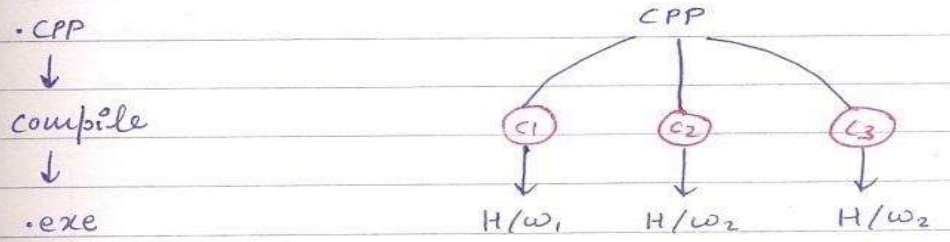
Java – Class based language

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

"Platform independency" ⇒

Example from CPP



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.

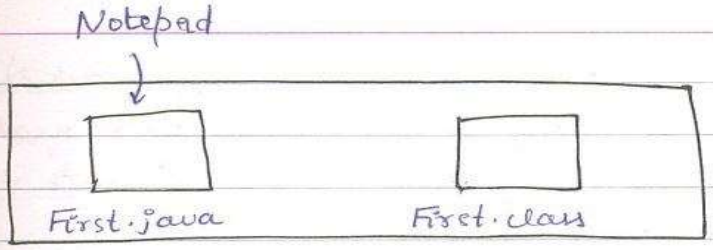
↓

JIT (Just In Time) Compiler

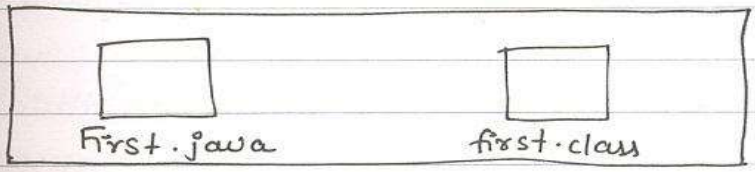
- It accepts code group by group and converts it into .exe.

First Java Program

```
user defined
data type
    ↑
Class First ——→ class name
{
    access specifier   return type   method   command line arguments
          ↑              ↑            ↑            ↑
    public static void main (String args[])
                 └→ can be accessed without object
    {
        System.out.println (" xyz ")
          ↓       ↓          └→ method
        In built  output
        class     variable
          └──────┘
              ↓
        output stream
    }
```



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 type

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 of 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 :

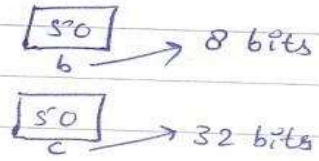
In java coding is done in blocks ({ ... }). A variable declared in a block is local to that

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;          i=i++
int j= ++i;        j=i:
S.o.p(i);          inc
S.o.p(j)           ↓
                   copy
```

o/p → 11
11

− postfix

```
int i=10;          j=i
int j= i++;        i=i+1
S.o.p(i);          copy
S.o.p(j);          ↓
                   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 { = }
```

iii) Nested if

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

iv) else... if Ladder

```
if (cond^n 1)
{
   =
}
else if (cond^n 2)
{ =
}
else if (cond^n 3)
{ =
}
else
{
   =
}
```

v) Switch case

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

* Loops →

I) while

```
initialization ;
while (cond^n)
{
    =
    iteration ;
}
```

II) do while

```
initialisation
do
{ =
  iteration ;
} while (cond^n) ;
```

III for

```
for (initialization ; cond^n ; 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)

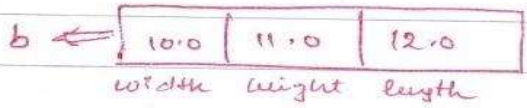
⇒ A class is a logical structure which can be implemented through object. An object is known as an instance of a class.

⇒ 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 :

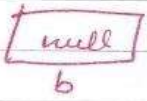
```
Box b = new Box();
```

We can split this statement into two statements as follows →

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

(`new` ↑ operator; `Box()` ↑ Default constructor)

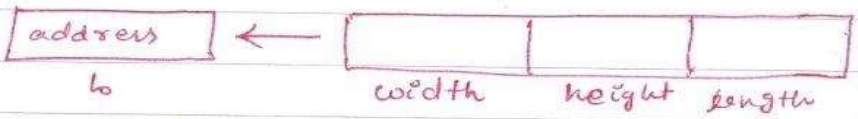
After the execution of 1st statement object reference of Box class named b is created in the memory



→ 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.

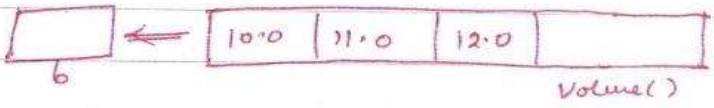


Methods

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();
    }
}
```

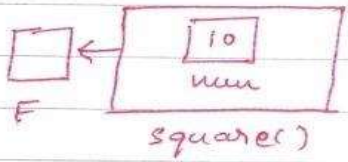


o/p → Volume = 1320.0

⇒ 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 :

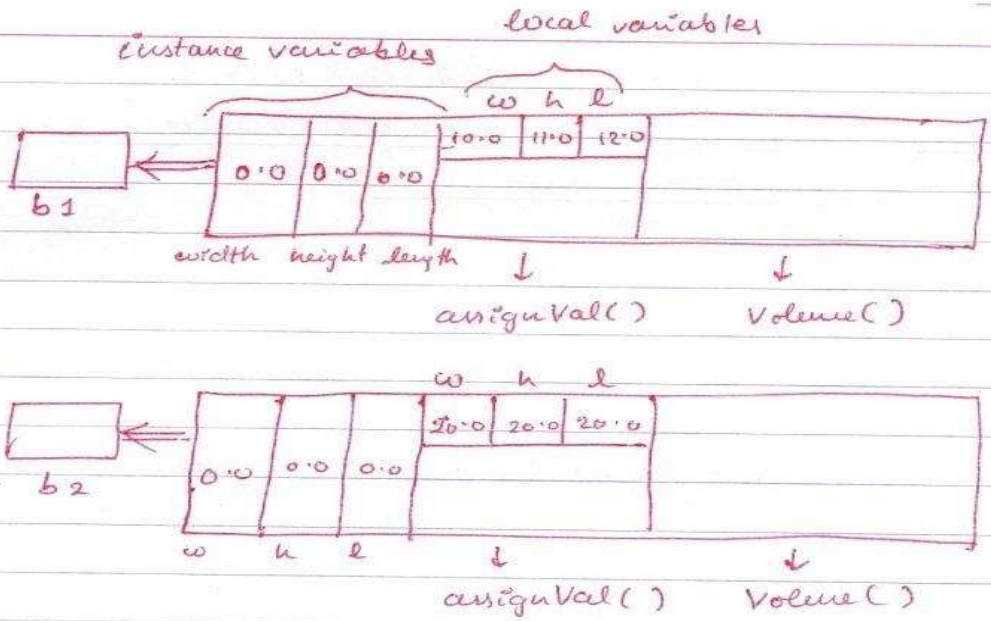
```
class First
{
    int square (int num)
    {
        return num * num;
    }
}

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.

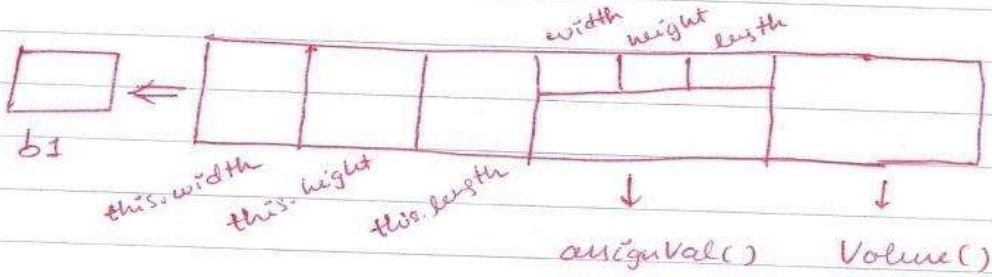
⇒ The 'this' keyword represents current object.

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.

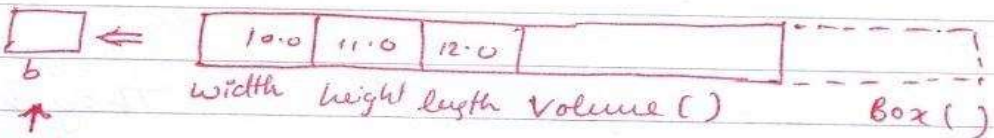
⇒ We can use 'this' keyword inside the body of constructors

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

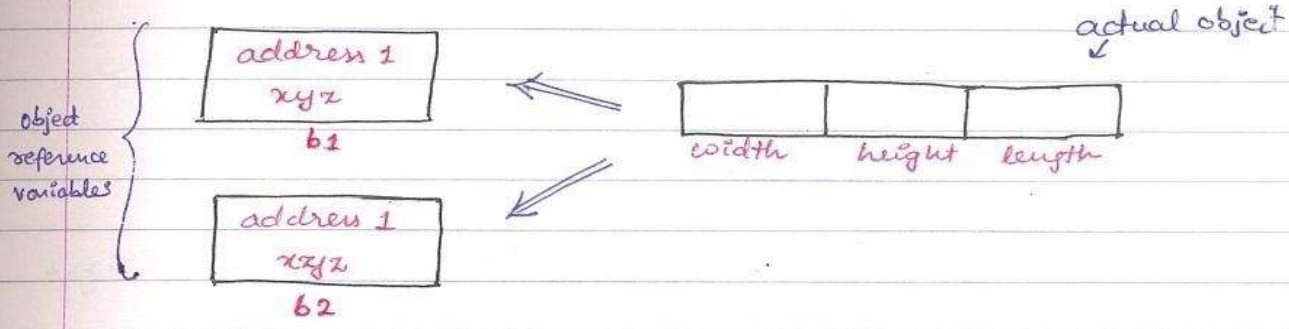
Parameterized constructor :

```
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

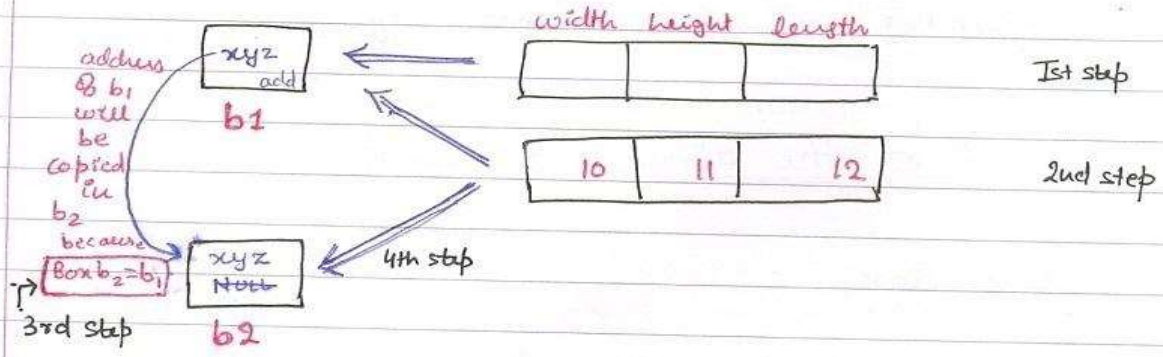
```
class Box
{
        double width, height, length ;
}
```

```
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

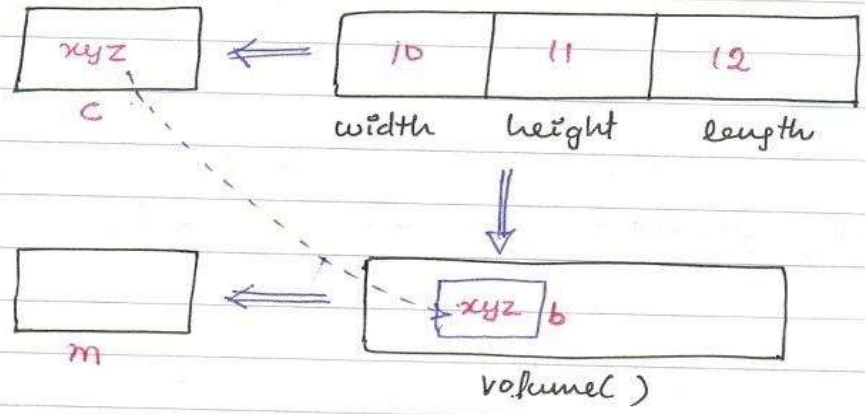
If we're making an object of a particular class having a parameterized constructor, then we can directly pass the parameters/values during formation of the object.

```
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

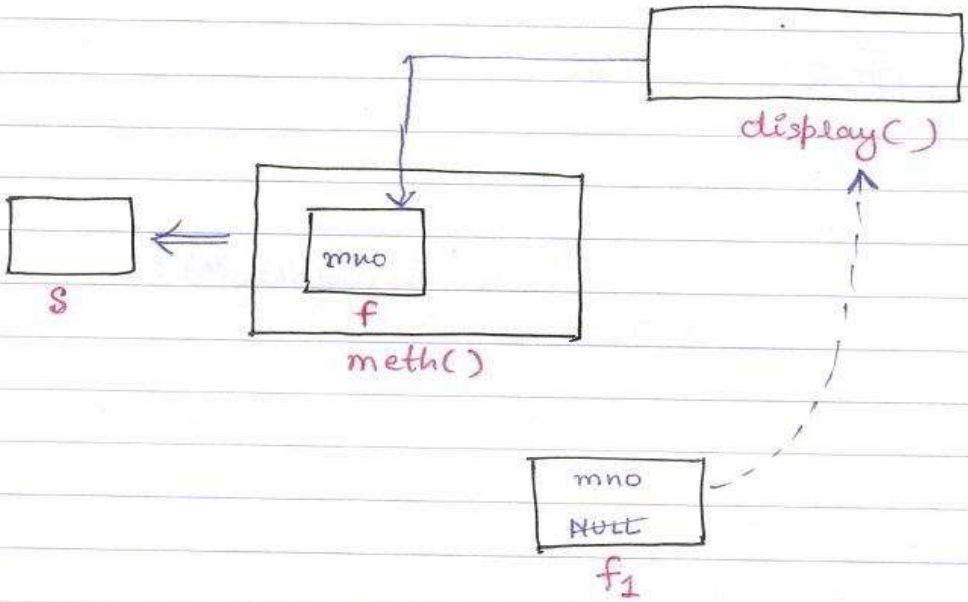
b → obj of Box in volume method

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 (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.

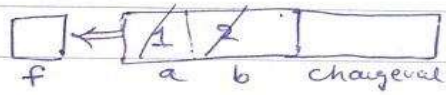
↓

Passing parameters to methods by value & by reference

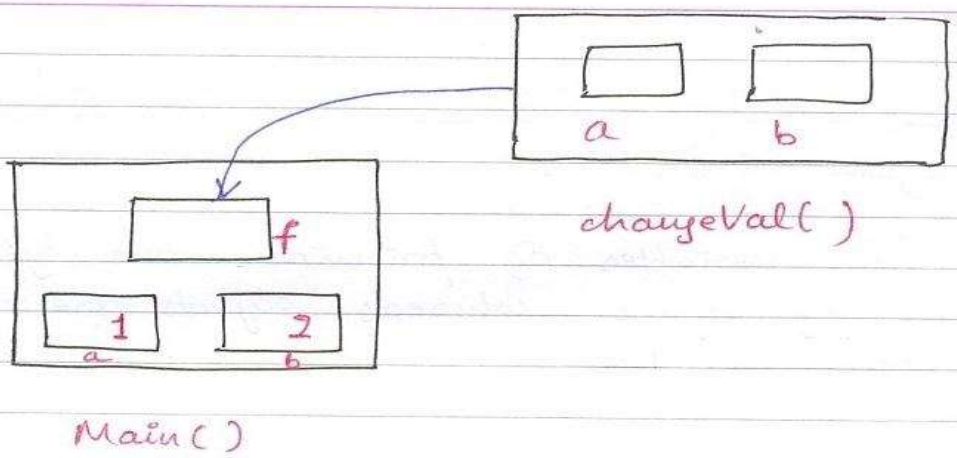
The variables of primitive data types are passed by value, whereas objects are passed by reference.

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 —

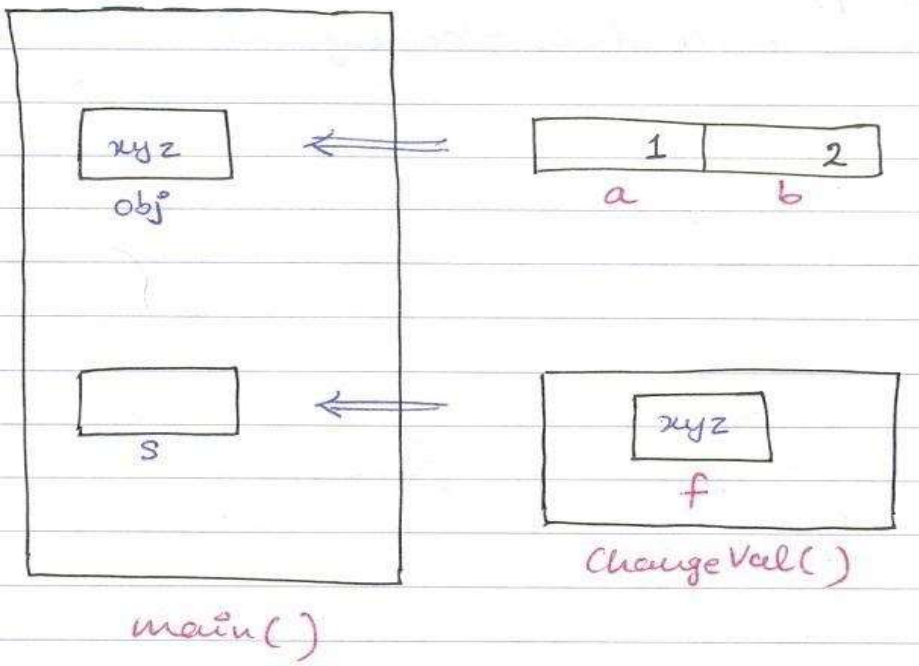


By Reference ⇒

```
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 = " +a "b = " +b)
                                       obj.   obj.
          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);
```

c variable is accessed → `System.out.println (First.a);`

atic method is accessed → `First.meth ( );`

```
        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' clases 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.

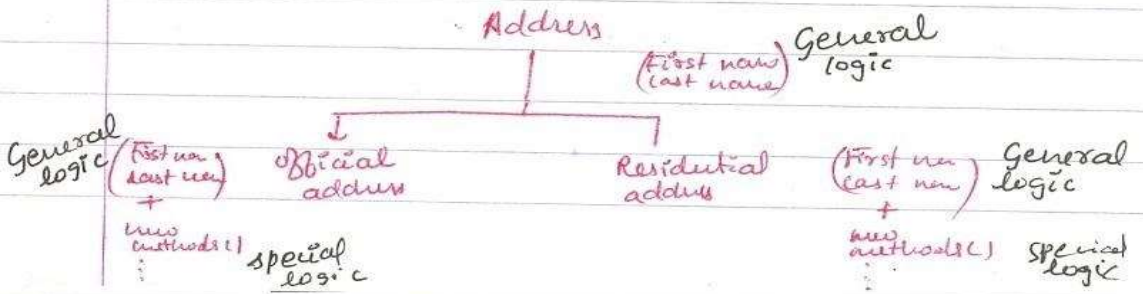
⇒ The extended class is known as super class and extending class is known as sub-class.

⇒ 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 showab()
    {
        System.out.println("a="+a+"b="+b);
    }
}
```

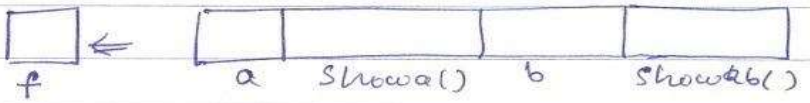
Member access in inheritance

⇒ 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 of refers those members that are defined in Super class and are inherited to sub class.

refer ⇒ can hold addres

```
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 capile time error
        f.showab();
    }
}
```

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

Super keyword

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

I. To call the constructor of super class :

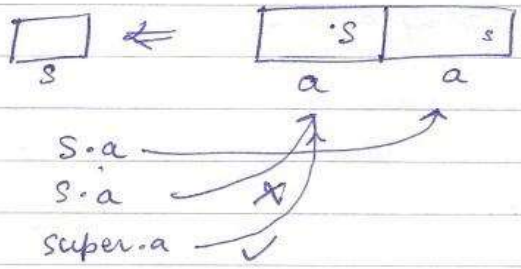
⇒ We can use super keyword in the constructor of sub-class to call the constructor of super class, as shown in following example.

```
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 access()
    {
        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 "

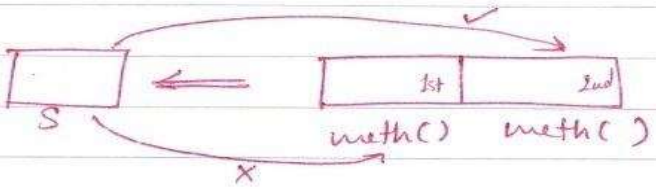
⇒ If we're having a method in the class third', then we can access it by
new Third.meth();
Now, If we'll write the same thing again then another object will be created & the same class & the constructors will be loaded in the memory again.

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.access();
    }
}
```

D.M.D. (Dynamic Method Dispatch)

— Runtime polymorphism of java

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

class First

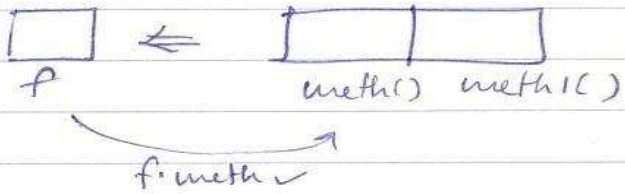
    void meth();

class Second

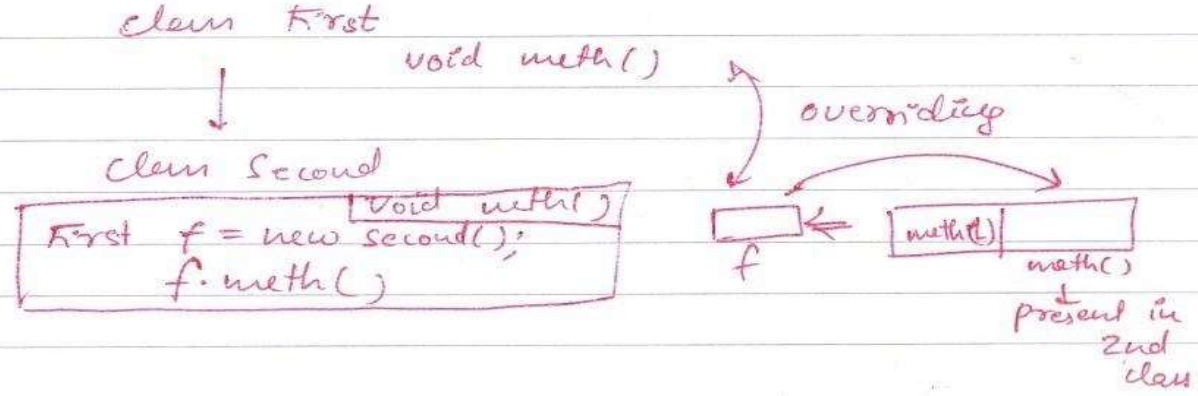
    void meth1();

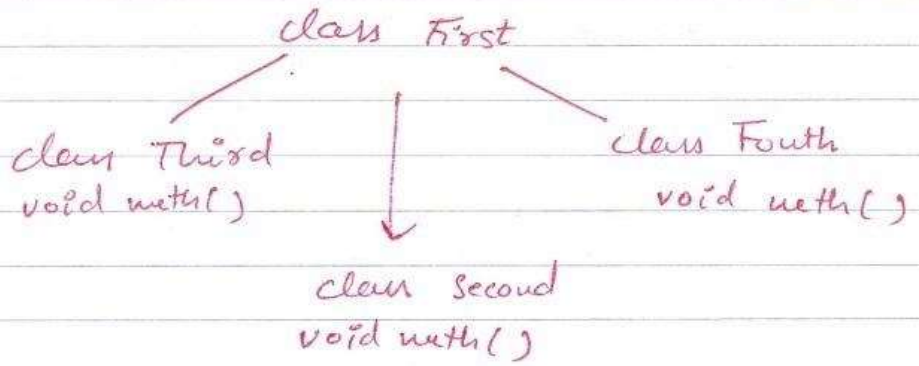
⇒ We can write

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 ↓

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

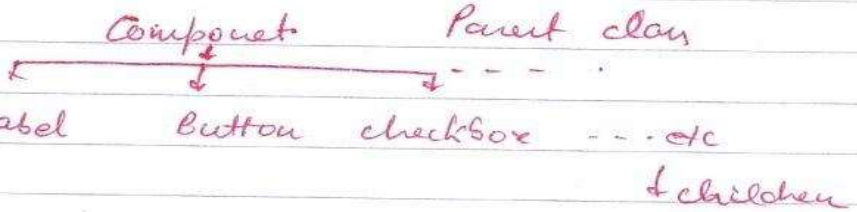
```
class Third extends First
{
    void meth()
    {
        S.o.p.("This is third method");
    }
}

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);
    }
}
```

extra

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.

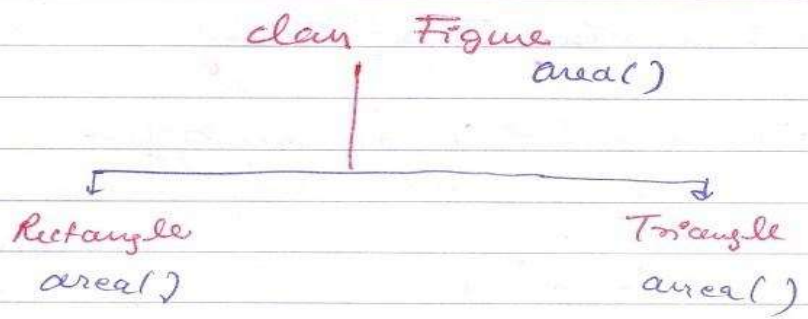
Now, Java has provided a method 'add' for adding the particular class on the applet.



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(~~tri~~ 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 myVoulme ( )
    {
        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.

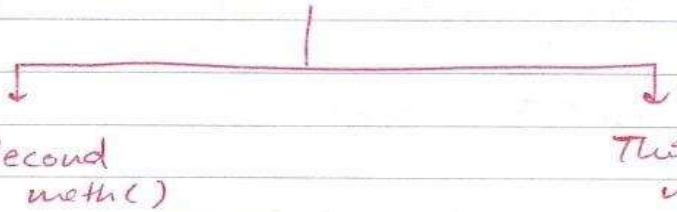
Eg

```
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 m1, int m2)
```

```
{ 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 addrss 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);
    }
}
```

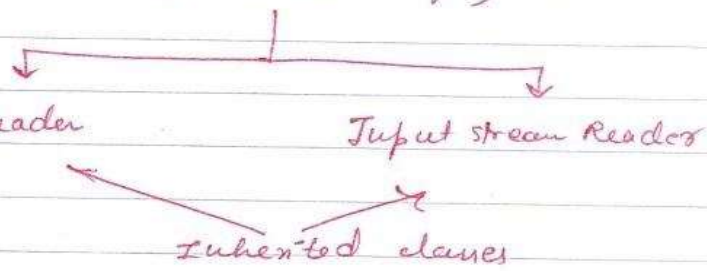
Eg from java library

⇓

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 ↴

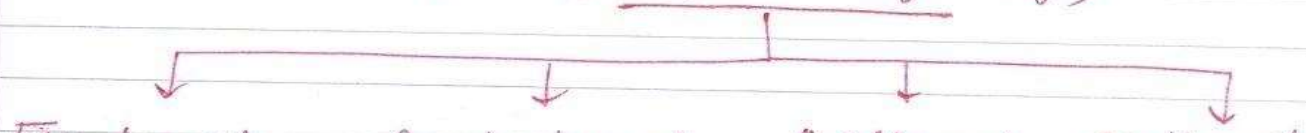
```
interface MyInter
{
        void meth();
}

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 class 2")
        }
}
```

```
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

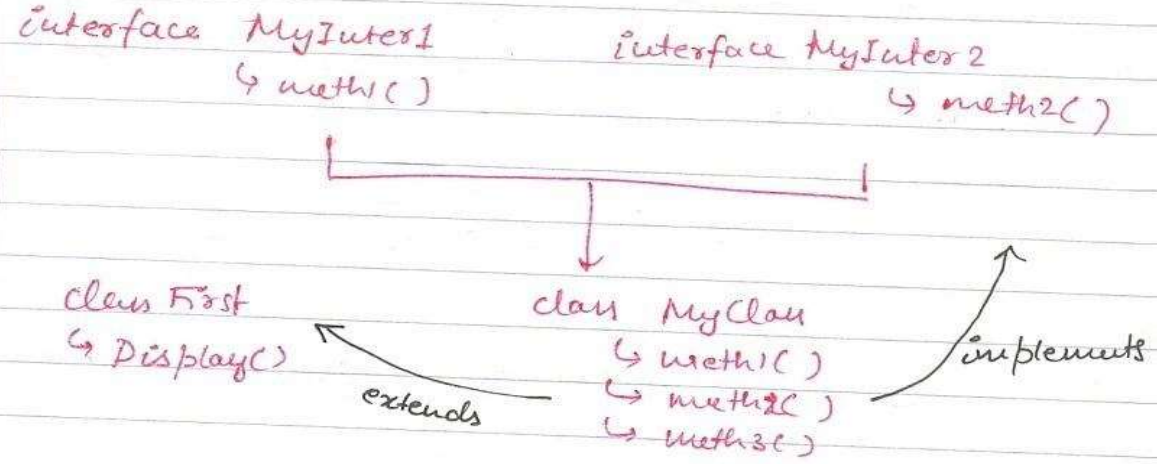
For setting layout of the window, Java has given a method setLayout which accepts the reference variable of LayoutManage which is an interface.

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 implemets 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
    {
    }
}
```

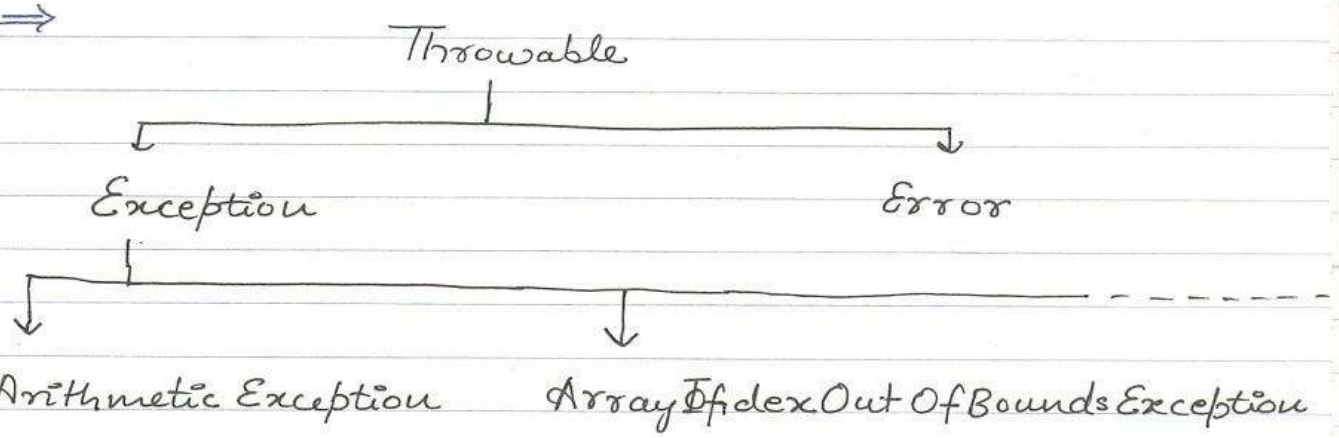
```
class First
{
    final void meth(){}
}

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 -

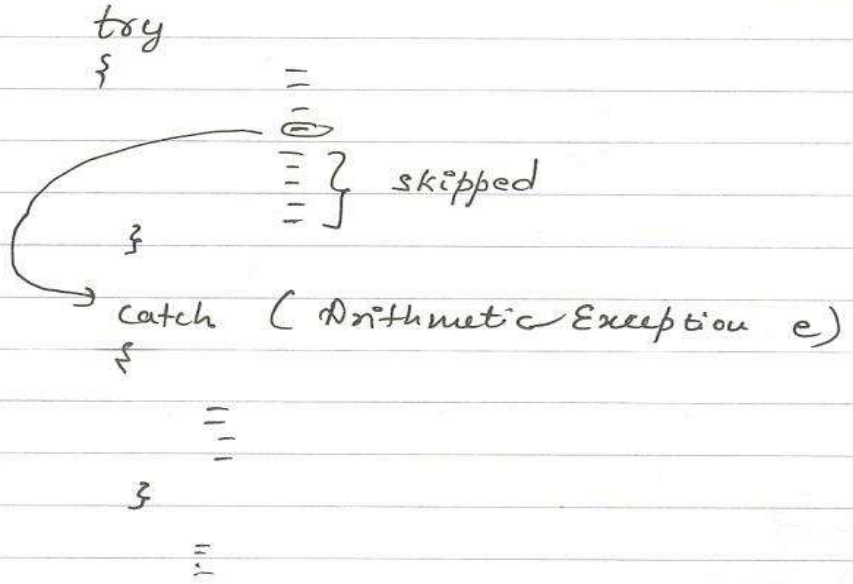
⇒ The code in which, there is possibility of exception is put in try block.

⇒ The code in try block is considered under observation when an exception is raised in try block, the control of program is passed to 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.o.p ("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("outer After 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

↓ ↓

ArithmeticException ArrayIndexOutOfBoundsException

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

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");
            throw ae;
        }
        catch ( ArithmeticException e)
        {
            System.out.println (e);
        }

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

enter another num

o/p :

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