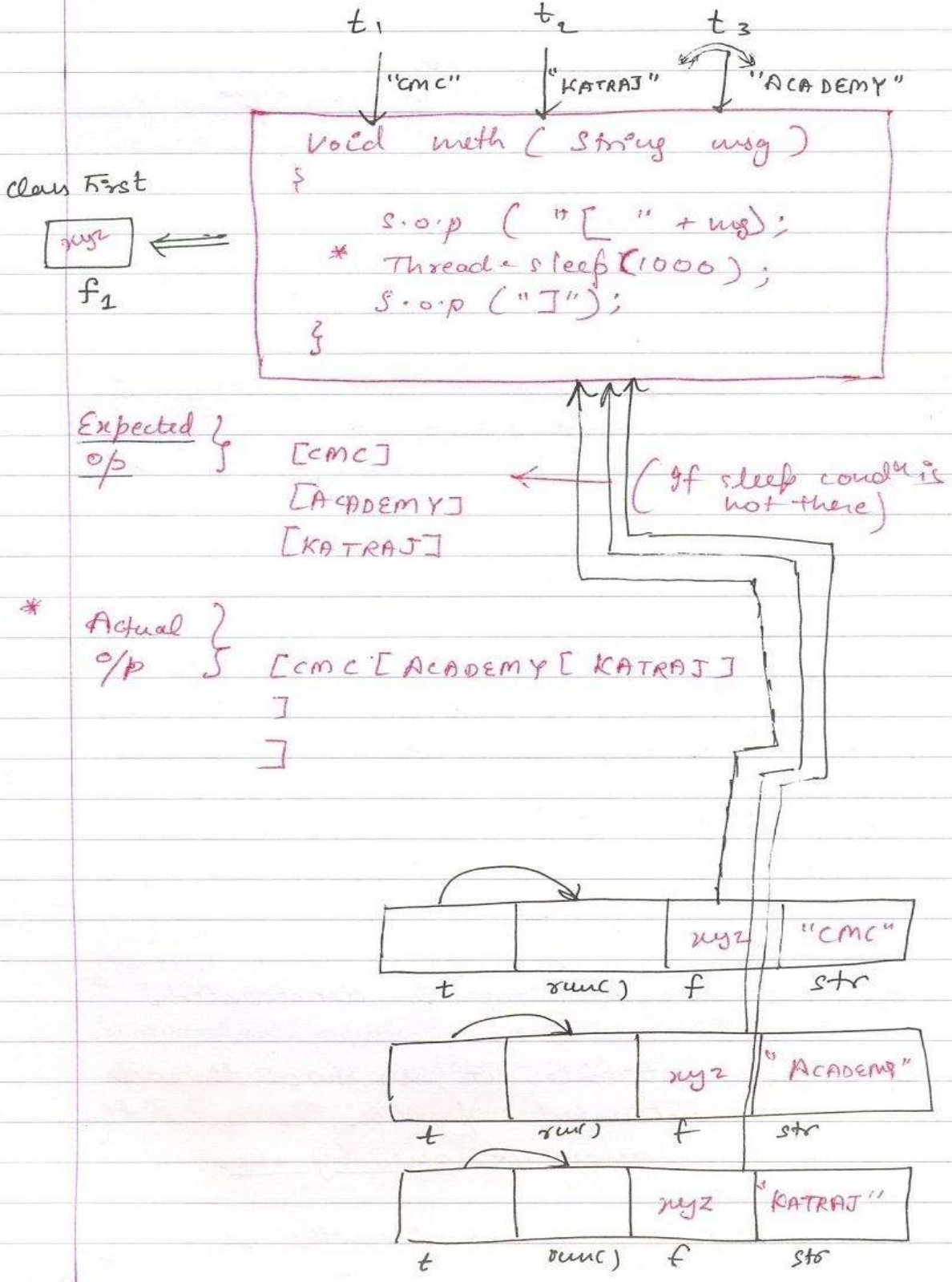
⇒ Every object in memory is covered by a monitor / semaphore.

⇒ By default, the monitor is deactivated.

⇒ When we activate it, no two threads can access activated portion concurrently, but they can access it one by one.

⇒ This automatically synchronises the system.

Memory Diagram 2



We've to make three objects of first class and then we've to access the meth().

```
class First
{
synchronized void meth (String msg)
        {
                System.out.print ("[" + msg);

                Try
                {
                        Thread.sleep(10,000);
                }
                catch(InterruptedException ie)
                {
                }

                System.out.println("]");

        }
}

class MyThread implents Runable
{
        Thread t;
        First f;
        String str = " ";

        MyThread (First z, String message)
        {
                f = z
                Str = message;
                t = new Thread (this);
                t.start();
        }
```

```
    public void run()
    {
        f.meth(str);
    }
}

class Demo
{
    public static void main(String args[])
    {
        First f1 = new First();

        new MyThread(f1, "CMC");
        new MyThread(f1, "Academy");
        new MyThread(f1, "Katraj");
    }
}
```

Interthread communication

Eg Two threads → with diff^{n} logic

t_1
run()
{
= ①
= ③
}

t_2
run()
{
= ②
= ④
}

Suppose, we want t_1 & t_2 to perform work in a particular sequence.

→ We cannot use sleep in this case because we cannot predict the time taken to execute a particular logic.

→ We cannot use join also.

Instead of synchronized methods, we do have to use Interthread commun^{n}.

We cannot govern the logic of sequence of thread.

Facts of multithreading

```
class First
{
    int i;

synchronized void put(int num)
    {
        i = num;
        S.o.p("Put : " + i);
    }

synchronized ~~void~~ int get()
    {
        S.o.p("Get : " + i);
        return i;
    }
}

class Putter implements Runnable
{
    Thread t; First f;

    Putter(First z)
    {   f = z;
        t = new Thread(this);
        t.start();
    }
}

~~class Getter implements Runnable~~

    public void run()
    {   int k = 0;
        while(true)
            ~~f.get()~~ f.put(++k);
    }
}
```

```
class Getter implements Runnable
{
    Thread t ; First f ;

    Getter (First z)
    {
        f = z ;
        t = new Thread (this) ;
        t.start () ;
    }

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

class Demo
{
    public static void main (String [args])
    {
        First f1 = new First () ;

        new Putter (f1) ;
        new Getter (f1) ;
    }
}
```

This program gives us the output which shows that instead of monitor / semaphore, we cannot stop the successive occurance of a particular thread because we know that thred used to work concurrently, hence we cannot decide / predict the (one-by-one) sequence. Hence, from (activity) semaphore, we can only make sure about the that only one thread can access the method / object at one time but we can't control the successive occurances.

Hence, we've to introduce other recources from the Object class, like wait(), notify(), etc.

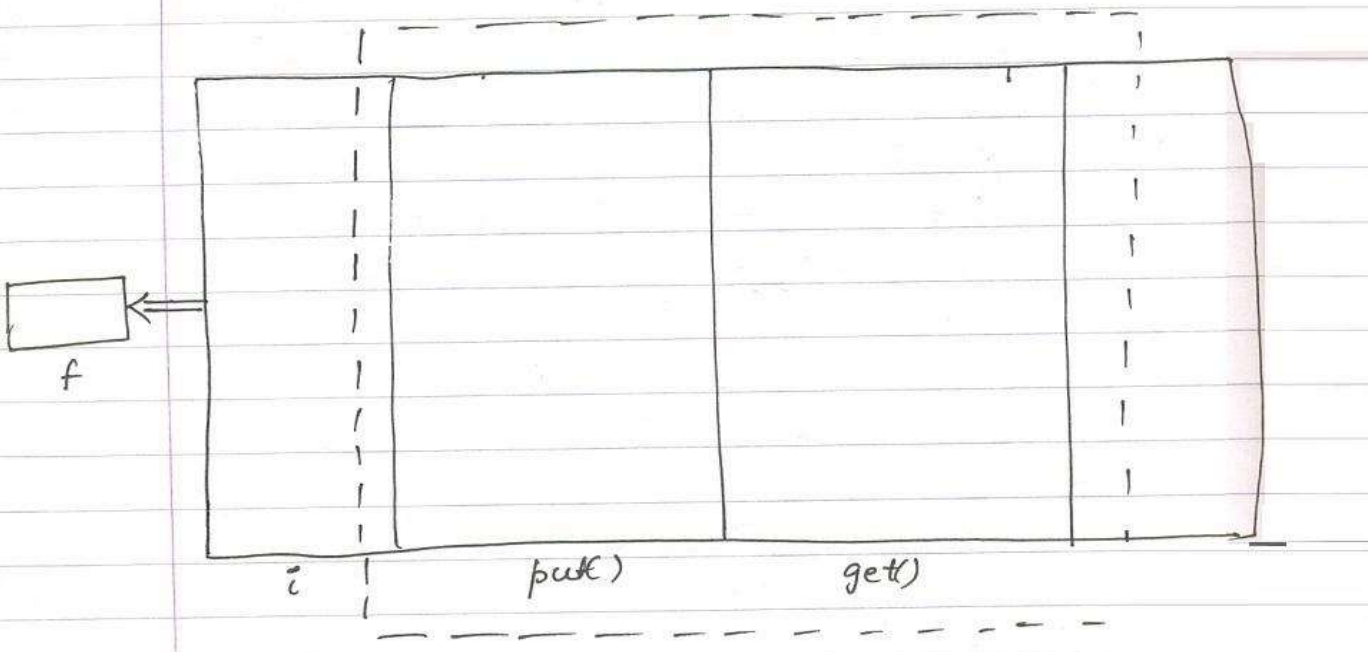
```
class First
{
    int i;
    boolean b = false;

    synchronized void put(int num)
    {
        if(b)
        {
            try
            {
                wait();
            }
            catch(InterruptedException ie)
            {
            }
        }

        i = num;
        S.o.p("Put : "+i);

        notify();
        b = true;
    }

    synchronized int get()
    {
        if(!b)
        {
            try
            {
                wait();
            }
```

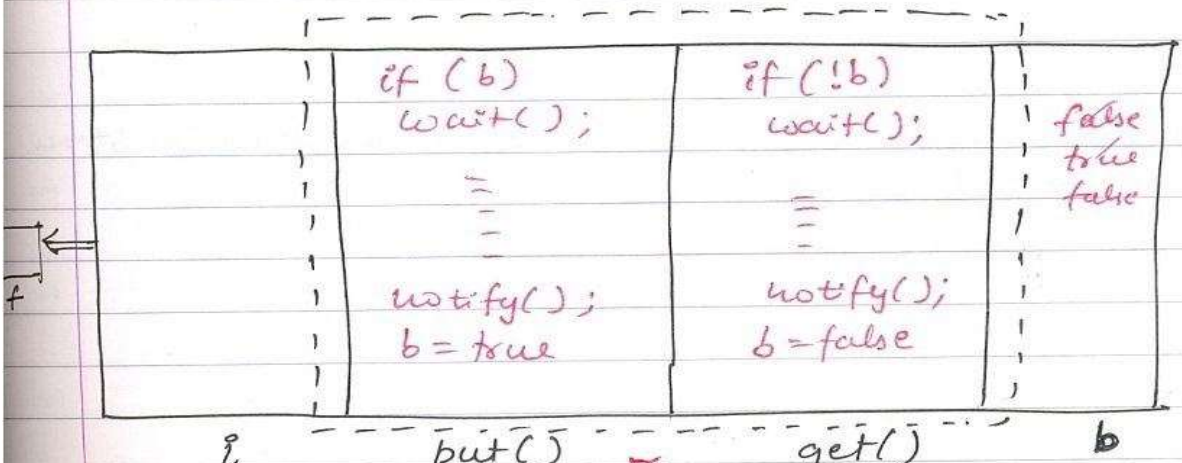
```
            catch (InterruptedException ie)
            {
            }
        }

        System.out.println ("Get :" +i);

        notify();
        b = false;
        return i;

    }
}

class Putter implements Runable
{
    Thread t;
    First f;

    Putter (First z)
    {
        f=z;
        t= new Thread (this);
        t.start();
    }

    public void run()
    {
        int k=0;
        while (f.i<30)
        f.put(++k);
    }
}
```

```
class Getter implements Runnable
{
        Thread t;
        First f;

        Getter(First z)
        {
            f = z;
            t = new Thread(this);
            t.start();
        }

        public void run()
        {
            while (f.i < 30)
              f.get();
        }
}

class Demo
{
    p.s.v.m (String args[])
    {
        First f1 = new First();
        new Putter(f1);

        try
        {
            Thread.sleep(5000);
        }
        catch(InterruptedException ie)
        {
        }
```

to check whether the interthread connection still happens

```
        ww Getter(f1);
    }
}
```





class Putter implements runnable

Thread t;

```
run()
{
    int k = 0;
    while (true)
        f.put(++k);
}
```

↳ ∞ loop

class Getter implements runnable

Thread t;

```
run()
{
    while (true)
        f.get()
}
```

In case of more than two methods in a monitor, we use a method named as 'NotifyAll'.

Methods of 'object'

i) ⇒ **wait( ) :-**

final void wait( ) throws InterruptedExaption

ii) **notify( ) :-**

final void notify( )

iii) **notifyAll( ) :-**

final void notifyAll( )

wait( ) →

⇒ The thread in whose body, the is suspended, it leaves the cunet monitor. And it remains suspended till any other thread entering the same monitor does not notify it

notify( ) & notifyAll →

⇒ The notify method notifies the last waiting thread whereas notifyAll method notifies all waiting Threads.

Applet Basics

⇒ ~~The~~ Applets provide GUI in java.

⇒ They don't contain 'main' method.

⇒ The runtime system for Applets is AppletViewer.

⇒ Java provides following resources to create applet →

Component
↓
Container
↓
Panel
↓
Applet
↓
MyApplet

} – java.awt package

– java.applet package

Making packages of our own :

```
// Program in which we've to use the packages ↴
// Store this file in : C:\Java\Jdk1.6.0_27\bin

import mypack1.MyClass1;
import mypack1.mypack2.MyClass2;

class Demo
{       psum(String args[])
```

```
    }
        MyClass1 m1 = new MyClass1();
        m1.myMeth1();

        MyClass2 m2 = new MyClass2();
        m2.myMeth2();
    }
```

```
// mypack1 2
// Store this file in : C:\Java\Jdk1.6.0\bin\mypack1

package mypack1;

public class Myclass1
{
    public void myMeth2()
    {
        S.o.p("This is myMeth1 in MyClass1 in
                Mypack1");
    }
}
```

```
// mypack2 In mypack1
// Store this file in: C:\Java\Jdk1.6.0\bin\mypack1\mypack2

package mypack1.mypack2;

public class Myclass2
{
    public void myMeth2()
    {
```

```
        S.O.P("This is myMeth2 in Myclass2 in
                mypack2");
    }
}
```

If we've same class name in the package as well as sub package then we can access the classes specifically by using dot operator b/w package name & the class name.

mypack1.myclass1
mypack2.myclass2

→ Every class of an applet is made public

→ java.awt.* & java.applet.* must be imported in every applet.

→ AppletViewer also considers the commented portion in which html tags are present.

→ Applet is saved by its class name.

MyApplet.java

Now, for compilation →

javac MyApplet.java

& for execution →

appletViewer MyApplet.java.

→

1440 pixel = 1 inch

For

→ simple applet showing text in the applet window →

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

/*
  <applet code = "MyApplet" width = "300" height = "300">
  </applet>
*/

public class MyApplet extends Applet
{
        public void paint( Graphics g )
        {
                g.drawString ("MC Katraj", 20,20);
        }
}
```

(name of the class)

Lifecycle of an Applet :

1) init()
initialises & loads the resources of the Applet class in memory & it is called first.

2) start()
applet window is called / start.

3) paint()
It redraws the contents on an applet.

4) stop()
When we minimize the window, the applet is said to be stopped / suspended.

5) destroy()
When user closes applet window, the stop method is called followed by destroy(). This method removes the contents of applet from memory.

init()
The variables used in applet code are initialised in this method.

start()
This method is immediately called after init. The thread executing applet is started due to this.

paint()
This method is called immediately after start. It redraws the contents of Applet whenever called.

stop()
→ When user minimises applet window, this method is called. It suspends the thread executing applet.
→ When user maximises applet window, the thread is restarted.
→ It means the start method is called again, followed by paint.

Program showing the methods of an applet class ↴

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

/*
 <applet code = " MyApplet " width = " 700 " height = "300">
 </applet>
*/

public class MyApplet extends Applet
{
        String str = " " ;

        public void init( )
        {
            str = str + " in init( )" ;
        }

        public void start( )
        {
            str = str + " in start( )" ;
        }

        public void paint( Graphics g )
        {
            try
            {
                Thread.sleep ( 1000 ) ;
            }
```

To check whether paint method redraws the applet window.

```
        catch ( InterruptedException ie)
        {
        }
        str = str + " in paint()";
        g.drawString (str, 20,20);
    }

    public void stop()
    {
        str = str + " in stop()";
    }

    public void destroy()
    {
        str = str + "in destroy()";
        |s.o.p (str);| ← for printing it on console.
                          generally it's not preferred.
    }
}
```

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

/*
<applet code = "MyApplet" width = "700" height = "300">
</applet>
*/

public class MyApplet extends Applet implements Runnable
{
    int x=0, y=0;                    so that value can be changed

    Thread t;

    boolean b1 = true;               for inner stroke (up, bottom)
    boolean b2 = true;               for outer stroke (left, right)

    public void init()
    {
        t = new Thread (this);
        t.start();
    }

    public void run()
    {
        while (true)                 infinite loop
        {
            repaint();               → calls the paint method
            try
            {   Thread.sleep(30)
            }
```

```
catch (InterruptedException ie)
{
}

if (b2)
{
    if (b1)
    {
        x = x+5;
        y = y+10;
    }

    if (y>270) b1 = false;

    if (!b1)
    {
        x = x+5;
        y = y-10;
    }

    if (y<=0) b1=true;
}

if (!b2)
{
    if (b1)
    {
        x = x-5;
        y = y+10;
    }
    if (y>270) b1 = false;
```

```
            if(!b1)
            {
                x = x-5;
                y = y-10;
            }
            if (y<=0) b1 = true;
        }
        if (x<=0) b2 = true;
    }
}
```

paint → doesn't let the figure washout

```
public void update( Graphics g)
{
    setBackground( (color.cyan);   → (final variable)
    setforeground (Color.red);

    g.fillOval( x, y, 30, 30);
}
}
```

java.awt package

but we know final variables are written in caps, but for the objects of Color class, java has provided us with both ~~the~~ caps & small letters for the final variables.

For ~~search~~ viewing the objects →
javap. Package-name. Class-name
java.awt Color

↳

for filling colour ↳

```
g.drawOval( x, y, w, h);
g.fillOval( x, y, w, h);
```

getCodeBase() & getDocumentBase() :

Syntax →

```
URL getCodeBase()
URL getDocumentBase()
```

URL → java.net package

getCodeBase method returns object of URL class, representing current folder

getDocumentBase method returns object of URL class representing current document/applet/file

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

/*
   <applet code = "MyApplet" width = "700" height = "300">
   </applet>
*/

public class MyApplet extends Applet
{
        public void init()
        {
            Font f = new Font("Arial", Font.BOLD, 15);
            setFont(f);
        }
```

Font() → java.awt.

→ Universal Resource Locator

```
public void paint ( Graphics g )
{
        URL u;
        g.setColor ( Color.red );
        u = getCodeBase( );
        g.drawString (u.toString( ), 20, 20);
        g.set Color ( Color.blue);
        u=getDocumentBase( );
        g.drawString (u.toString( ), 20,40);
    }
}
```

For increasing font, we've

```
public void init( )
{
        Font f= new Font("Arial", Font.BOLD , 15 );
        setFont (f);
}
```

name Style Size

__/__/____

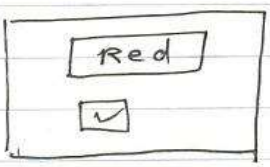
Delegation Event Handling

0) Sources →

Source means origin. When user interacts with source, an event is generated
eg. Button, checkbox are events.

Sources are the objects that generates events.

eg.



java.awt.event
↓
1) ActionEvent
2) ItemEvent

Events are generated when user interacts with the

AppletViewer
↓ catches the events & sends it to
Listener
(takes action)

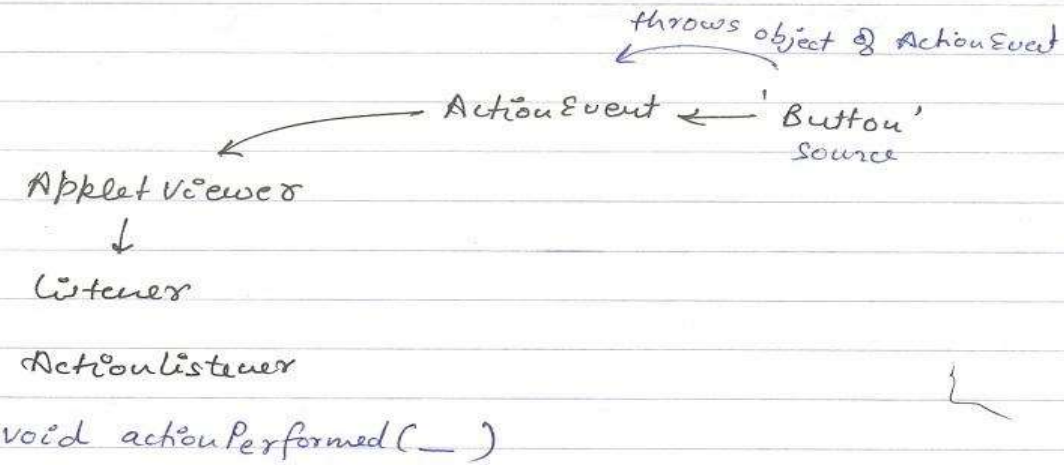
Events are the objects

Execution of applet takes place under the influence of AppletViewer

We know in java.awt package, there is a readimade class Button along with constructors. By calling the constructors with the new operator we can apply the components.

Checkbox & radiobutton are in the same compont

→ Listeners are interfaces



---

① Sources –

Sources are the objects that generate events.

② Events –

→ Events are the objects that represent change in the internal state of sources.

→ Events are generated when user interacts with GUI.

③ Listeners –

→ When an event is created, an appletviewer does catch it and sends it to the listener.

→ The listener is responsible to take corresponding action on event.

→ As soon as this action is completed, the control of the program is passed back to AppletViewer

Mouse Events

Behind every event, there has to be a class. Similarly 'Event class' is there.

Event class :- java.awt.event.MouseEvent

Constructors : MouseEvent (Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean popupflag)

src - Source (our applet window initially can also be said as the source, before creating any other source)

type :- 7 public static final int variables of 'MouseEvent' class :-

MouseListener
- MouseEvent.MOUSE_CLICKED — same loc^{n} → press + release
- MouseEvent.MOUSE_PRESSED — downmost
- MouseEvent.MOUSE_RELEASED — uppermost
- MouseEvent.MOUSE_ENTERED — entered in applet window
- MouseEvent.MOUSE_EXITED — exit from the applet window

MouseMotionListener
- MouseEvent.MOUSE_MOVED
- MouseEvent.MOUSE_DRAGGED

when - Represents System time (at what time mouse event has been performed.)

modifiers - (Special keys)
↳ to represent, again we've public int static final variable
SHIFT_MASK
ALT_MASK
CTRL_MASK
for eg, sometimes we use the mouse button with these special keys.

x & y : coordinates of x & y (mouse pointer) at the time of event.

clicks : click events (Total) in the Applet's life so far.

popupflags : represents whether any pop-up window is being opened due to event.

---

3) Listener →
MouseListener - clicked, press, release, enter, exit
MouseMotionListener - move, dragged.

In mouselistener we're given with 5 abstract methods (which accepts the object of MouseEvent's object.

→ Now we've to override all the methods listed in both the ~~list~~ interfaces or rather we've to give

Methods of 'MouseListener' interface :-

public abstract void mouseClicked (MouseEvent me)
" " " mousePressed (")
" " " mouseReleased (")
" " " mouseEntered (")
" " " mouseExited (")

Methods of 'MouseMotionListener' interface :

public abstract void mouseMoved (MouseEvent me)
public abstract void mouseDragged (").

The object me contains ~~the~~ all the 8-parameters.

If we'll remove the x & y coordinates or any other parameter, ~~or~~ then we're given with some of the methods of mouseEvent class. ↘

Methods of 'MouseEvent' class -

① getX() :- int getX() — when we want a certain string to printed at the very same location where the event is performed in the applet window.

② getY() :- int getY()

Binding sources with listeners →

Eg:- Say we've a source Button b1. Now, when the button is clicked, object is passed to ActionListener. Illy

binding b/w src & Listener

Button b1 → ActionEvent → ActionListener

Checkbox ch1 → ItemEvent → ItemListener

At somewhere, we've to mention the binding of source to the listener in the code & we're provided with a method for achieving this. i.e. TypeListener

```
public void addTypeListener (TypeListener obj)
```

```
b1.addActionListener ( ActionListener obj )
```

↓

can be replaced by 'this' because first of all we cannot make obj of an interface, & hence we to send the object of a class which will implement this interface.

```
b1.addActionListener ( this obj );
```

Similarly for checkbox

```
ch1.addItemListener (this);
```

In the program →

Since we're using events, hence we've to import our more package, i.e.

```
java.awt.event.*
```

source → ~~event~~ applet window itself

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



public void init()
{                                          // binding
    this.addMouseListener(this);
    addMouseMotionListener(this);
}
```

can also be written but since it is inherited from the previous panel class, hence there is no need to write it, ~~but~~ for.

Key Events

Event class :- KeyEvent ← class

Constructor :- KeyEvent (Component src, int type, long when, int modifiers, int keycode, char ch)

type :- 3 public static final int variables of 'KeyEvent' class :-

KeyEvent . KEY_PRESSED } keycode ✓, ch X → CHAR_UNDEFINED
KEY_RELEASED }
KEY_TYPE } ch ✓, keycode X ✓

The keys which are responsible for typing

Keycode :- A key may or may not have keyascii value or it may have two keyascii values.

⇒ But every key has keycode value & it is unique for a key.

⇒ For representing keycodes, public static final integer values are provided. eg.

VK_A (65) to VK_Z (66), VK_0 (48) to VK_9 (57) (for number)

VK_RIGHT, VK_LEFT . . .

ch : character

Listener :- KeyListener

For small letters & for capital letters, there is a unique keycode, whereas this is not the case of ascii values.

⇒ Keys may or may not have ascii value or key may have two ascii value.

Methods of 'KeyListener' interface :-

public abstract void KeyPressed (KeyEvent ke)
——"—— KeyReleased (——"——)
——"—— KeyTyped (——"——)

Methods of 'KeyEvent' class :-

(i) getKeyCode( ) : int getKeyCode( )

(ii) getKeyChar( ) : char getKeyChar( )

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

/*
<applet code = "KeyDemo" width = "300" height = "300">
</applet>
*/

public class KeyDemo extends Applet implements KeyListener
{
    String str = "";

    public void init( )
    {
        addKeyListener( this );
    }
```

```
public void keyPressed (KeyEvent ke)
{
    setBackground (Color.red);
}
public void keyTyped ( KeyEvent ke)
{
    str= str + ke.getKeyChar();
    repaint();
}

public void paint( Graphics g)
{
    g.drawString (str, 20,20);
}
}
```

Page-168

Creating sources ⇒

Until now, we had the applet window as the source. Now we'll be to built our own sources.

↳ Creating Labels

← A class is given by java

sequence to follow ↴

⇒ AWT class :- Label

⇒ Constructors :-

Label ( )

Label (String str)

Label (String str, int alignment)

Public static final int variables

↳ Label . RIGHT
LEFT
CENTER

They have values

⇒ Methods of 'Label' class :

(i) setText( ): void setText (String str)

(ii) getText( ): String getText ( )

(iii) setAlignment( ): void setAlignment (int alignment)

(iv) getAlignment( ): int getAlignment ( )

⇒ Labels don't generate events

⇒ Labels are passive controls, they don't generate events.

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

/*
 <applet code = "LabelDemo" width="300" height = "300">
 </applet>
*/

public class LabelDemo extends Applet
 {
      Label L1, L2, L3;

      public void init()
      {
           L1 = new Label ("One");
           L2 = new Label ("Two");
           L3 = new Label ("Three", Label.RIGHT);

          add (L1)
          add (L2);
          add (L3);

          L1.setBackground (Color.cyan);
          L2.setBackground (Color.cyan);
          L3.setBackground ( Color.cyan);

           L2.setAlignment ( Label.CENTER);
           L2.setText (" Hello");
      }
}
```

/ /

Adding sources to applet window →

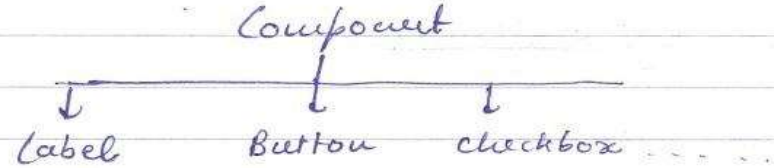
We'll study 9 components and they are the children of Component class.

Component add (Component obj)

This add method accepts object of the parent class and returns also, the parent class, i.e. component.

We've to use this method to add every component / source to the applet window because the components won't be visible if this method is not used.

We know,



Adding component to applet window through add() method →

Component add(Component obj)

Creating Buttons

AWT class - Button

Constructor -

1) Button ( )
2) Button (String str)

Methods -

1) setLabel( ) :- void setLabel (String)
2) getLabel( ) :- String getLabel( )

uptil this, button will be formed

Event class -

ActionEvent

Listeners -

ActionListener

Methods of 'ActionListner' interfaces -

public abstract void actionPerformed (ActionEvent ae)

events are called

important points →

→ source should be added to the listener

Method of ActionEvent class -

1) getX( ) :- int getX
2) getY( ) :- int getY

Method of 'ActionEvent' class –

String getActionCommand()

↳ This method returns label of pressed button

Creating Checkboxes

AWT class: Checkbox

Constructors:

Checkbox()
Checkbox(String str)
Checkbox(String str, boolean on, CheckboxGroup cg)
Checkbox(String str, CheckboxGroup cg, boolean on)

Methods:

1) setLabel()
2) getLabel()
3) setState() :— void setState(boolean on)
4) getState() :— boolean getState()

event, listener, abstract methods

Event class: ItemEvent

Listener: ItemListener

Methods of 'ItemListener' interfaces:

⇒ itemStateChanged(ItemEvent ie)

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

/*<applet code = "CheckboxDemo" width = "300" height="300">
  </applet>
*/

public class ChecksDemo extends Applet implements
                                        ItemListener
{
    Label L;
    Checkbox ch1, ch2, ch3, ch4;

    public void init()
    {
        L = new Label("Choose players:-");

        ch1 = new Checkbox("Sachin");
        ch2 = new Checkbox("Saurav");
        ch3 = new Checkbox("Rahul");
        ch4 = new Checkbox("Anil");

        add(L);
        add(ch1);
        add(ch2);
        add(ch3);
        add(ch4);

        ch1.addItemListener(this);
        ch2.addItemListener(this);
        ch3.addItemListener(this);
        ch4.addItemListener(this);
}
```

```
public void itemStateChanged (ItemEvent ie)
{
        repaint();
}

public void paint( Graphics g)
{
        String str = " ";

        if ( ch1. getState())

        str = str + ch1.getLabel + "  ";

        else if (ch2.getState())

        str = str + ch2.getLabel+"  ";

        else if (ch3 .getState))

        str = str + ch3.getLabel + "  ";

        else if (Ch4. getState())

         str = str + ch4.getLabel

         g.drawString (str, 20, 100);

    }
}
```

Checkbox group → radio button

appletviewer calls init, start, paint at the start of the applet window.

Creating Choices

AWT class : Choice

Constructor : Choice()

Methods :

1) ⇒ add() – void add(String str)
2) ⇒ getSelectedItem() – String getSelectedItem()
3) ⇒ getSelectedIndex() – int getSelectedIndex()
4) ⇒ getItemCount() – int getItemCount()
5) ⇒ getItem() – String getItem(int index)
6) ⇒ select() – void select(int index)

more methods can also be accessed by javap.

Event class : ItemEvent

Listener : ItemListener

Method of 'ItemListener' interface

⇒ itemStateChanged(ItemEvent ie)

– elseif can be used in the code because single selection has to be done.

Creating Lists

AWT class - List

Constructors of 'List' class -

List (int numrows)

List (int numrows , boolean multselect)

Methods - All methods & choice are applicable

extra methods

getSelectedItems ( ) :- String [ ] getSelectedItems ( )

getSelectedIndexes ( ) :- int [ ] getSelectedIndexes ( )

Event class – ItemEvent / ActionEvent

→ When user clicks on list item, ItemEvent is genrated

→ When double clicks on list item, ActionEvent is genrated

⇒ Creating Textfield

AWT class : TextField

Constructors : TextField()
TextField (int numchars)
TextField (String str, int numchars)

Methods : (i) setText()
(ii) getText()
(iii) getSelectedText() :-
String getSelectedText()

Event class : actionEvent

When user presses enter button in textfield, actionEvent is generated.

Single line → textField
Multiple lines → TextArea

⇒ Creating Textfield Area

AWT class : TextArea

Constructors : TextArea(int numrows, int numcols)
TextArea(String str, int numrows, int numcols)

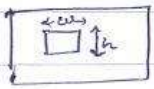
⇒ no event.

All methods of textField are applicable to textArea.

→ implements nothing since no event is generated.

```
public class TextAreaDemo extends Applet
{
    TextArea t;

    public void init()
    {
        String str = "India is \n my country";
        t = new TextArea (str, 5, 20);
        add(t);
    }
}
```

⇒ SetBounce method of component class :



w → width
h → height

setBounce (x, y, w, h)

By default, a fixed flow layout is carried out by java, if we want to ~~set~~ use setBounce method, then first of all we've to remove the fixed default layout, by doing -

⇒ setLayout (null);

Code :

Layout Managers

↳ interface

void setLayout (LayoutManager obj)

FlowLayout BorderLayout GridLayout CardLayout

↳ classes implementing the interfaces
↳ and these classes have their constructors to override.

FlowLayout

If we want to set our own layout other than the default centre layout, then we've to use the FlowLayout class along with its constructors. It's constructors accept 3 parameters

BorderLayout

Changes the BorderLayout by it's constructor which accepts two parameters.

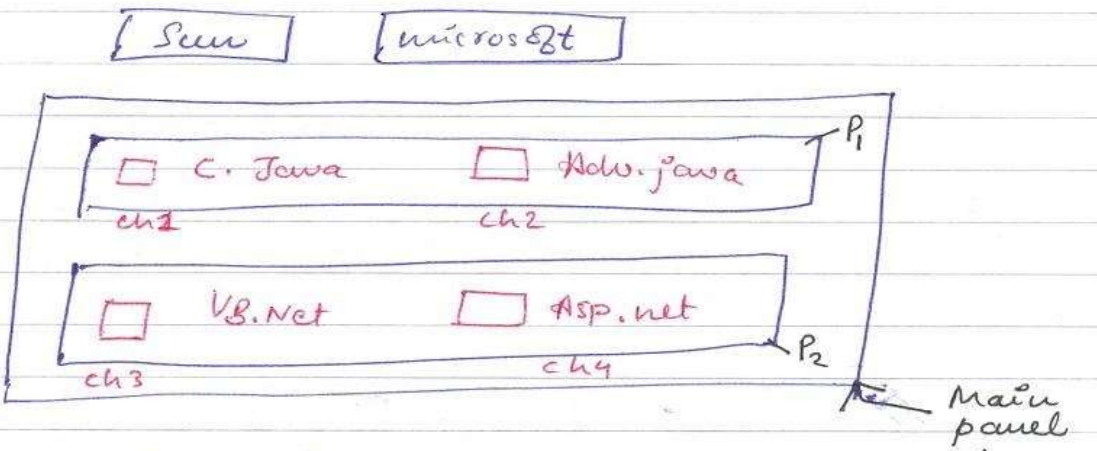
GridLayout

CardLayout :

(Same as Tabcontrol in VB

Eg

If we've two tabs & three panels ← same as frows



The requirements are →

- Two buttons b1, b2
- Four checkboxes ch1, ch2, ch3, ch4
- Three panels p1, p2, mainpanel

→ make two buttons
→ add them in appletwindow
→ make four checkboxes & add them in panels

```
ch1 = new Checkbox ("    ");
ch2 = new Checkbox ("    ");
p1.add (ch1);
p1.add (ch2);
```

→ like for checkboxes 3 & 4

→ Now add p1 & p2 to mainpanel by names "One" & "two" respectively.
& set the mainpanel to cardLayout declare it above.

CardLayout cl = new CardLayout( );

Now, since the whole thing is going to happen at the click event of buttons hence we've to write →
implements actionlistener

→ And now add the main panel to the Applet

→ we can also change the colour of the ~~layou~~ panels.

Creating Scrollbar

AWT class : Scrollbar

Constructors :

Scrollbar (int Style) → Scrollbar.HORIZONTAL / VERTICAL

Scrollbar (int style, int initvalue, int thumbsize, int min, int max)

Methods :

1. setValue ( )
   void setVale (int val)
2. getValue ( )
   int getValue ( )
3. getMinimum ( )
   int getMinimum( )
4. getMaximum ( )
   int getMaximum( )

Range of RGB - 0-255

__/__/____

Event class: AdjustmentEvent

Listener : Adjustment_Listener

Method of 'AdjustmentListener' interface –

public abstract void adjustmentValueChanged
(AdjustmentEvent ae)

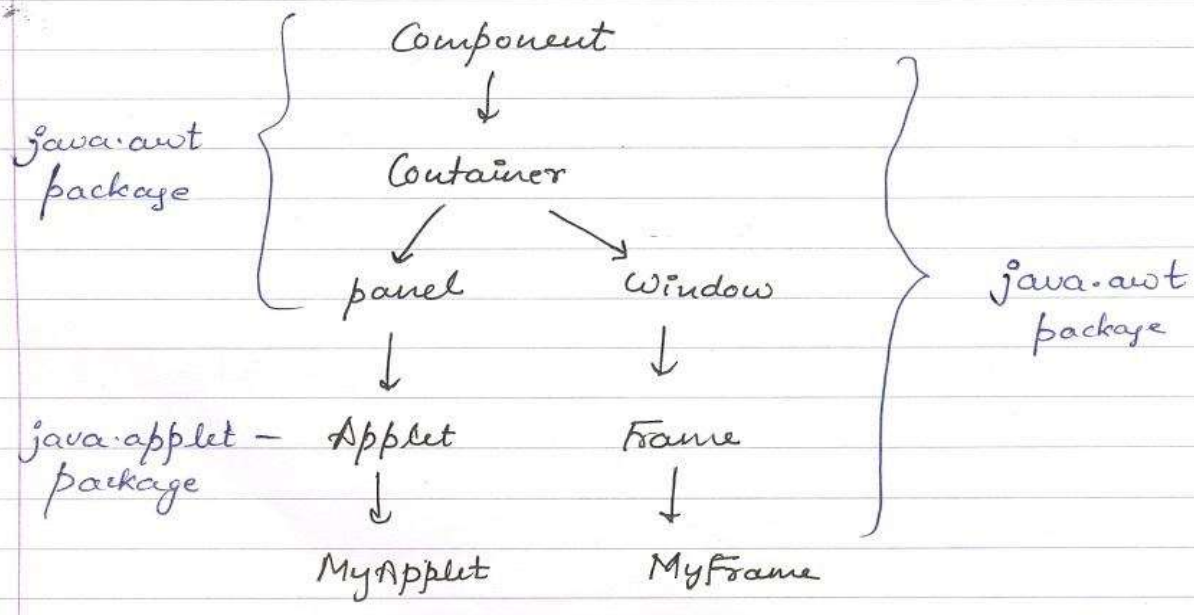
Creating Frames :

⇒ Frames are light weight as compared with applets.

⇒ Frames provide some extra awt components/controls like, menus and file dialog boxes.

⇒ Frames are less secured and hence mostly used in database applications.

Resources are different for creating frames —



** Frames can be called from applets as well as they can be directly made in the console based application. In console based applications, we do not need to make it public and the applet portion

→ For handling the events of the frame like, close, minimise, & maximize, we've to use WindocoListener which is the part of WindowEvent. There are 7-methods which has to be overridden. The methods accept object of windowEvent.

→ For making the window invisible after clicking on cross button, we've to use setVisible(false) but, the window will be dissapeared only. For making the program to stop we've two options— (i) Ctrl+c & (ii) a method dispose().

→ Same as applets, we can make buttons, checkboxes, etc. in the frame also. But instead of doing them in the init method, we've to use the constructor of the frame only for this purpose.

→ Now, we know default layout of an applet is set but in case of frames, it is not so. Hence, we've to set our own. This can be done by implementing one more class i.e. ActionListener. And hence, it's method ActionPerformed will be called.

code :-

→

File dialog box

⇒ works with frames only.

In java.awt package, we've an inbuilt class FileDialog which accepts 3 parameters — (i) frame obj (ii) name of dialog box

Two public static final variables of FileDialog
FileDialog.OPEN
" , SAVE
→ returns only address
→ doesn't save really

Two methods are there to get the address
→ fd.getDirectory() & fd.getFile()