Javapाटील.com

An initiative by CPC Academy Pune

IN JUST
16000/-
Rs. + Tax

Learn Java from Basics to all advanced technologies with live Project in struts or spring

100%
PLACEMENT ASSISTANCE

Content :

- Basic Java
- J2EE
- Struts 1.3
- Struts 2.0
- Hibernate
- Spring
- Ajax
- Jquery
- JSON
- Maven
- Angular JS
- Web Services
- JasperReports
- Boot Strap
- Eclipse
- Net Beans

CPC Academy Pune
True Education

9921558775, 9075578252

Shop No. 4/5, C Wing, Ashoka Agam, Dattanagar Road, Ambegaon, Katraj, Pune

Ph. 020-65242122

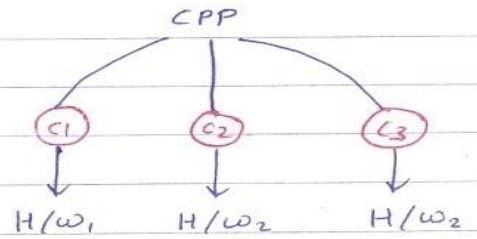
Java – Class based language

Need of Java – Java is made for embedded softwares where we wanted platform independency.

"Platform independency" ⇒

Example from CPP

.CPP
↓
compile
↓
.exe



In case, if we want to execute a single .CPP program on different hardwares then we build different compilers.

Hence, to remove this complexity of building compilers, java came into existence.

Code extension –

- Java
↓
compile
↓
.class
(Byte code) ← optimized set of instructions.
↓
JVM – Java Virtual Machine

- Runtime system for java
- It does process byte code line by line, Hence, java is known as an interpreter for java. Hence, java is known as compiled as well as interpreted language.
- we need different JVM for each platform.
- The designing of JVM is easier. Hence, java is known as platform independent language.

↓

JIT (Just In Time) compiler

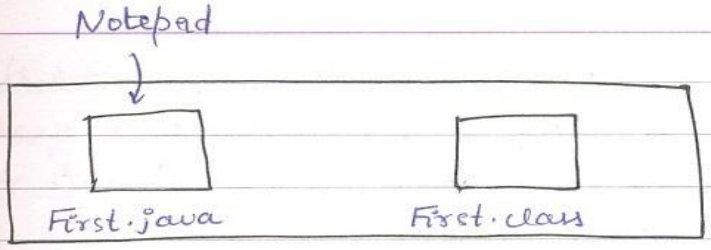
- It accepts code group by group and converts it into .exe.

⟹ **First Java Program**

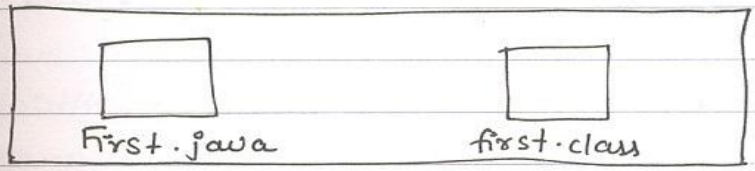
user defined data type ↑ Class First ⟶ class name

```
Class First
{
    public static void main (String args[])
    {
        System.out.println (" xyz ")
    }
```

- Class → user defined data type
- First → class name
- public → access specifier
- static → can be accessed without object
- void → return type
- main → method
- String args[] → command line arguments
- System → In built class
- out → output variable
- System.out → output stream
- println → method



C:/Program files/java/JDK 1.6/bin
C:/cd Program Files/java/jdk 1.6/bin ↵
c:/ . . . . . /bin > javac First.java ↵
C:/ . . . . . /bin > java First ↵



C:/ cd Sachin ↵
C:/ sachin > set path = "C:\Program files... \bin ↵
c:/Sachin > javac First.java ↵
C:/ Sachin > java First ↵

#

bin = java_home
because all the java core resources are present in the bin.

In 1st case, since we've saved the file in the bin itself, hence all the necessary tools/methods for the compilation are directly available hence javac can be directly used. In 2nd case we've saved a file in c or other directory components for the compilation & execution. Hence, before javac, we've to set the path of bin.

We can save java file with any name. After compilation of the file, java will make the .class file for every class. Execute the method in which main method is present.

```
class First      → main()
{
   =
}
class Second
{
   =
}
class Third
{
   =
}
```

javac First.java ↵
java First

java xyz.java
java First

→ **Data types in Java** - Primitive Data types

1) Numeric -

Integer values		Float values	
byte	8 bits	Float	32 bits
short	16 bits	double	64 bits
int	32 bits		
long	64 bits		

Variable decⁿ

int i
i is assigned 0 by default in case of Java while in c the variable is assigned garbage value

2) Character

In 'C'

→ C considers char as integer
→ width - 8 bits
→ range - 0 to 255
→ Ascii code

In 'Java'

→ Java also considers char as integer
→ width - 16 bits
→ range - 0 to 65,535
→ unicode (universal code)
↓
Internationalization & code

In 0-255 all the English characters are available. In Java, 0-65,535 is provided so that it will include all the possible characters from every human language in the world.

Java gives character & byte stream both while other languages support byte stream.

---

eg.

```
Class Demo
{
    public static void main (String args[])
    {
        char ch1, ch2;
        ch1 = 'A';
        ch2 = 66;
        System.out.println (ch1);
        System.out.println (ch2);
    }
}
```

output : A
B

3) Boolean-

- Java does not consider boolean as integer.
- Any relational statement results in boolean

eg.

```
Class Demo
{
    public static void main (String args[])
    {
        boolean b;
        b = true;
        System.out.println (b);
        int m = 100, n = 200;
        b = m>n
        System.out.println (b);
    }
}
```

output : True
False

In C, CPP & VB, boolean is considered as integer while Java does not ~~support~~ consider boolean as integer.

Predefined keywords are there - true & False. In other languages (C, CPP, VB), 0 is considered as False while any other non-zero number is considered as True.

Handling Strings

```
String str1 = "xyz";
S.O.P (str1);
String str2 = "pqr";
sop (str1 + "  " + str2);
```

(+ : string concatenation character)

- Every string declared in java is an object of string class, which is part of 'java.lang' package. This class contains many functions that can be used for string manipulations. (By default java.lang is called in every programs).

Scope of a variable :

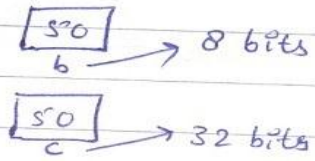
In java coding is done in blocks ({ ... }). A variable declared in a block is local to that

block and can be accessed anywhere in the same block after declaration. But, it can't be accessed in any other block.
⇒ In case of nested blocks, a variable declared in outer block can be accessed in inner block. But a variable of inner block can't be acessed in inner block. But a variable of inner block can't be accessed in outer block.

Automatic Type promotion :

I) Widening conversion —

```
byte b = 50;
int c = b;
```



As shown in above eg, the byte or short value can be copied to int variable, the float value can be copied to double variable, and int value can be copied to long variable. This widening conversion can be possible only if foll- conditions are true.

1) Both data types should be compatible
2) The target data type should be bigger than original data type.

II) Automatic Type Promotion in Mathematical Expressions

```
byte b = 50;
    c = b * 2;
int
```

When we perform some mathematical operation on byte and short, its result is auto-promoted to int value. Similarly, the result of mathematical operations on float is auto-promoted to double value.

Command line arguments ⇒ *

```
Class Demo
{
    ~~Private~~
    Public static void main (String args[])
    {
        System.out.println(args[1]);
    }
}
```

```
                    args[0]  args[1]  args[2]
java Demo abc mno PQR ↵
 dp - mno
```

```
class Demo
{
    public static void main (String args[])
    {
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);
        int sum = num1 + num2;
        System.out.println("Sumation of " + num1 +
                           "and" + num2 + "is" + sum);
    }
}
```

o/p → 10 30
Sumation of 10 and 20 is 30

Operator →

① mathematical

+
−
/
*
%

② Shortcut assignment operators

+=
−=
/=
*=
%=

++ → inc. operator
−− → dec. operator

eg.
i+=2;
i=i+2;

− prefix

```
int i=10;
int j= ++i;
s.o.p(i);
s.o.p(j)
```

i=i++
j=i;
inc
↓
copy

o/p → 11
11

− postfix

```
int i=10;
int j= i++;
s.o.p(i);
s.o.p(j);
```

j=i
i=i+1
copy
↓
inc

o/p → 11
10

③ Relational

<
>
<=
>=
!=

④ Logical

& → AND
| → OR
! → NOT

⑤ Assignment operator

=

- Short circuit AND (&&) -

```
if ( condition 1 && condition 2 )
{
    =
}
```

This operator does not check the 2nd condition if it's false. The o/p of 2 conditions is considered false. But, if 1st condition is true, then the o/p of 2 condition is considered true otherwise false.

Short circuit OR (||)

```
if ( cond^n || cond^n )
{
    =
}
```

This operator does not check the 2nd condition if it is true. The o/p of 2nd condition is considered true if 1st condition is false, it checks the 2nd cond^n and if it is true the total o/p is true otherwise false.

Control flow statements :

* Selection statements —

i) if

```
if (cond^n)
{ = }
```

ii) if else

```
if (cond^n)
{ = }
else { = }
```

ii) Nested if

```
if (cond^n 1)
{
    if (cond^n 2)
    {
      =
    }
}
```

iv) else... if ladder

```
if ( condn 1 )
{
   =
}
else if (condn 2)
{ =
}
else if (condn 3)
{ =
}
else
{
   =
}
```

v) Switch case

```
Switch (expression)
{
  case value 1
    =
  break ;
}
default ;
  =
```

* Loops →

I) while

```
initialization ;
while (condn)
{
   =
   iteration ;
}
```

II) do while

```
initialisation
do
{ =
  iteration ;
} while (condn) ;
```

III for

```
for (initiaration; condn ; iteration)
{
  =
}
```

eg.
```
class Demo
{
    public static void main (String args[])
    {
        int num = Integer.parseInt (args [0]);
        int sum =0;
        int rev = 0;

        while (num>0)
        {
            sum = sum + num % 10;
            rev = rev *10 + num %. 10;
            num = num /10;
        }

        System.out.println ("Summation of digits = "+sum);
        System.out.println ("Reverse of digits = "+rev);
    }
}
```

Class fundamentals :-

Defining class –

```
class ClassName
{
    returntype variablename1;
    :
    returntype methodName (Parameter list)
    {
        =
        =
    }
}
```

⇒ Class is a user-defined data type which encapsulates two things →
- → Data (variables)
- → Code (methods)

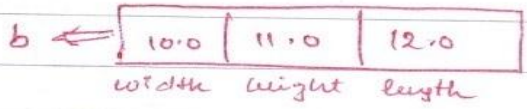
⇒ A class is a logical structure which can be implemented through object. An object is known as an instance of a class.

⇒ The variables declared in a class are known as instance variables because a new copy of variables is created in memory with every object of a class.

⇒ As methods hold all logical part, they are known as code.

⇒ Collectively variables & methods are known as members of a class.

A simple java class




```
class Box
{
    double width, height, length;
}
class Demo
{
    public static void main(String args[])
    {
        Box b = new Box();
        b.width = 10;
        b.height = 11;
```

```
        b.length = 12;
        System.out.println("Volume = " + b.width *
                           b.height * b.length);
    }
}
```

o/p → volume = 1320.0

Creating an object

To create an object of Box class we use the following statement :

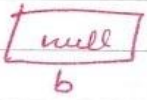
```
Box b = new Box();
```

We can split this statement into two statements as follows →

```
Box b;
b = new Box();
```

↑ operator ↑ Default constructor

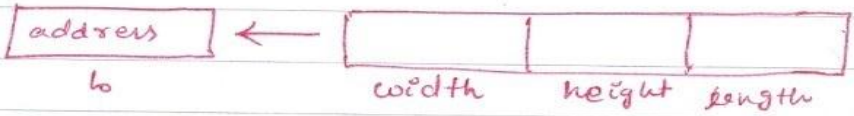
After the execution of 1st statement object reference of Box class named b is created in the memory



→ object reference variable which holds null value initially.

After the execution of 2nd statement, new operator along with default constructor allocates space from object in memory. The address of this space is put in variable b. Hence, any part of object space

can be referred by using variable along with . operator.

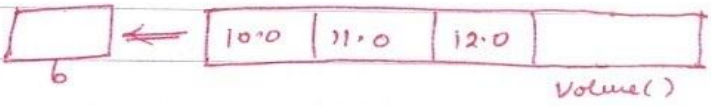


Methods

1) Simple Method :

```
class Box
{
    double width, height, length ;

    void volume()
    {
        S.o.p ("Volume=" + width * height * length);
    }

}

class Demo
{
    P. S. V. M. ( String args[ ])
    {

        Box b = new Box( );
        b. width = 10;
        b. height = 11;
        b. length = 12;
```

```
        b.volume();
    }
}
```

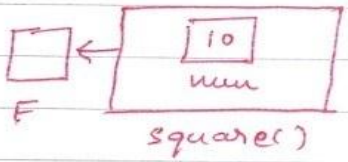


o/p → volume = 1320.0

⇒ The methods are instance methods. It means that a new copy of methods come in memory with every object of a class

2) Method accepting parameter :

```
class First
{
    void square(int num)
    {
        System.out.println("Square=" + num * num);
    }
}

class Demo
{
    p.s.v.m (String args[])
    {
        First f = new First();
        f.square(10);
    }
}
```



o/p – 100

⇒ The variables passed as parameter to method or declared in the body of the method, are known as local variables of that method.

3) Method returning parameter :

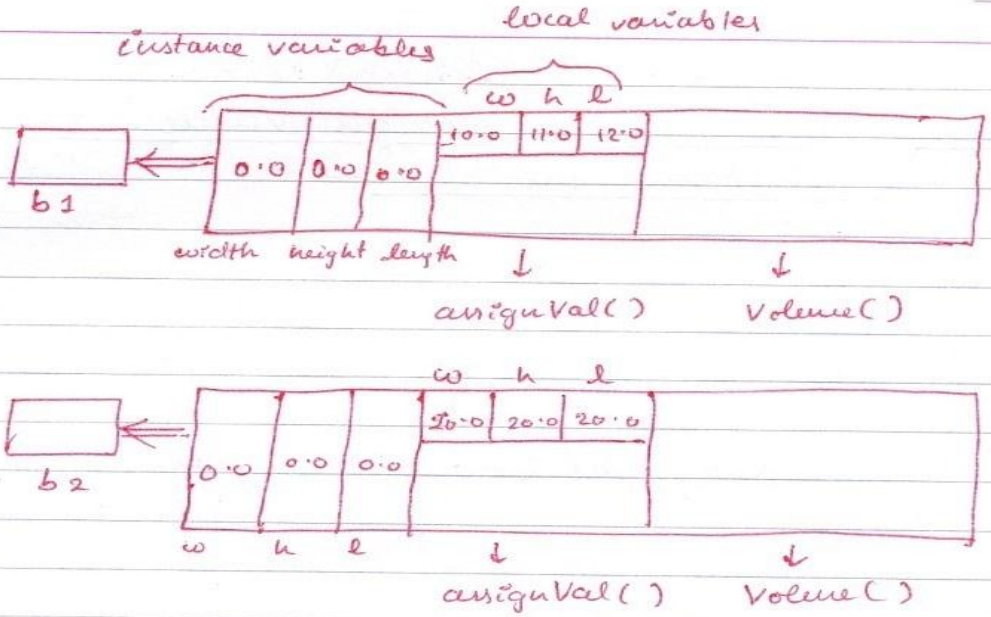
```
class First
{
    int square (int num)
    {
        return num * num;
    }
}

class Demo
{
    p.s.v.m (String args[])
    {
        First f = new First();
        int res;
        res = f.square (10);
        S.o.p ("Square" + res);
    }
}
```

o/p ⇒ square = 100

⇒ We write method returning values to get any intermediate purpose to get solved.

Using method to 'Assign Values' to the set of instance variables —

```
class Box
{
    double width, height, length;

    void assignVal (double w, double h, double l)
    {
        width = w;
        height = h;
        length = l;
    }

    void volume()
    {
        S.o.p("volume=" + width * height * length);
    }
}

class Demo
{
    p.s.v.m(String args[])
    {
        Box b1 = new Box();
        b1.assignVal (10, 12, 12);
        b1.volume();
        Box b2 = new Box();
        b2.assignVal (20, 20, 20)
        b2.volume();
    }
}
```



o/p → volume = 1320.0
volume = 8000.0

'this' keyword –

⇒ When the local variable name in method is conflicted with instance variable name of a class, by name, method does recognise to its local variable and not the instance variable.

↳ This is known as instance variable hiding.

⇒ To recognise instance variables in such case, method may use 'this' keyword with '.' operator.

* Local variables can't be accessed outside its method.

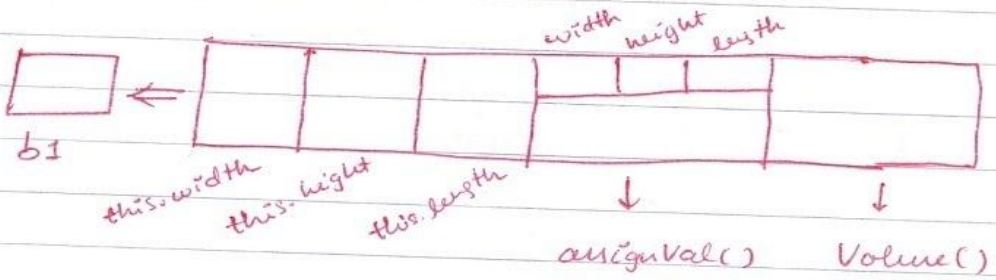
⇒ The 'this' keyword represents current object.

eg

```
class Box
{
    double width, height, length;

    void assignVal (double width, double height,
                    double length)
    {
        this.width = width;
        this.height = height;
        this.length = length;
    }

    void volume()
    {
        S.o.p ("volume" + width * height * length);
    }
}

class Demo
{
    p.s.v.m (String args[]);

    {
        Box b1 = new Box();

        b1.assignVal (10, 11, 12);

        b1.volume();
```

```
        Box b2 = new Box();
        b2.assignVal(20,20,20);
        b2.volume();
    }
}
```

o/p → volume = 1320.0
volume = 8000.0



Memory diagram ↑

Constructors ⇒

⇒ Constructor are as good as methods. They have the same name as that of the class and they don't have any return type. The constructors are called implicitely at the time of creation of objects. Hence, mostly they are used to assign values to a set of instance variables. The constructors can't be called explicitely.

⇒ We never show constructor in memory digram.

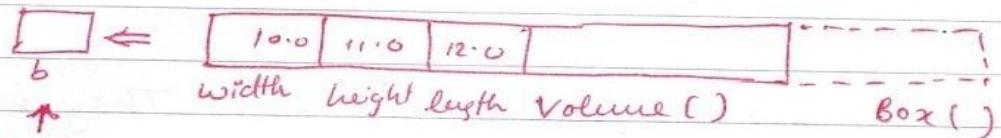
⇒ We can use 'this' keyword inside the body of constructors

Parameterless constructor :

```
class Box
{       double width, height, length;
        Box()
        {       width = w;
                height = h;
                length = l;
        }
        void volume()
        {
                S.o.p ("Volume = "+ l*b*h);
        }
}
class Demo
{       P.S.V.M (String args[])
        {
                Box b = new Box();
                b.volume();
        }
}
```

o/p → Volume = 1320

Memory allocation diagram →



object reference variable

constructor Box() will be the part of the current object

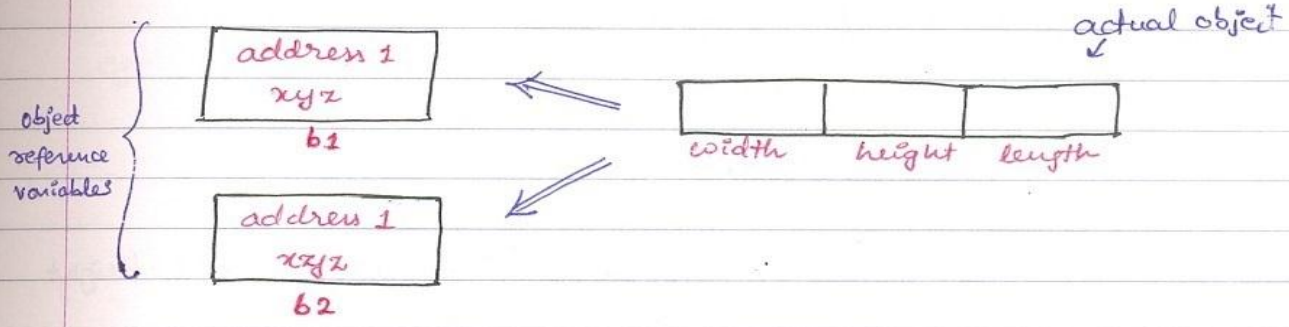
Parameterized constructor :

```
class Box
{
    double width, height, length ;
    Box ( double w, double h, double l)
    {
        width = w ;
        height = h ;
        length = l ;
    }
    void volume( )
    {
        S.o.p-("Volume=" + width * height * length) ;
    }
}

class Demo
{
    p.s.v.m (String args[ ])
    {
        Box b = new Box (10, 11, 12)
        b.volume( ) ;
    }
}
```

o/p → volume = 1320

Assigning Object Reference Variable

```
Box b1 = new Box();
Box b2 ;
b2 = b1 ;
```



→ Both b_1 and b_2 refer to the same object.

→ At first b_2 will hold NULL, then when b_2 is made equal to b_1, then the address of b_1 will be copied to b_2.

→ In the above cases, both object reference variables b_1 and b_2 refer to one and the same object of box class in memory.

Proof of the above statement

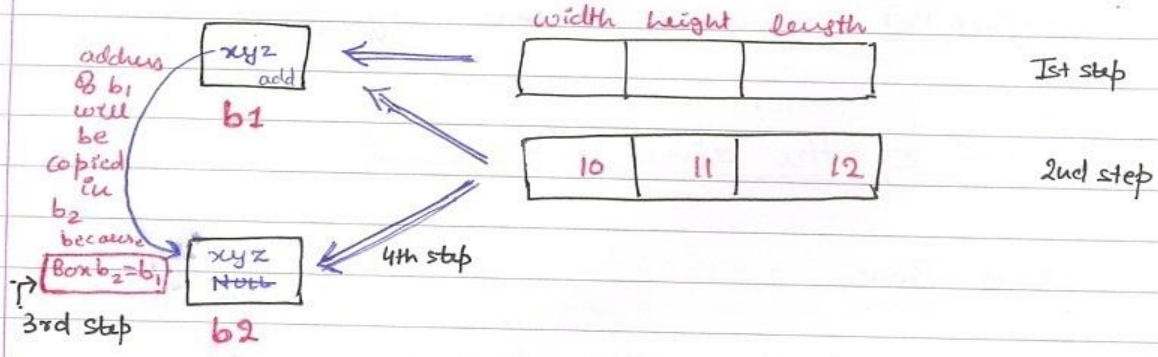
```
class Box
{
        double width, height, length ;
}
```

```
class Demo
{
    public static void main (String args[])
    {
        Box b1 = new Box();

        b1.width = 10;
        b1.height = 11;
        b1.length = 12;

        Box b2 = b1 ;

        S.o.p.("Volume = " + b1.width * b1.height
                * b1.length );

        S.o.p.( "Volume = " + b2.width * b2.height
                * b2.length );
```

change in the value of b_1 results in the same change in b_2. Hence proved.

```
        b1.width = 20
        S.o.p. ( b2.width);
    }
}
```

Memory diagram →



xyz ↓ address of object of Box class is stored in the reference variable of box class i.e. in b1

Method accepting object as a parameter

The object of any class can be treated as a variable whose data type is class. Hence, we can write methods accepting as well as returning objects.

eg. `void display (int num)`

`void display (Box num)`

`(Box b)`

↑ class ↑ obj

eg
```
class Box
{
        double width, height, length;
Constructor  Box ( double w, double h, double l )
        {
                width = w;
                height = h;
                length = l;
        }
}

class Myclass
{
        void volume ( Box b )
        {
                S.O.p. ( "Volume = " + b.width * b.height * b.length);
        }
}
```

In main meth or in any other method

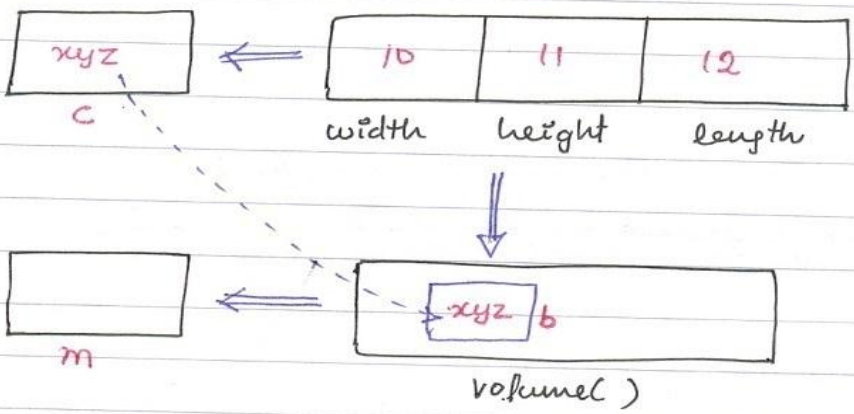
If we're making an object of a particular class having a parameterized constructor, then we can directly pass the parameters/values ~~in the~~ during formation of the object.

```
class Demo
{
    public static void main (String args[])
    {
        Box c = new Box (10, 11, 12);
        MyClass m = new MyClass ();

        m.volume (c);
    }
}
```

values are passed to object 'c'

we've passed 10, 11, 12 in the object 'c' of Box

Memory diagram ⇒



c → obj of Box in main method

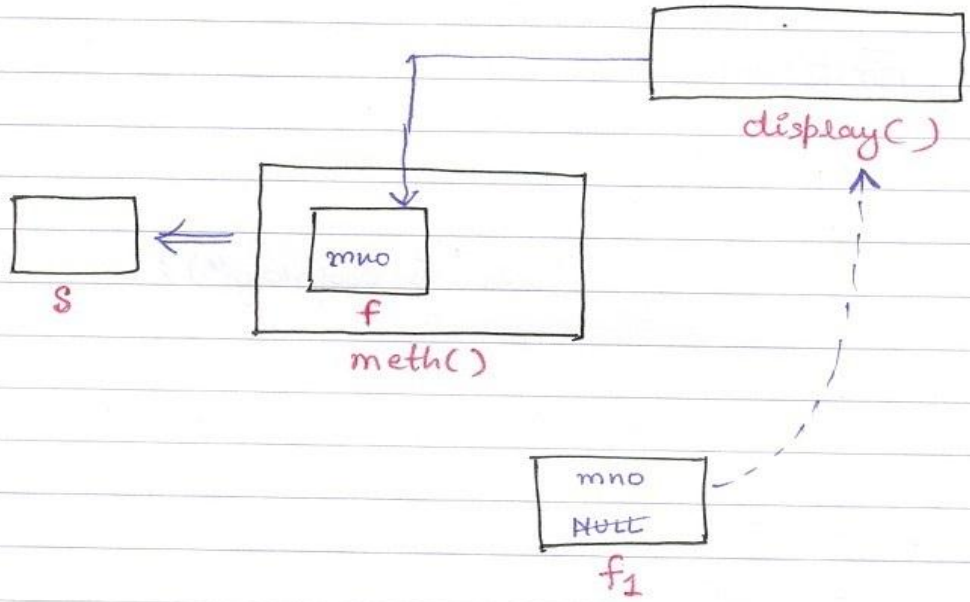
b → obj of Box in volume method

Method returning objects

```
class First
{
        void display ( )
        {
                S.o.p. ("This is display");
        }
}

class Second
{
        First meth ( )
        {
                First f = new First ( );
                return f ;
        }
}

class Demo
{
        private static void main (String args[ ])
        {
                Second s = new Second ( );
                First f1 ;
                f1 = s.meth ( );
                f1.display ( );
        }
}
```

o/p – This is display

Memory diagram



Program showing object acceptance and return together

- returns double of the values passed

```
class Box
{
        double width, height, length

        Box C double w, double h, double l)
        {
                width = w;
                height = h;
                length = l;
        }
}
```

```
class Myclass
{
        Box meth ( Box b )
        {
            Box c = new Box (b.width *2, b.height *2,
                             b.length *2 ) ;
            return c ;
        }
}

class Demo
{
    public static void main (String args[])
    {
            Box b1 = new Box (10, 11, 12) ;

            MyClass m = new MyClass ( ) ;

            Box b2 = m.meth ( b1 ) ;

            System.out.println ( b1.width + " " + b1.height + " "
                                 + b1.length )
            System.out.println ( b2.width + " " + b2.height + " "
                                 + b2.length )
    }
}
```

It can be done in another way also which we can do without making the new object.

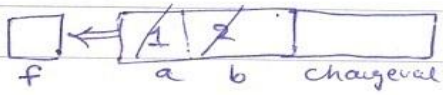
↓

Passing parameters to methods by value & by reference

The variables of primitive data types are passed by value, whereas objects are passed by reference.

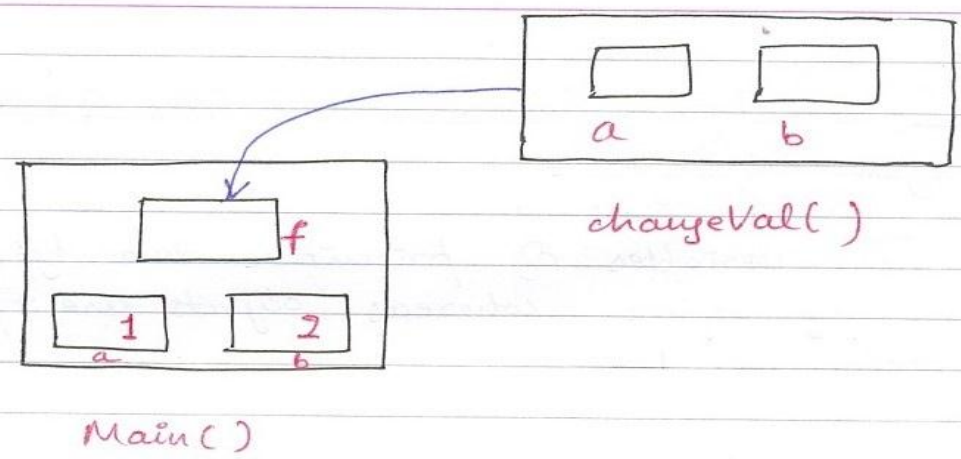
By value ⇒

```
class First
{
    void changeVal ( Int a , Iut b)
    {
        a= a+100 ;
        b= b+100;
        S.o.p. ( " Iuside changeVal a = " +a + " b = " +b);
    }
}

class Demo
{
    public static void main ( String args[ ])
    {
        iut a=1 ;
        iut b=2 ;
        sop ( "Before changeVal a= " +a+ " b= " +b);

        First f = new First ( );
        f. changeVal (a, b );
        sop ( "After changeVal a= "+a + " b= " +b );
    }
}
```



o/p →

```
Before changeVal    a = 1    b = 2
Inside changeVal    a = 101  b = 102
After changeVal     a = 1    b = 2
```

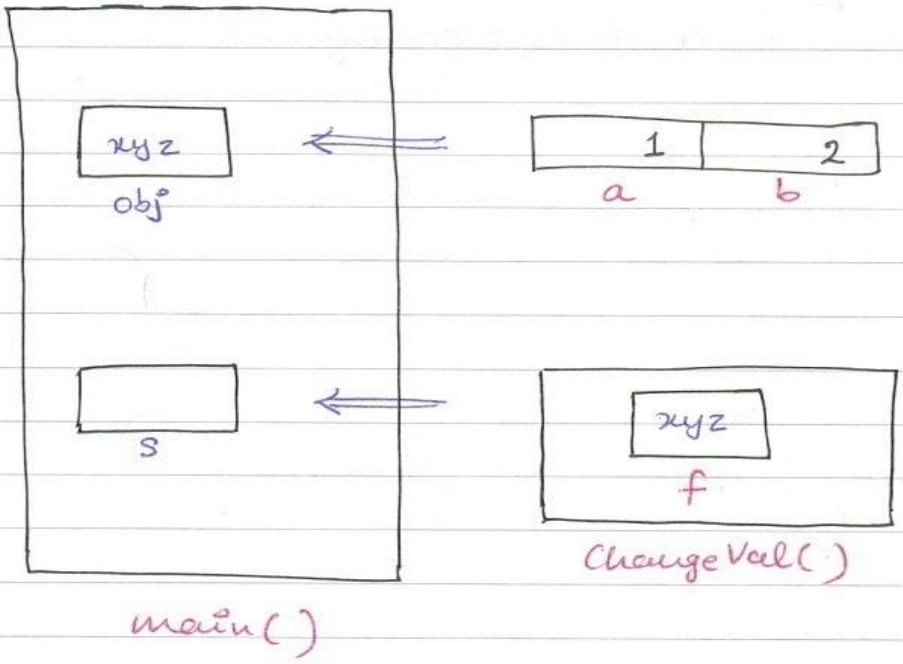


By Reference ⇒

```
class First
{
        int a=1 ;
        int b=2 ;
}

class Second
{
        void changeVal ( First f )
        {
            f.a = f.a +100
            f.b = f.b +100
        }
}

class Demo
{
      public static void main ( String args[ ])
      {
          First obj = new First( );
          sop ("Before changeVal a = " +obj.a "b = "+obj.b)
          Second s = new Second ( );
          s.changeVal (obj)
          sop (" After changeVal a= "+a "b = "+b)
      }
}
```

o/p.

Before changeVal obj.a = 1 obj.b = 2
After changeVal obj.a = 101 obj.b = 102



Method overloading

When a class contains multiple methods sharing the same name but having different parameters - structures, it becomes method-overloading.

During runtime, the multiple versions of methods are found in memory and the proper version is choosen depending on parameters passed to methods.

This is one of the types of polymorphism provided by java.

The return type of method does not play any role in overloading.

Eg

```
class First
{
    void display()
    {
        Sop("This is display");
    }

    void display(int num)
    {
        Sop("square = " + num * num);
    }
```

```
    void display (int num1, int num2)
    {
        Sop("Mult = " + num1 * num2);
    }
}

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

        f.display(10);
        f.display();
        f.display(10, 11);

    }
}
```

o/p → square = 100
This is display
Mult = 110

Constructor Overloading

When a class contains multiple constructors with different parameter structures, it becomes constructor overloading.

During runtime, the proper version of constructor is choosen depending on parameters passed to it.

This is one of the types of polymorphism provided by java.

eg

```
class Box
{
    double heigh, width, length;

    Box( double ~~height~~ h, double l, double w)
    {
        width = w;
        height = h;
        length = l;
    }

    Box ( )
    {
        width = height = length = 1 ;
    }
```

* Compile-time polymorphism is a wrong concept because it doesn't happen actually

```
    Box ( double side)
    {
        width = height = length = side ;
    }
}

class Demo
{
    public static void main ( String args[])
    {
        Box b = new Box ( ) ;
        Box b = new Box ( 10, 11, 12) ;
        Box b = new Box (10) ;

        System.out.print

        Box b ;

        b = new Box ( ) ;
        b.volume ( ) ;

        b = new Box ( 10, 11, 12) ;
        b.volume ;

        b = new Box ( 10) ;
        b.volume ;
    }

}
```

Static - keyword

⇒ The static members of a class are loaded in memory when we try to use class for the first time during execution.

⇒ It means the static members are loaded in memory. before creation of any of the objects of that class.

⇒ It means the static members can be accessed without object by using class name with dot operator (.).

⇒ One and the same copy of static members in memory is shared by all objects of that class, and accessed without using a particular object.

⇒ The static methods and blocks cannot access non-static members.

⇒ The static method cannot use this keyword.

⇒ Objects of classes can access the static members but the static methods can't access the non-static members, because the static members are loaded in the memory before the creation of objects.

Volume = density x mass

Since these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred to as class variables & class methods in order to distinguish them from instance variables & instance methods

```
class First
{
        static int a = 100;

        static void meth ( )
        {
                System.out.println ("This is static method");
         }

        static
        {
            System.out.println ("This is static block ");
         }
}

class Demo
{
        static
        {
            System.out.println ("This is static block in
                                 Demo class");
        }

        public static void main ( String args[])
        {
                System.out.println ("Hello");
                System.out.println (First.a);
                First.meth ( );

                First f1 = new First();
                First f2 = new First();

                System.out.println (f1.a);
                System.out.println (f2.a);
```

ic variable is accessed →

atic method is accessed →

```
        f1.a = 200;
        System.out.println(f2.a);
        System.out.println(First.a);

        f1.meth();
        f2.meth();
    }
}
```

o/p ⇒

```
This is static block in Demo class
Hello
This is static block in First class
100
This is static method 100
100
100
200
200
This is static method 200
This is static method 200
```

final - Keyword

All the keywords in Java are declared in small case letter.

① The 'final' variables are constants of java and they should be assigned values at the time of declaration itself.

There values can be accessed throughout code but can't be changed.

② The 'final' methods cannot be overriden

③ The 'final' classes cannot be inherited.

```
class First
{
        final int i=100;
}

class Demo
{
    public static void main (String args[])
    {
            First f = new First( 2);
            System.out.println( f.i);

            // f.i = 200 → if uncomented, will raise compile time error.
    }
```

```
First.
{
    int i= 200;
}
First ( int num)
{
    i= num ;
}
```

Constructors can also contain final variables or rather we can assign value (final) in the constructors. The value can be assigned differently in the constructors.

Nested Classes

3 things to notice about Nesting of classes →

(1) The members of outer class can be accessed in the body of inner class without creating object of outer class.

(2) ~~The ne~~ To access the members of inner class in the body of outer class, we need to create object of inner class.

(3) The scope of inner class lies within the body of outer class.
It means inner class cannot be accessed directly outside the body of outer class.

```
class Outer
{
    int a = 100;
    Inner obj = new Inner();          // another method of accessing showa() of inner

    void access()
    {
        Inner i = new Inner();
        i.showa();
    }                                 // showa() method has been called from the method of outer class which can be directly called by making object of inner class.

    class Inner
    {
        void showa()
        {
            System.out.println("a=" + a);
        }
    }
}

class Demo
{
    public static void main(String args[])
    {
        Outer o = new Outer();
        o.acces();

        o.obj.showa();
    }
}
```

object of outer class ← o.obj → object of inner class

```
        Outer.Inner oi = new Outer().new Inner();
        oi.showa();
    }
}
```

Inheritance

⇒ Inheritance helps to carry functionality of one class to other class.

⇒ The key-word 'extends' is used for this purpose.

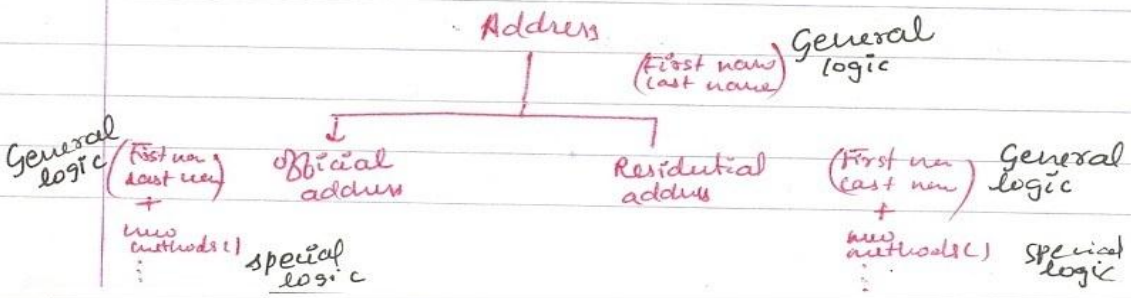
⇒ The extended class is known as super class and extending class is known as sub-class.

⇒ The inheritance creates hierarchial classification in class library where the super classes represent 'General logic' and the sub classes represent 'special logic'.

Though inherited superclasses can be used independently by creating their objects.

eg: class 1 → var + methods
class 2 → ↓ + new methods

```
class Class2 extends Class1
{
=
}
```



⇒ Simple inheritance

```
class First
{
    int a;
    void showab()
    {
        System.out.println("a="+a+"b="+b);
    }
}

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

        s.a = 100;
        s.b = 200;
        s.showab();
        s.showa();
}

class Second extends First
{
    int b;
    void showb()
    {
        System.out.println("a="+a+"b="+b);
    }
}
```

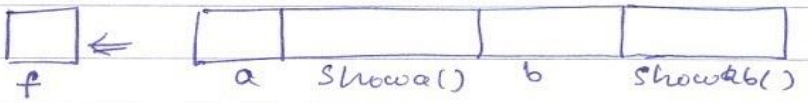
Member access in inheritance

⇒ The private members of Super class are never inherited to sub class.

⇒ The object reference variable of super class can refer object of a sub-class.

⇓

First f = new Second();



f.a ✓
f.showa() ✓
f.b ✗ } not possible
f.showab() ✗ }

advantage: in polymorphism.

f (object of First) can only hold the addres of second. From f we can access only those many objects which are defined in the class and the inherited ones.

⇒ The object reference variable of Super class can refer object of a sub class but in that object f refers those members that are defined in Super class and are inherited to sub class.

refer] can hold address

```
class First
{
    int a;
    void showa()
    {
        System.out.println("a=" +a);
    }
}

class Second extends First
{
    int b;
    void showab()
    {
        System.out.println("a=" +a+ "b=" +b);
    }
}

class Demo
{
    public static void main(String args[])
    {
        First f = new Second();
        f.a =100;
        f.showa();

        f.b =100;   → If uncomented, will raise compile time error
        f.showab();
    }
}
```

→ The parameters in the super call must match the order & type of the instance variable declared in the superclass.

→ only used in subclass constructor

→ call to superclass constructor must appear as the first statement within the subclass constructor

Super keyword

I. To call the constructor of super class →

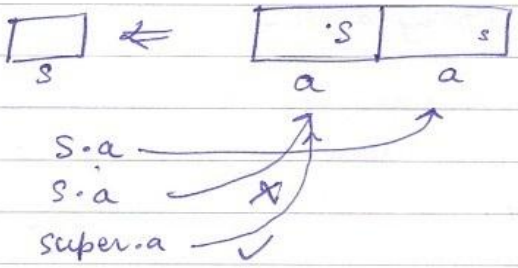
⇒ We can use super keyword in the constructor of sub-class to call the constructor of super class, as shown in following example.

```
class Box
{
    double width, height, length;

    Box( double w, double h, double l)
    {
        width = w;
        height = h;
        length = l;
    }

    Void volume ()
    {
        S.o.p ("Volume=" height * width * length);
    }
}

class MyBox extends Box
{
    double density;
    MyBox (double w, double h, double l)double d
    {  super(w, h, l) ;
       density = d;
    }
}
```

```
void mass
{
    S.o.p ("Mass = "+ width * height * length * density);
}
}

class Demo
{
    p.s.v.m (String args[])
    {
        MyBox m = new MyBox (10,10,10,5);
        m.volume();
        m.mass();
    }
}
```

II. To refer member inherited from super class →

```
class First  →  int a;
   ↓
class Second →  int a;

Second s = new Second();
```



⇒ ~~class~~ When Superclass and subclass hold members of same name, the object reference variable of subclass always refers member defined in subclass and not the inherited member.

⇒ To refer inherited member we need to use 'super - keyword' with dot (.) operator.

```
class First
{
    int a = 100;
}

class Second extends First
{
    int a = 200;

    void accen ()
    {
        System.out.println (super.a);
    }
}
class Demo
{
    P.S.V.M (String args[])
    {
        Second s = new Second();
        System.out.println (s.a);
        s.access();
    }
}
```

o/p : 200
100

Multilevel hierarchy of inheritance

The subclass of any class can again be extended to create its sub-class. This forms a multi-level hierarchy in the class-library.

class First → int a, showa()
↓
" Second → int b, showab()
↓
" Third → int c, showabc()

First f = new Third();

```
class First
{
    int a;

    void showa()
    {
        S.O.P("a="+a);
    }
}
```

super should be the first statement in the constructor of subclass.

```
class Second extends First
{
        int b;

        void showab()
        {
            S.o.p("a="+a+"b="+b);
        }
}
class Third extends Second
{
        int c;
        void showabc()
        {
            S.o.p("a="+a+"b="+b+"c="+c);
        }
}

class Demo
{
        p.s.v.m(String args[])
        {
            Third t = new Third();
            t.a = 100;
            t.b = 200;
            t.c = 300;
            t.showa();
            t.showab();
            t.showabc();
        }
}
```

How the constructors are called in multi-level hierarchy?

⇒ In multi-level hierarchy when we try to create object of any class, the constructors are loaded in memory from that of the class at top level to current class sequentially.

⇒ If all the constructors in super class are parameterised, we need to write 'super' keyword in the constructor of sub-class and we need to pass it parameters that are matching with the parameter structure of any of the constructors of super class.

```
class First
{
    First()
    {
        System.out.println("Constructor of First class");
    }
}

class Second extends First
{
    Second()
    {
        System.out.println("Constructor of Second class");
    }
}
```

```
class Third extends Second
{
    Third()
    {
        System.out.println("Constructor of
                            third class");
    }
}

class Demo
{
    p.s.v.m (String args[])
    {
        new Third();
    }
}
```

o/p

Constructor of First class
" " Second "
" " Third "

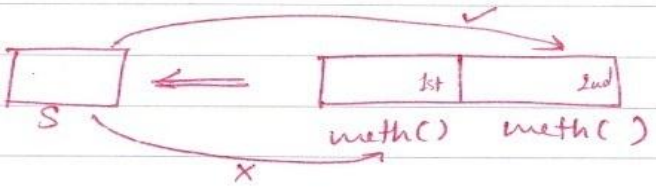
⇒ If we're having a method in the class third, then we can access it by
new Third.meth();
Now, If we'll write the same thing again then another object will be created of the same class & the constructors will be loaded in the memory again.

Method overriding

class First → void meth()

↓

class Second → void meth()



S.meth();

⇒ When super class and sub-class hold methods of same name and having same parameter structure, it becomes method overriding.

⇒ In such case, the object reference variable of sub-class refers method defined in sub class and not the ~~overriden~~ method defined in super-class.

⇒ To refer method (inherited), we need to use 'super - keyword' with dot operator.

```
class First
{
    void meth()
    {
        S.o.p.("meth defined in First class");
    }
}
```

```
class Second extends First
{
    void meth()
    {
        S.o.p("meth in 2nd class");
    }

    void access()
    {
        super.meth();
    }
}

class Demo
{
    p.s.v.m (String args[])
    {
        Second s = new Second();
        s.meth();
        s.acess();
    }
}
```

D.M.D. (Dynamic Method Dispatch)

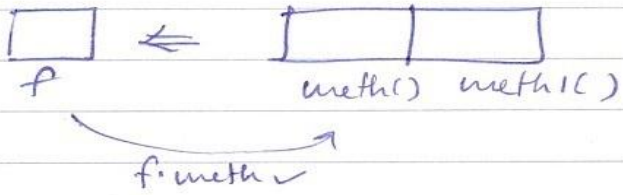
— Runtime polymorphism of java

— method overriding is resolved at runtime instead of compile-time

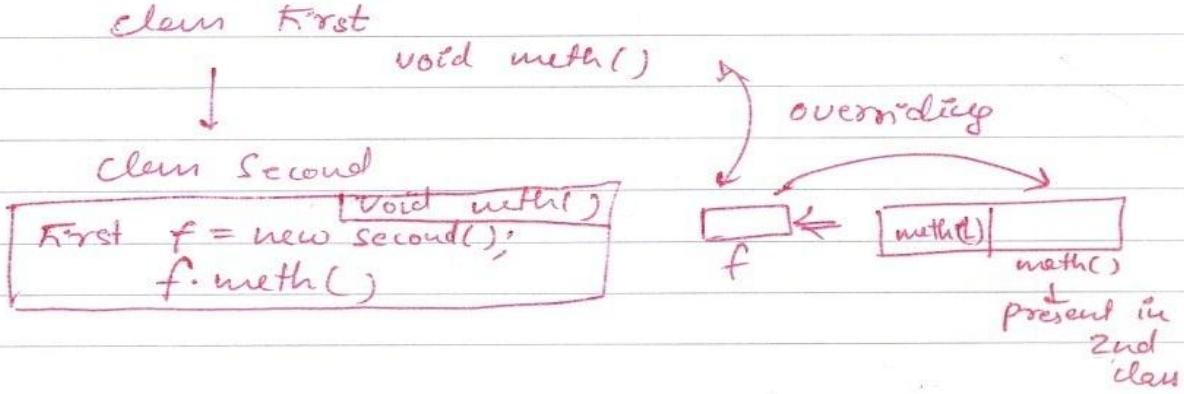
```
class First
      |        void meth();
      |
class Second
              void meth1();
```

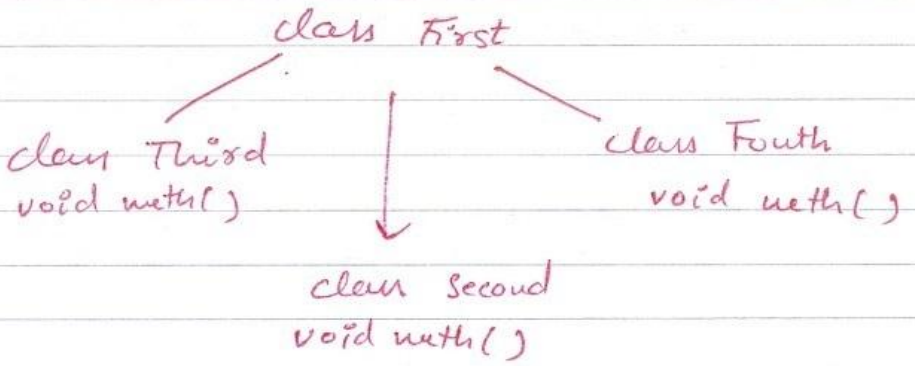
⇒ We can write

```
First f = new Second();
```



In case of overriding, the method present in the object is accessed by the reference variable.






```
First f;

f = new Second();
f.meth();

f = new Third();
f.meth()

f = new Fourth();
f.meth();
```

eg we can have Demo class → as following also ↓

~~Def~~ The object reference variable of super class can refer object of a sub-class but in that object, it refers those members that are defined in super class and are inherited to subclass.

⇒ When sub-class overrides method of super-class, the object reference variable of super-class refers method overrided in sub-class.

⇒ Hence, by creating multiple subclasses of a super-class, we can make object reference variable of super class refer methods overriden in sub-classes.

⇔ This is Run-time polymorphism of Java and is known as DMD or DMI.

eg

```
class First
{
    void meth()
    {
        S.o.p("This is first method");
    }
}

class Second extends First
{
    void meth()
    {
        S.o.p("This is 2nd method");
    }
}
```

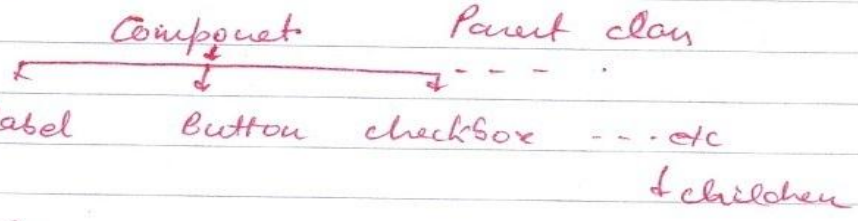
One method which gives diff o/p for diff i/p provided — Polymorphism

```
class Third extends First
{
    void meth()
    {
        S.o.p.("This is third method");
    }
}

class Fourth extends First
{
    Void meth()
    {
        S.o.p("This is fourth method");
    }
}

class MyClass
{
    void mymeth(First f)
    {
        f.meth();
    }
}
class Demo
{
    P.S.V.m(String args[])
    {
        First f = New First();
        :
        Fourth ft = New Fourth();
        MyClass P = new P();
        P.meth(f);
        P.meth(s);
        P.meth(ft);
    }
}
```

Q8 Polymorphism in Java library

In case, we want to create an applet, then for buttons, scrolls, lists, etc. we've inbuilt classes in the Java library like Label, Checkbox, etc. ~~But~~ We can access these by calling the constructor present in the particular class. But, it will not be shown on the applet window.

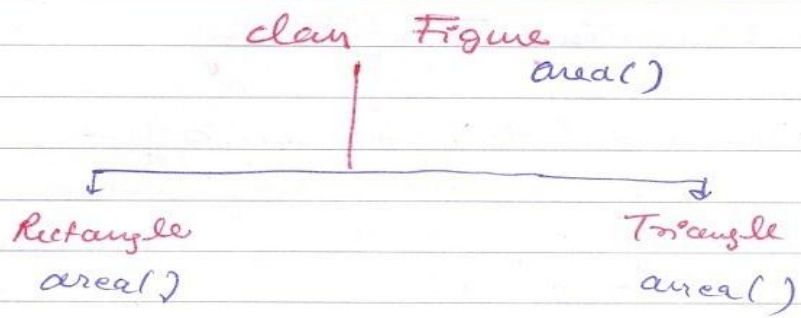
Now, Java has provided a method 'add' for adding the particular class on the applet.



Now,

add method has return type & ~~accepts~~ as object & it accepts object also.

In Java class library, there are numerous polymorphism defined.



→ Figure f ;

f = new Rectangle (_) ;
f. ~~meth~~ area ( ); ← method of Rect will be called

f = new Triangle (_) ;
f. area ( );

eg →

```
class Figure
{
    double dim1 , dim2 ;
    Figure( double d1 , double d2)
    {
        dim1 = d1 ;
        dim2 = d2 ;
    }
}   Figure ( double d1 )
    {
        dim1 = d1 ;
    }
    void area ( )
    {
        s.o.p ("This is area in Figure class");
    }
}.
```

```
class Rectangle extends Figure
{
    Rectangle ( double width, double length )
    {
        super ( width , length );
    }

    void area( )
    {
        S.o.p (" Area of rectangle = " + dim1 * dim2);
    }
}
class Triangle extends Figure
{
    Triangle (double base, double height)
    {
        super ( base, height )
    }
    void area ( )
    {
        S.o.p. ( " Area of triangle = " + 0.5*dim1 * dim2);
    }
}
class Circle extends Figure
{
    Circle ( double radius )
    {
        super ( radius )
    }
    void area( )
    {
        S.o.p ( " Area of Circle = " + 3.14 . dim1 * dim1);
    }
}
```

Pass values to the predefined constructors ⇒ concept of polymorphism

Make obj & pass to the constructor

```
class Demo
{
    P.s.v.m(String args[])
    {
        ~~Figure f~~ ;
        Rectangle rect = new Rectangle(10,20)
        Triangle tri = new Triangle(10,20)
        Circle c = new Circle(10)

        ~~of class A~~
        MyClass m = new MyClass();
        m.area(~~10,20~~ rect);
        m.area(~~10,20~~ tri);
        m.area(~~10~~ c);
    }
}
class MyClass
{
    void myArea(figure f)
    {
        f.area()
    }
}
```

Create a method MyVolume() showing polymorphism.

```
class Figure
{
    double radius, height, side

    Figure ( double r, double h)
    {
        radius = r;
        height = h;
    }

    Figure ( double side )
    {
        side = s;
    }

    void myVolume ( )
    {
        Soop.("This is volume in figure class");
    }
}

class Cylinder extends Figure
{
    Cylinder ( double radius, double height)
    {
        super ( radius, height);
    }
    void volume ( )
    {
        Soop ("Volume of cylinder = "+ 3.14*radius*radius*
                                                height);
    }
```

```
class Cone extends Figure
{
    Cone(double radius, double height)
    {
        super(radius, height);
    }
}

class Sphere extends Figure
{
    Sphere(double radius)

    void myvolume()
    {
        S.o.p("Volume of cone = "+ 1.0/3.0 *3.14* radius*radius*height)
    }
}
class Sphere extends Figure
{
    Sphere(double radius) ~~double~~
    {
        Super(radius);
    }

    void myvolume()
    {
        S.o.p("Volume of sphere = "+ 4.0/3.0 *3.14*radius*radius*radius)
    }
}
```

```
class Cube extends Figure
{
    Cube (double side)
    {
        super (side);
    }

    void myvolume()
    {
        S.op ("Volume of cube=" + side*side*side);
    }
}

class Demo
{
    P.S.V.m (String args[])
    {
        Cylinder cy = new Cylinder(10,20);
        Cone     co = new Cone(10,20);
        Sphere   sp = new Sphere(10);
        Cube     cu = new Cube(10);

        MyClass m = new MyClass();
        m.area(10,20);   m.myvolume(cy);
        m.area(10,20);   m.myvolume(co);
        m.area(10);      m.myvolume(sp);
    }
}

class MyClass
{
    void myArea (Figure f)
    {
        f.area()
    }
}
```

Abstract method

- Incomplete method
- without body
- not instantiated
- purely used for inheritance

```
abstract class First
{
    abstract void area();
}
```

⇒ Abstract classes

⇒ We can write method without body in java, such method is known as an abstract method.

⇒ Such method is an incomplete method, and when we put such method in any class, that class also becomes an incomplete class and needs to be declared as an abstract class.

⇒ Once, we declare any class as an abstract class, we cannot create its object.

⇒ The abstract classes are purely used for inheritance purpose.

⇒ When any class extends an abstract class, it should override all abstract methods of its super class.

* Without Overriding, DMD isn't possible

We can make object reference variable of the abstract class but we can't instantiate with object.

⇒ Otherwise, it should be declared as 'Abstract'.

⇒ The abstract classes may contain concrete methods and constructors

⇒ The abstract classes are ultimately used to achieve DMD alongwith data hiding.

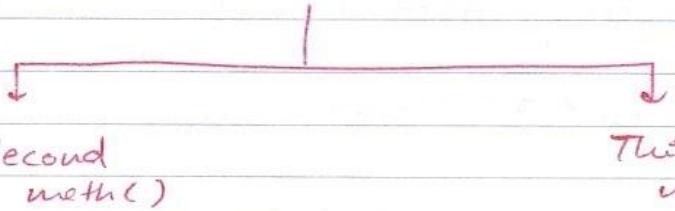
Eg

```
abstract class First
{
    int a;

    First (int num)
    {
        a = num;
    }
    abstract void meth1();
    abstract void meth2();

    void meth3()
    {
        System.out.println("This is meth3");
    }
}

class Second extends First
{
        int b;

        Second (int num1, int num2)
```

```
{ s
    super(num1);
    b=num2;
}

void meth
{
    S.o.p("This is meth");
}

void meth2()
{
    S.op ("This is meth2");
}

void showab()
{
    S.o.p.("a="+at "b="+b);
}
}

class Demo
{
    p.s.v.m (String args[])
    {
        Second s = new Second(10,11);
        s.meth1();
        S.meth2();
        s.meth3();
        S.showab();
    }
}
```

D.M.D. using Abstract classes

eg.
```
abstract class First
{
    abstract void meth()
```




```
First f = new first();
```
✗ cannot make object

```
First f ;
```
✓ object reference variable can be formed which will hold the address of objects of second & Third.

```
f = new Second();
f.meth();
```

```
f = new Third();
f.meth();
```
⇐ At run-time, we'll get the o/p

⇒ Though we cannot create an object of abstract class, we can create its object reference variable.

⇒ This variable can hold the address of object of a class which ~~impl~~ extends it.

⇒ Hence, we can achieve DMD as shown in following example →

↓

```
abstract class First
{
        abstract void meth();
}

class Second extends First
{
        void meth()
        {
            S.o.p. ("This meth is in Second class")
        }
}

class Third extends First
{
        void meth()
        {
            S.o.p. ("This meth is in Third class")
        }
}

class Myclass
{
        void mymeth ( First f)
        {
            f.meth();
        }
}
```

```
class Demo
{
    p.s.v.m (String args[])
    {
        Second s = new Second();
        Third t = new Third();

        MyClass m = new Myclass();
        m.myMeth(s);
        m.myMeth(t);
    }
}
```

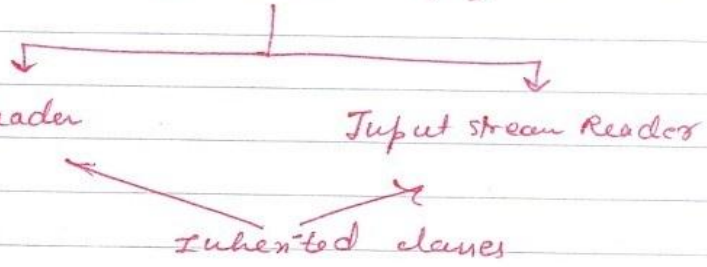
Eg from java library

⇓

Reading data from file to console

```
br = new BufferedReader (Reader obj)
```

(Reader → abstract class)



for DMD, java provides us with the abstract classes & methods.

Interfaces

Defining an interface :-

```
interface InterfaceName
{
    returntype methodName1 (parameter list);
    returntype methodName2 (  "  );
        :
        :
}
```

Implementing an Interface :-

```
class ClassName implements InterfaceName
{
        "
}
```

⇒ Interfaces are as good as classes.

⇒ All methods in an interface are public and abstract.

⇒ Interface cannot have concrete methods and constructors.

⇒ We cannot create an object of interface.

⇒ The interfaces are used for inheritance purpose, by implementing them in a class.

⇒ When any class implements an interface, it must override all methods of ~~its~~ that interface.

⇒ Otherwise, it should be declared as abstract.

⇒ The interfaces are ultimately used for achieve dmd, data hiding alongwith multiple inheritance.

⇒ We can create object reference variable.

⇒ Extending interfaces

⇒ Like classes, interfaces can also be extended. i.e., an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses.

```
interface name2 extends name1
{       body of name2
}
```

⇒ We can also combine several interfaces together into a single one

```
interface ItemConstants
{       int code = 1001;
        String name = "Fan";
}
interface ItemMethods
{       void display();
}
```

```
interface Item extends ItemConstants, ItemMethods
```

Eg

```
interface MyInter
{
        void meth1();
        void meth2();
}

class Myclass implements MyInter
{
        public void ~~my~~meth1()
        {
                S.o.p("This is meth1");
        }

        public void meth2()
        {
                S.o.p("This is meth2");
        }
}

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

                m.meth1();
                m.meth2();
        }
}
```

D.M.D. using Interfaces

⇒ Though we cannot create an object of an interface, we can create its object reference variable. This variable can hold the address of object of a class which implements.

⇒ Hence, we can achieve effect of D.M.D as shown ↴

```
interface MyInter
{
        void meth();
}

class Class1 implements MyInter
{
        public void meth()
        {
                System.out.println("meth in Class1")
        }
}

class Class2 implements MyInter
{
        public void meth()
        {       S.o.p. ("meth in class2")
        }
}
```

```
class MyClass
{
    void myMeth (MyInter m)
    {
        m.meth( );
    }
}

class Demo
{
    p.s.v.m ( String args[ ])
    {

        Class1 c1 = new Class1( );
        Class2 c2 = new Class2( );

        MyClass m = new MyClass( );

        m.myMeth (c1);
        m.myMeth (c2);
    }
}
```

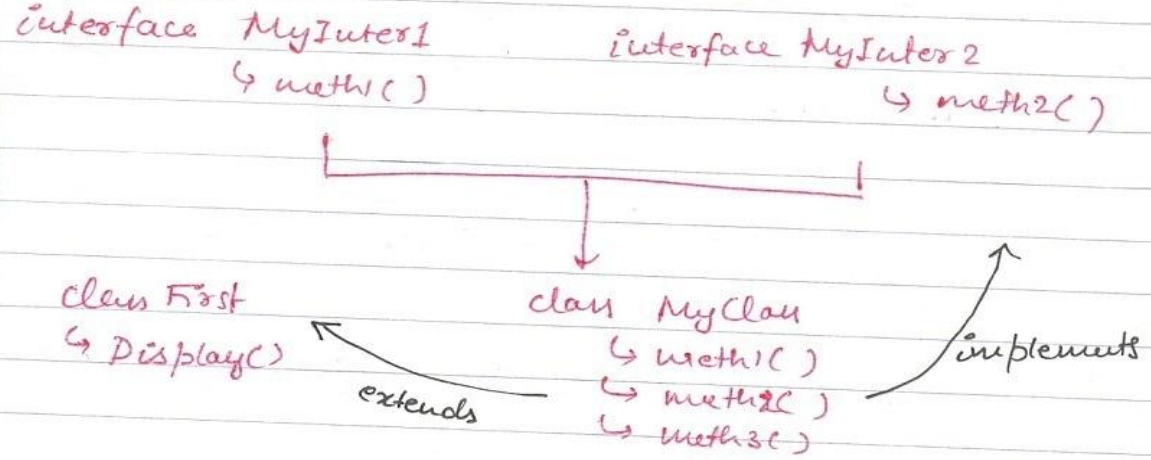
eg. from Java library

- For setting layout of the window, Java has given a method setLayout which accepts the reference variable of LayoutManage which is an interface.

void setLayout (LayoutManage obj)

These 4 classes ~~are~~ implements LayoutManage and for the creation of objects, Java has given ~~can~~ inbuilt constructors in them.

Multiple Inheritance using Interfaces



⇒ A class can implement multiple interfaces, but in such case it should override all methods of all implemented interfaces.

⇒ This is how multiple inheritance is implemented in java.

⇒ A class may ~~imple~~ extend a class and implement multiple interfaces.

```
interface MyInter1
{
        void meth1();
}

interface MyInter2
{
        void meth2();
}

class First
{
        void display()
        {
                S.o.p. ("This is display");
        }
}

class MyClass extends First implements MyInter1, MyInter2
{
        void meth1()
        {
                S.o.p ("This is meth1");
        }
        void meth2()
        {
                S.o.p ("This is meth2");
        }
        void meth3()
        {
                S.o.p ("This is meth 3");
        }
}
```

```
class Demo
{
    public static void main (String args[])
    {
        MyClass m = new MyClass();
        m.meth1();
        m.meth2();
        m.meth3();
        m.display();

        MyInter m1 = new MyClass(); m
        m1.meth1();
        // m1.meth2();
        // m1.meth3();
        // m1.display();

        MyInter m2 = m;
        //m2.meth1();
        m2.meth2();
        // m2.meth3();
        // m2.display();

        MyInter f = new First();
        //f.meth1();
        // f.meth2();
        // f.meth3();
        f.display();
    }
}
```

Variables in an Interfaces

⇒ The variables in an interface are public, static and final.

⇒ They are by default public static & final & hence there is no need to write these keywords.

⇒ Variable must be assigned at the time of declaration only.

```
interface MyInter
{
    int a = 100;
}

class MyClass implements MyInter
{
}

class Demo
{
    public static void main (String args[])
    {
        S.o.p (MyInter.a);
        S.o.p (MyClass.a);

        // MyClass.a = 200 → If uncommented will raise compile time error
    }
}
```

o/p – 100
100

Using 'final' to restrict inheritance

⇒ The final methods cannot be overridden and final classes cannot be extended.

```
final class First
{
    void meth() {}
}

class Second extends First
{
    void meth(){}
}

class Demo
{
    psvm
    {
    }
}
```

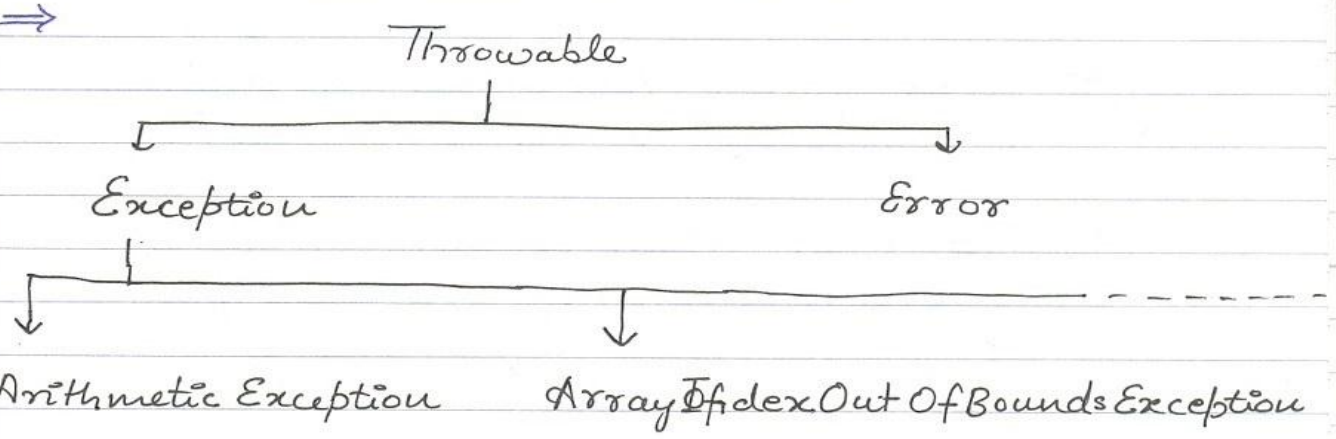
```
class First
{
    final void meth(){}
}

class Second extends First
{
    void
}
```

Exception handling = helping hand during run-time

⇒ Exception is an abnormal condition, raised in java program due to breaking of any fundamental rule of java.

⇒ Java provides following class hierarchy for exceptions.

⇒



⇒ In above hierarchy, the class 'Exception' represents runtime abnormal conditions that can be handled through code.

⇒ Whereas, the class Error, represents 'system errors' that can't be handled through code.

⇒ Technically, every exception, raised in java program is an object of any of the subclasses of RuntimeException class or Exception class.

Uncaught Exception

⇒ When we don't handle exception, it is passed to java's default exception handler.

⇒ This handler prints the description of exception on console, and program is terminated on the same line.

⇒ This abrupt termination of program leads user to the confusion.

⇒ To avoid this, we need to write exception handling code in java program.

⇒
```
class Demo
{
    P.S.V.M ( String args)
    {
        int m=100, n=0;
        int k = m/n;
        System.out.println ("This will not be printed");
    }
}
```

o/p ⇒

java.lang.ArithmeticException : / by zero
at Demo.main (Demo.java : 6)

1- try - catch block -

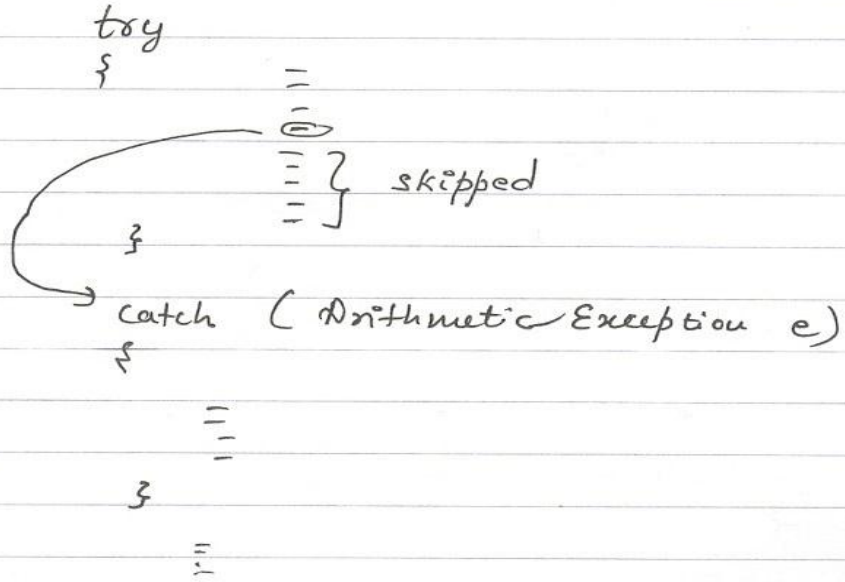
⇒ The code in which, there is possibility of exception is put in try block.

⇒ The code in try block is considered under observation when an exception is raised in try block, the control of program is paused to ~~d~~ catch block ~~sequentially~~ following that try block

⇒ The catch block accepts object of corresponding exception class and holds code necessary to handle that exception.

⇒ After the execution of catch block, the code after catch block is executed.

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



```
class Demo
{
    public static void main (String args[])
    {

      try
      {
          int m = 100 , n = Integer.parseInt (args[0]);
          int k = m/n ;
          System.out.println ("k=" + k);
      }
      catch (ArithmeticException e)
      {
              S.o.p ("can't divide by zero");
      }
      S.o.p ("After catch");
}
```

2- Multiple catch blocks

```
try
{
        =
        =
        =
}
catch ( Arithmetic e )
{
   =
   -
}
catch ( Ar ——— e}
{
   =
   -
}
-
```

⇒ The code in which, there is possibility of exception is put in a try block followed by multiple catch blocks.

⇒ So, when an exception is raised in try block, the control of the program is passed to all catch blocks sequentially & wherever exception object is matched, the corresponding catch block is executed.

eg

```
class Demo
{
    public static void main (String args[])
    {
      try
      {
          int m=100, n = Integer.parseInt (args[0]);
          int k = m/n;
          System.out.println ("k=" +k);
      }
      catch (ArithmeticException e)
      {
          S.o.p ("Can't divide by zero");
      }
      catch (ArrayIndexOutOfBoundsException e)
      {
          S.o.p ("Invalid array Index");
      }
      catch (NumberFormatException e)
      {
          S.op ("Invalid Input");
      }
      S.o.p ("After catch");
    }
}
```

o/p :

```
_____ java Demo 2
k=50
After catch

_____ java Demo 0
can't divide by zero
After catch

_____ java Demo
Invalid array index
After catch
```

Eg - 8 null pointer exception

```
class Box
{
    double width, hight, length;

    Box (double w, double h, double l)
    {
        width = w;
        height = h;
        length = l;
    }
}
class MyClass
{
    void volume ( Box b )
    {
        S.O.P ( b.width * b.height * b.length);
    }
}
```

```
class Demo
{
    P.s.v.m (String args[])
    {
        Box c = null;
        new Myclass().volume(c);
    }
}
```

o/p →

```
Exception in thread "main" java.lang.NullPointerException
    at Myclass.volume(Demo.java:17)
    at Demo.main(Demo.java:26)
```

Nested try catch

⇒ In case of nested try, if an exception is raised above inner try or below inner catch, it is passed to outer catch for handling.

⇒ But if an exception is raised in inner try, it is passed to ~~outer~~ inner catch for handling.

⇒ But if inner catch can't handle it, it becomes an uncaught exception of outer try and is passed to outer catch for handling

```
class Demo
{
    public static void main(String args[])
    {
        try
        {
            System.out.println("Start of outer try");

            try
            {
                System.out.println("Start of Inner try");
                int m = 100, n = Integer.parseInt(args[0]);
                int k = m/n;
                System.out.println("k = "+k);
                System..out.println("End of Inner try");
            }
            catch (ArithmeticException e)
            {
                System.out.println("Can't divide by zero");
            }

            Catch System.out.println("End of outer try");
        }

        catch (ArrayIndexOutOfBoundsException ae)
        {
            System.out.println("After outer catch");
        }
}
```

o/p : java Demo | ArrayIndexOutOfBoundsExcep.

Start of outer try
Start of Inner try
outer catch

java Demo | ArithmeticException

Start of outer try
Start of inner try
can't divide by zero
End of outer try

java Demo 4

Start of outer try
Start of inner try
k = 25
End of inner try
End of outer try

Describing an Exception

Object → public String toString()
(super class of all classes)
↳ method of object passed over to all the classes for overriding

Throwable
↓
Exception
↓
RuntimeException

An object of a class that extends Throwable can be thrown & caught.

ArithmeticException ArrayIndexOutOfBoundsException

eg. → code that may be present

```
public class ArithmeticException ~~extends Round~~ extends RuntimeException
{
    public String toString()
    {
        return "java.lang.ArithmeticException :/ by zero";
    }
}
```

throw new Throwable subclass;

⇒ tostring is called automatically
⇒ tostring is the only method which is called implicitely.

Eg

```
class Myclass extends Exception
{
    public String toString()
    {
        return "Myclass : description of myclass";
    }
}

class Demo
{
    p.s.v.m ( String args[])
    {
        try
        {
            int m = 100, n = Integer.parseInt (args[0]);
            int k = m/n ;
            System.out.println ("k=" +k);
        }
        catch ( ArithmeticException e)
        {
            System.out.println (e );
        }
        catch ( Exception e)
        {
            system.out.println (e);
        }
```

```
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println(e);
}

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

'throw' keyword

Two uses →

⇒ I) To throw exception object explicitly ⇒

eg

```
int m=100, n=0;
int k = m/n;
```

↳ ArithmeticException will be generated
↳ it means the object of ArithmeticException will be thrown by the java.

⇒ we can also throw our own exception when we use/ provide our own primary resource.

⇒ We can create object of Exception class and can throw them in java programs.

⇒ Mostly, this is used to throw manual exceptions.

javap, java.lang.ArithmeticException

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

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

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 programmer

```
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 name 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.

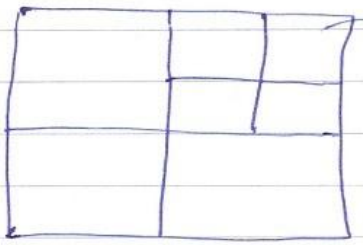
eg

```
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");
        }
    }
}
```

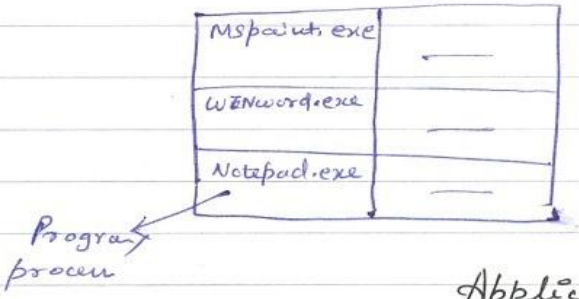
```
class Demo
{
    public static void main (String args[ ])
    {
        First f = new First();

        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 (demon) -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.

⇒ Peculiarities ⇒

→ 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

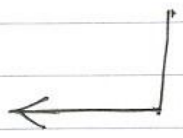
```
public void run()
{

}
```

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()
(Thread ← 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 { in memory where thread is stored.
group

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);

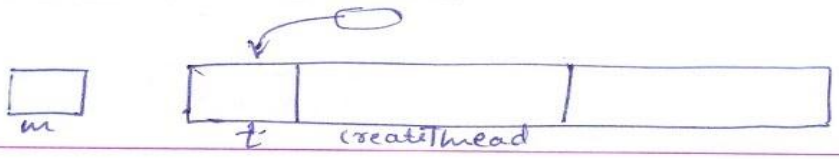
Since, we've ~~to~~ can't pass the object of an interface, hence we've to pass the object of such a class which implements that interface & overrides the method.

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 ( InterruptedException 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.t → child thread
            m.createThread ( );
            m.beginThread( );
            // m.t.start ( );
        }
}
```

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 e)
            {
                S.o.p ("Thread interrupted");
            }
        }
    }
}
class Demo
{
    p.s.v.m (String args[])
    {
        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, if 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 'runnable' 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( )
    {
        1
        2
        3
        4
    }
}
class cDemo
{
    p. s. v. m (String args[ ])
    {
        MyThread m1 = new MyThread (this, "One");
        MyThread m2 = new MyThread ("Two");
        MyThread m3 = new MyThread ("Three");
    }
```

Rough es'

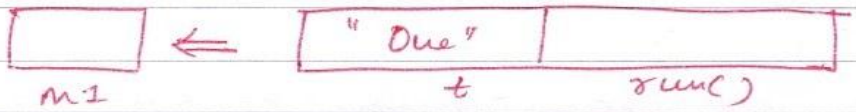
'Threads work ~~on~~ not simultaneously ~~not~~ concurrently.'

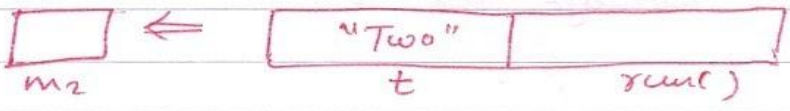
```
class MyThread implements Runnable
{
    Thread t;

    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");
```

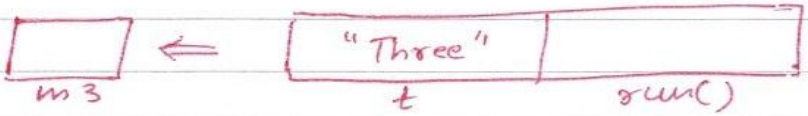
```
MyThread m3 = new MyThread("Three");

S.o.p (m1.t.isAlive());
s.o.p (m2.t.isAlive());
s.o.p (m3.t.isAlive());

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());
s.o.p (m3.t.isAlive());
}
}
```

Status of thread

gives False False False







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.

⇒ ~~Use~~ 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.

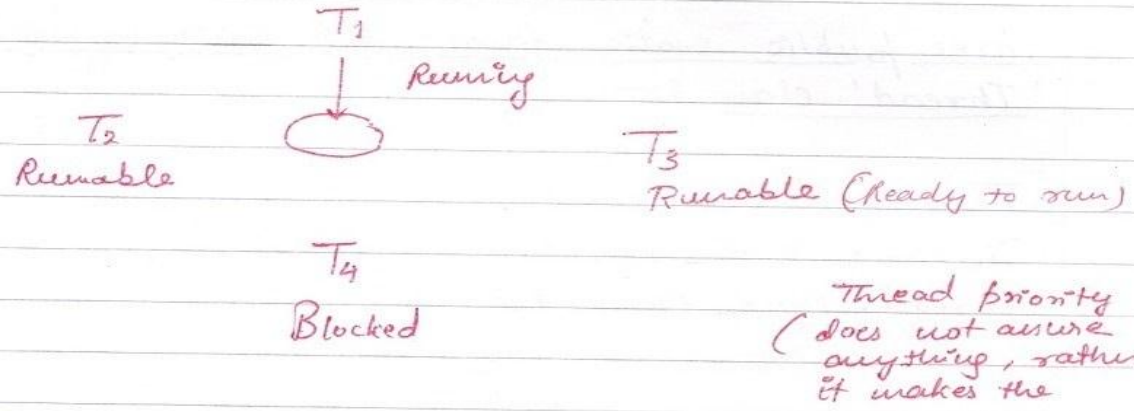
↳ We are not managing the control sequence rather we manage the sequence of sleeping of threads.

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 procen 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 procen are working concorrently, 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)

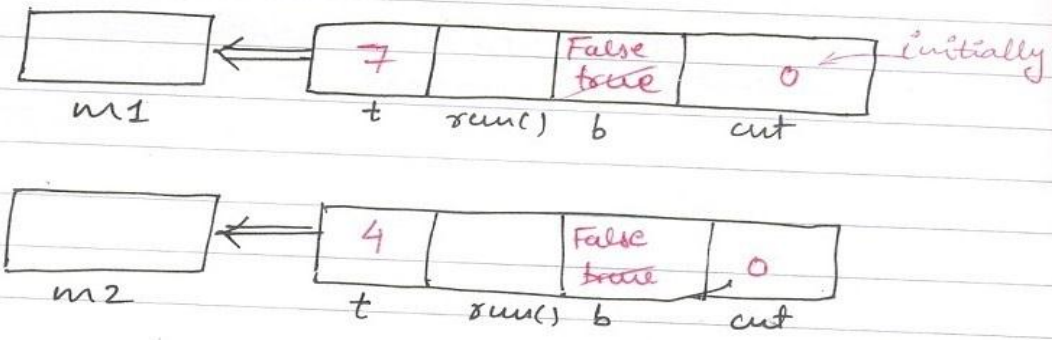
ii) getPriority() :

final int getPriority()

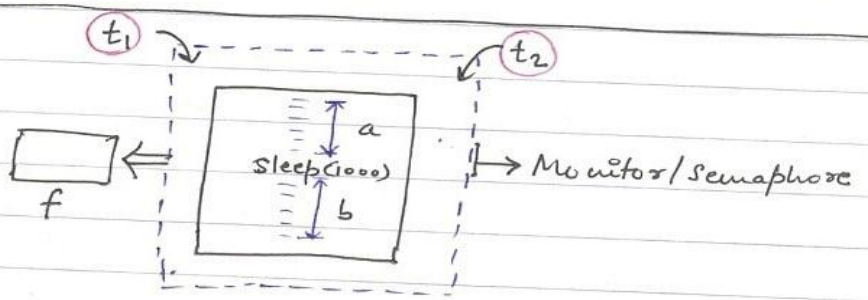
* final variables are written in capitals.

```
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.

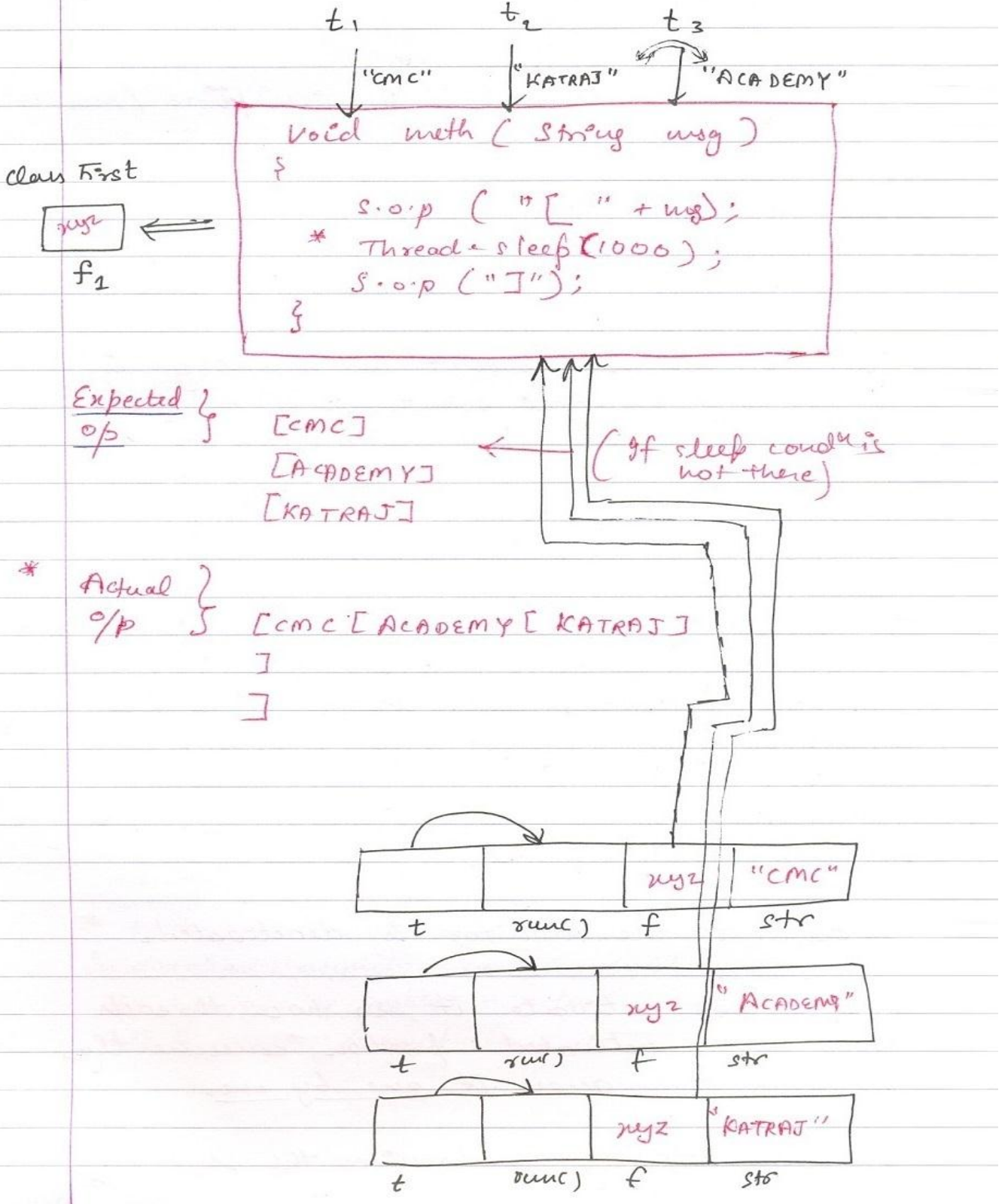
⇒ Every object in memory is covered by a monitor / semaphore.

⇒ 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



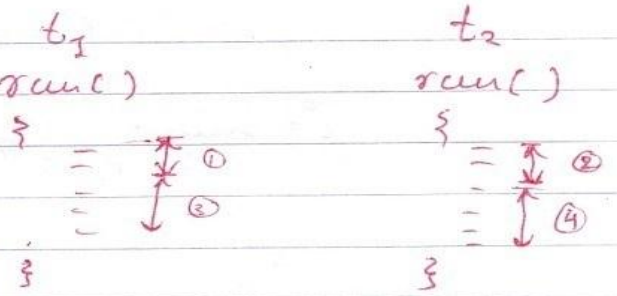
We've to make three objects of first class and then we've to access the meth().

```
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 → t_1 with diffⁿ logic



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 (activating) 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.

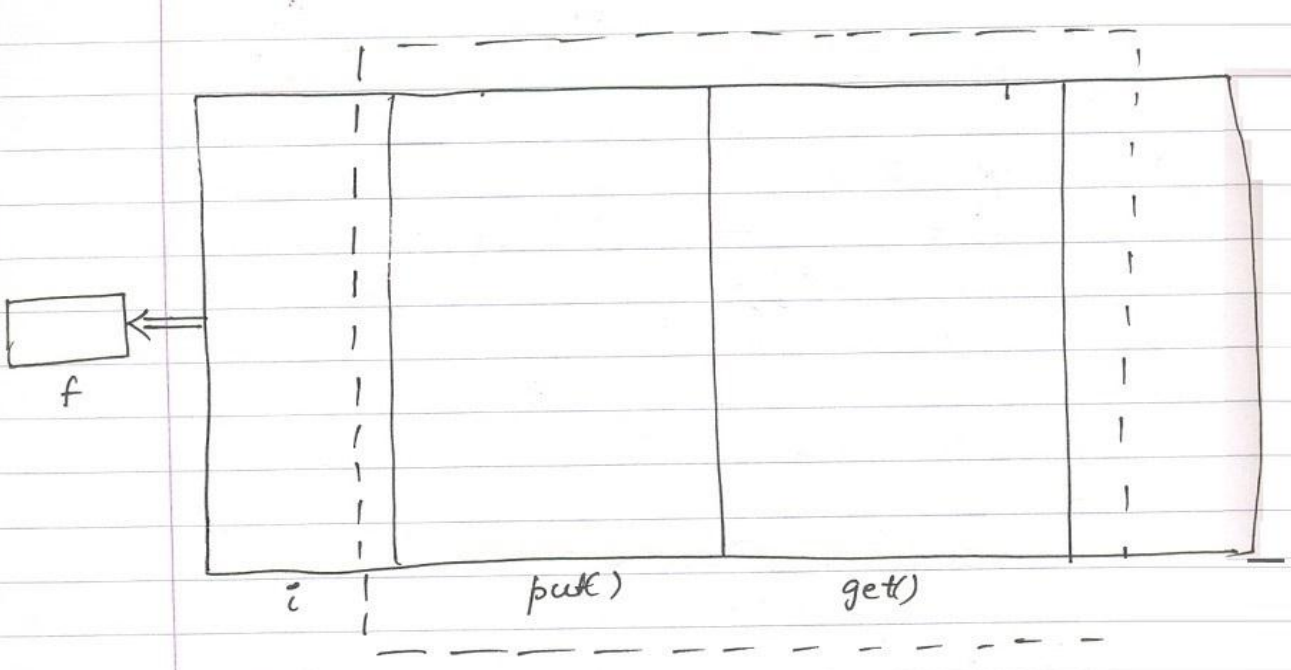
```
class First
{
    int i;
    boolean b = false;

    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();
            }
```

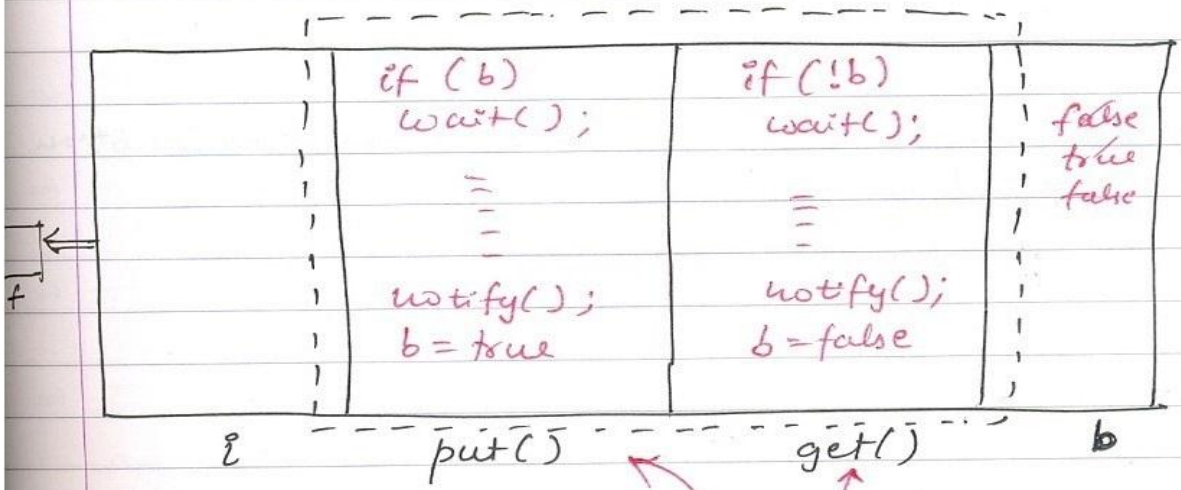
```
            catch (InterruptedException ie)
            {
            }
        }

        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.

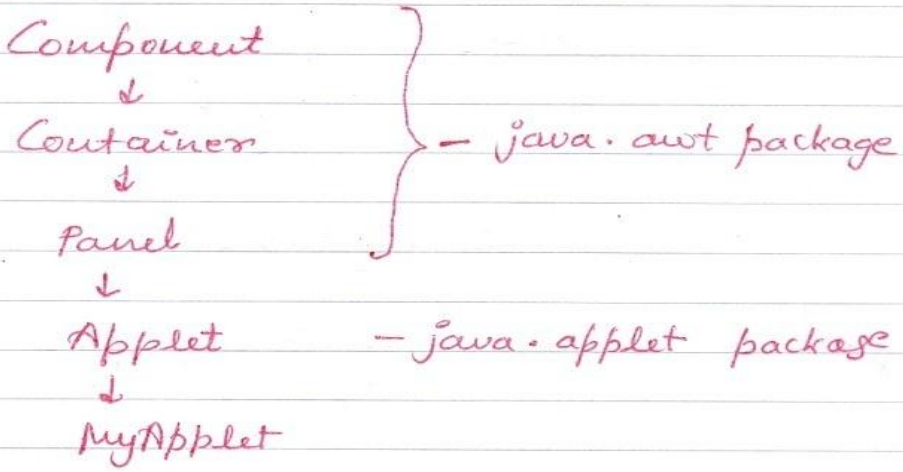
Applet Basics

⇒ ~~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 →



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 ("CMC 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);
    setForeground (Color.red);

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

java.awt package

SetBackground(Color.cyan); → final variable)

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.setColor (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 ↓

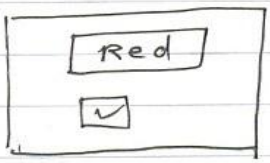
Delegation Event Handling

1) 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
interact

Events are generated when user with the

Events are the objects

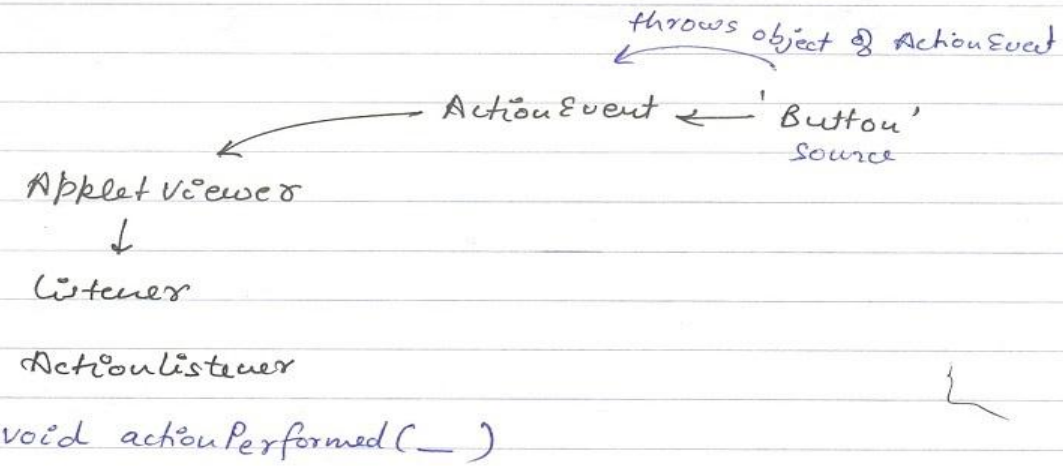
AppletViewer
↓ catches the events & sends it to
Listener
(takes action)

Execution of applet takes place under the influence of AppletViewer

We know in java.awt package, there is a readimade class Button alongwith 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 all the 2-parameters.

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

Methods of 'MouseEvent' class -

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

when we want a certain string to printed at the very same location where the event is performed in the applet window.

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 can.

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 ✗ → CHAR_UNDEFINED
KEY_RELEASED }
KEY_TYPE } ch ✓ keycode ✗ ✓

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)
{
    setBackgroad (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 →

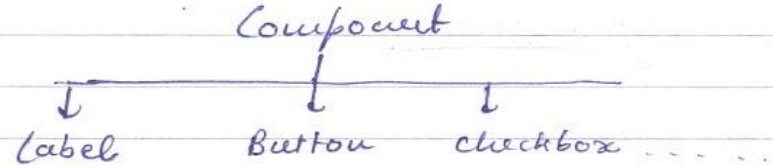
We'll study 9 components and they are the children of Component class.

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 -

ActionListeners

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' interface:

⇒ 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 of 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)

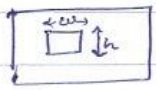
⇒ no event.

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 -

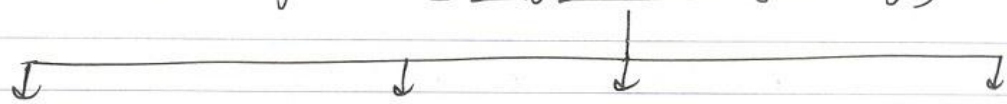
⇒ setLayout (null);

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.

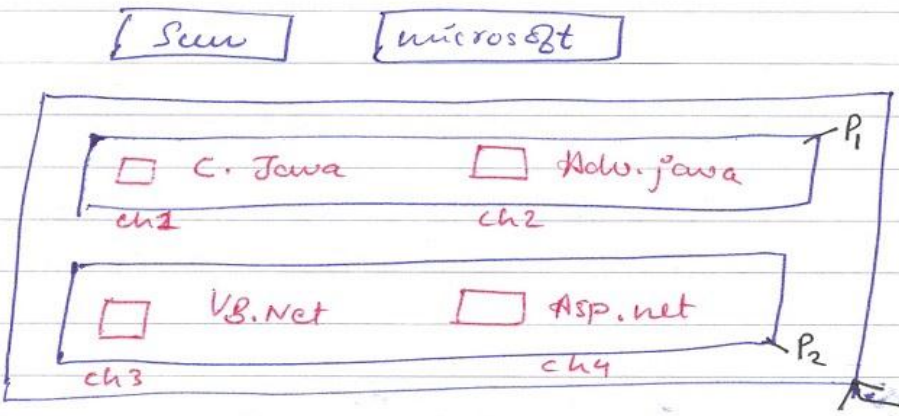
GridLayout

CardLayout :

(Same as Tabcontrol in VB

Eg

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



The requirements are →

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

→ make two buttons
→ add them in applet window
→ 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)

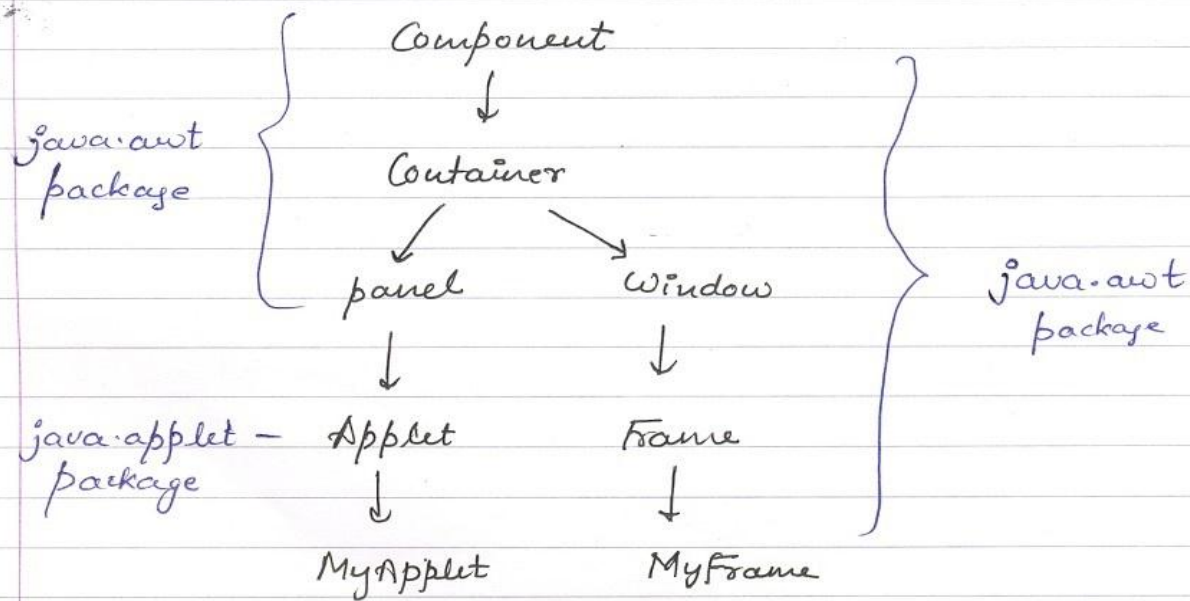
Creating Frames :

⇒ 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()