group of statements worked together to perform an operation.

without return type without parameter

```
Class Addition
    {
void add(){
    int a = 10;
    int b = 20;
    int c = a+b;
    system.out.println(c);
    }
Public. static void main (string[]args)
    {
    Addition ad= new addition();
        ad.add();
    }
}
```

o/p

30

with return type without parameter

```
class Addition
int add() {
    {
    int a = 10;
    int b = 20;
    int c = a+b;
    return c;
    }
Public static void main (string[]args)
```

```
    Addition ad = new Addition();
     int result = ad.add();
    System.out.println(result);
      }
  }
```

O/P

30

without returntype with Parameter :

without returntype use void keyword

```
class Addition
  {
 void add (intx, inty)
  {
 int z = x+y;
system.out.println (z);
  }
Public static void main (String[] args)
  {
 Addition ad = new addition();
  ad.add (10, 20);
      }
  }
```

O/P

30

with returntype with Parameter :

```
  class Addition
     {
  int add (intx, inty)
    int z = x+y;
    return z;
    }
  Public static void main (string[] args)
```

```
}
Addition ad = new Addition();
int result = ad.add(10,20);
System.out.println(result);
}
```

int result = ad.add(10,20);

o/p
30

Polymorphism

Ex -> A person go to thier friend behave friendly
-> A person goto class behave student
↓
single Entity behave multiple behaviours
called polymorphism

1) if a single entity shows multiple forms or multiple behaviours, then it is called as polymorphism.
2) Polymorphism is classified into 2 types.
   (i) Compiler Time polymorphism
   *) if the polymorphic nature of an entity is decided by the compiler during the compilation time then it is called as compiler time Polymorphism.
   * we will use method overloading to achieve CTP.
   (ii) Run Time polymorphism
   *) if the polymorphic nature of a entity is decided by the run time then it is called as run time Polymorphism
   *) we will use method overriding to achieve RTP.

method overloading :-

The process of specifying multiple methods, having different signature with same method name is called as method overloading.

* method name same but diff signature
return type diff/same.

Method signature

The signature include 3 parts
i) Number of parameters
ii) Type of parameters
iii) order of Parameters

Polymorphism

Compiler will decide -> compiler Time
* It is called as static Binding / Early Binding.

(1) number of parameters

```
display ()    // 0 parameters
  {
  }
display (int x)   // 1 parameters
  {
  }
```

* using same name to create methods
* signature of different
* it is in method overloading

Type of parameters

```
display ()
  {
  }
display (int x, float x)
  {
  }
display (int x, int y){
  }
```

* number of parameters also same then go to parameter are different are not.
* it is different.

order of parameters

```
display (){
  }
display (int x, float y)
  {
  }
display (float x, int y)
  {
  }
```

* method name same
* type of parameters same . Then go to order of parameters
* ~~it is~~ order different
* So, it is also called method overloading.

No. of Parameter

```
class Addition
  {
  void add ( int x, int y)
  system. out. println ( "Result 1 :" + (x+y));
  }
  void add ( int x, int y, int z)
  {
  system. out. println ( "Result 2 :" + (x+y+z));
    }
  public static void main (String [] args)
    {
   Addition ad = new Addition ();
       ad. add (3,4);
      }
    }
```

O/P

Result 1 : 7

Type of Parameter

```
class Addition
  {
  void add (int x, int y)
     {
  system.out. println ("Result 1 :" + (x+y));
     }
  void add ( int x, double y)
     {
   system. out. println ("Result 2 :" + (x+y));
     }
  Public static void main (String [] args)
    {
   Addition ad = new Addition ();
      ad. add (4,5.6);
    } }
```

O/P

Result 2 : 9.6

order of parameters

```
class Addition
  {
  void add (double x, int y)
    {
  system.out. println ("Result 1 :" + (x+y));
    }
  void add (int x, double y)
    {
  system.out. println ("Result 2 :" + (x+y));
    }
 Public static void main (String [] args)
  {
 Addition ad = new Addition ();
   ad. add (4, 5.6);
  }
}
```

o/p

Result 2 : 9.6

Applying method overloading

1) Instance method
2) static method
3) main method
4) constructor

5) same class
child class

static method :

```
class samjole
    {
   static void show (int x)
   system.out. println ("welcome to int parameter");
   }
```

```
static void show (double x)
 {
system.out. println ("welcome to double parameter");
 }
public static void main(string[] args)
 {
show(5.8);
 }
}
```

o/p

welcome to double parameter.

main - Method - applying method overloading

```
class samjole
 {
  public static void main (int [] args)
  {
  system. out. println ("Int array");
  }
  public static void main (int args)
  {
  system.out. println ("int parameter");
  }
public static void main (string [] args)
  {
   main (new = int[] {4,5,6,7});
   main (12);
  }
}
```

o/p

Int array

o/p:

int Parameter

constructor - Method overloading

```
class sample
    {
  sample ()
     {
  system.out.println ("zero Param const");
   }
  sample ( string. name)
   {
  system.out. println ("My name is : "+name);
   }
  public static void main (string []args)
   {
   sample s = new sample ("Abhishek");
   }
}
```

o/p

My name is : Abhishek.

same class - child class Applying to inheritance for MO

```
class sample
   {
  void display ()
   {
  system.out.println ("parent class")
   }
}
class perfect extends sample
    {
    void display (int x)
     {
  system.out.println ("child class")
    }
public static void main(string[] args)
   {
   perfect P = new Perfect ();
       }   } P.display ();
```

o/p Parent class

-> Polymorphism

```
sample s = new sample();
    } }
```

o/p zero param const

```
();
Perfect P = new Perfect
P.display (5);
} }
```

o/p Child class

~~Runtime overriding :~~

~~method name same sign also same return type same.~~

overloading	overriding.
1) Method name same	1) Method Name same
2) sign different	2) same
3) diff same return type	3) same
4) instance static main constructor to apply M overloading	4) Method overriding is only applied to instance method
5) same/child class (class) runing M overloading	5) parent and chid class runing Method overriding
6) It is also called static binding early binding	6) It is also called dynamic binding / late binding.
7) JVM compilation	7) JVM run time.

Runtime - overriding → Pure polymorphism

* Method name same sign also same return type same.
  * single class we do not apply overriding.

1) Declaring a method in subclass which is alredy present in parent class is know as Method overriding.
2) overriding means to override the functionality of an existing method.
3) Method overriding is an example of runtime Polymorphism.
4) Stalic and final methods cannot be overriden as they are local to the class.
5) overriding → dynamic binding → late binding.
6) In the runtime, JVM figures out the object type and would run the method that belongs to that Particular object.

```
class overridingDemo
  {
  void msg()
    {
  system.out. println ("Parent method")
  } }
  class Demo extends overridingDemo
    {
    Void msg()
      {
    system.out. println ("child class")
      }
    }
  Public static void main (string[]args)
    {
    overridingDemo d = new overridingDemo();
    d.msg();
    }
  }
```

object

reference

O/P

~~child class~~

Parent method

```
Demo d = new Demo()
d.msg();
}
}
```

O/P

child method

* it is excutes which object it is parent class or child class. if the object is parent class Then print inside of parent class. if the object is child class the print inside of child class

```
class overridingDemo
  {
  void msg()
    {
  system.out.println ("Parent method");
    }
class Demo extends overridingDemo
  {
  void msg()
```

```
{
System.out.println("child class Method");
}
Public static void main(String[]args)
{
overriding Demo d = new Demo();
d.msg();
}
}
```

child reference object

parent class reference

o/p

child method

Recursion

If a method call itself multiple times then it is called recursion

```
class Factorial
{
int fact(int n)
{
if(n==1)
return 1;
int x = n* fact(n-1);
return x;
}
Public static void main(String args[])
{
Factorial f = new Factorial();
int result=f.fact(5);
System.out.println(result);
}
}
```

condition true returns 1

if conditional false

create another variable

creating another variable

again method call using method name (fact)

after executing value is stored in x

o/p

120

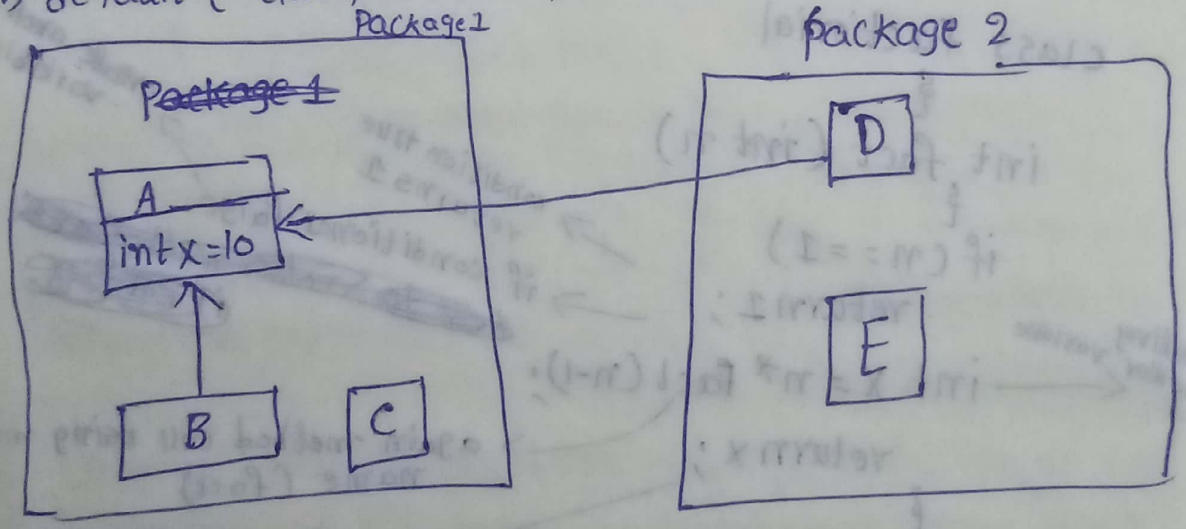
Access Specifiers / Modifiers

1) The Access Specifiers can be used to define the level of accessibility of either the members in the class/interface or the level of accessibility of a class/interface in a package.

2) Using the access specifiers we can define the scope of a class/interface or its members and provide security.

3) The Java language provides 4 levels of acess specifiers with three keywords

(i) public (class, interface, variables, methods & constructors)
(ii) private (variables, methods & constructors)
(iii) protected (variables, methods, & constructors)
(iv) default (class, interface, variables, methods & constructors)




```
package package1;
class A{
 int x=10;
 public static void main (String[]args)
 {
 }
}
```

```
package package1;
class B extends A
{
public static void main (string[]args)
{
}
}
```

```
package package1;
class C
{
Public static void main(String
{
}
}
```

```
package package2;
class D extends A {
public static void main(string[]args)
{
}
}
```

```
package package2;
class E
{
Public static void main(String
{
}
}
```

```
Package package1;
class A {
private int X=10;
Public static void main (string[]args)
{
A a = new A();
system.out.println(a.X);
}
}
```

```
package package1;
class B extends A
{
Public static void main (string[]args)
{
B b = new B();
system.out.println (b.X);
}
}
```

O/P

X has private Access in A

system.out.println(b.X);

so unable to use child class if there is a private keyword

```
package package1;
class C
{
Public static void main (String[]args)
{
A a = new A();
system.out.println(a.X);
}}
```

O/P X has private acess

```
Package Package2
import Package1.*;
class E
  {
  Public static void main (String[]args)
{
A a = new A();
System.out.println (a.x);
  }
}
```

o/p
Error

Acess specifiers / Modifiers

Modifiey	class	Package	subclass	world
private	yes	no	no	no
default	yes	yes	no	no
protected	yes	yes	yes	no
Public	yes	yes	yes	yes

Inner class (or) Nested class

The classes that are declared inside another class are called as inner class.

1) Inner class have a special type of relationship that it can access all the members of outer class including private.
2) More readable and maintainable.
3) Inner class requires less code.
4) if a class is useful to only one other class then it is logical to embed it in that class and keep the two together.

1) Regular Inner class
2) Method local Inner class
3) Anonymous Inner class
4) static Inner class

Regular Inner class / Java Member Inner class

* calling another method of i

*

```
class outer
  {
  class inner
     {
    void innermethod()
     {
    system.out.println("inner class, method");
     }
  }
  Void outerMethod()
     {
    system.out.Println("outer class method");
     Inner i = new Inner();
      i.innerMethod();
  }
  Public static void main (string[] args)
    {
    outero = new outer();
       o.outerMethod();
      }
    }
```

o/p

outer class Method
inner class Method

```
class outer
    {
    class Inner
        {
        void Inner Method();{
        system.out.println("Inner Method");
        }
}
void outer Method();
    {
    system.out.println("outer Method");
    }
public static void main(String[]args)
    {
    outer o = new outer()
        o.outer Method()
    outer.Inner i = new outer().new inner();
        i.inner Method();
        }
    }
```

calling from static method

O/P

outer class Method
inner class Method

Method local Inner class / Java local inner class

```
class outer
    {
    void outer Method()
        {
        class Inner()
            {
            void Inner Method()
                {
                system.out.println("Inner Method")
                } }
        Inner i = new inner();
            i.Inner Method();
```

```
}
public static void main(string args[])
{
outer o = new outer();
    o.outerMethod();
    }
}
```

o/p

Inner Method

Anonymous Inner class

```
class Parent1
    {
    void msg()
        {
        system.out.println("hi friends");
        }
    }
class Parent
    {
    public static void main (string args[])
        {
        Parent1 p1 = new Parent();
            p1.msg();
        Parent1 p2 = new Parent1()
            {
            void msg()
                {
                system.out.println("bye friends");
                }
            };
            p2.msg();
            }
        }
```

o/p

hi friends
bye friends

Final with methods

```
class Final Demo
    {
final void display()
        {
        int X=20;
        system.out.println(X);
        }
    }
class Final Demo2 extends Final Demo
    {
    public static void main (String []args)
    {
    Final Demo2 fd = new Final Demo2();
        fd.display();
    }
    void display()
        {
        int Y=10;
        system.out.println(Y);
        }
}
```

o/p

error: display() in final Demo ? cannot override display() in final Demo.

Static variable & static methods:

1) static keyword in Java is used for memory management it makes your program memory efficient [i.e it saves memory].
2) It can apply for variables, methods, blocks.
3) The static variable can be used to refer the common Property of all objects.
4) Static variables gets memory only once in class area at the time of class loading.

Java static variable

```
class student
  {
    int rollno;   // instance variable
    string name;  // instance variable
static string college "PITT";  // static variable

 student (int r, string n)
      {
       rollno=r;
       name=n;
      }
   void display()
     {
      system.out. println (rollno+" "+name+" "+college);
     }
public static void main (string [] args)
      {
       student s1 = new student (101, "Abhishek");
       student s2 = new student (102, "Raju");
        s1.display();
        s2.display();
      }
  }
```

Java static method

```
class student
        {
          int rollno;
        string name;
       static string college = "PTIT";
      student (int r, string n)
         {
          roll no = r;
          name = n;
        }
 Static void change()
      {
       college = GOVT
```

```
}
void display()
 {
System.out.println("rollno+" "+name+" "+college);
 }
Public static void main(String[]args)
   Student.change();
   Student s1 = new student (101,"Abhishek");
   student s2 = new student (102,"Raju");
      s1.display();
      s2.display();
       }
   }
```

o/p

101 Abhishek Govt

102 Raju Govt

command line arguments

```
class Add Demo
   {
Public static void main(String abc[])
   {
int x = Integer.parseInt(abc[0]);
int y = Integer.parseInt(abc[1]);
int z = Integer.parseInt(abc[2]);
   int a=x+y+z;
   System.out.println(a);
   }
}
```

single inheritance

```
class Employee
    {
     float salary = 40000;  //instance variable
    }
class programmer extend Employee
  {
  float bonus = 10000;
public static void main (string [] args)
    {
  Programmer P = new programmer();
  system.out.println ("Total salary :" + (P.salary + P.bonus));
  }
}
```

o/p

Total salary 50 000.

multi-level inheritance

```
class Add
    {
    int a=15, b=10;
    void add().
      {
      system.out.println (a+b);
    }
class sub extends Add
    {
    void subc)
      {
   system.out println (a-b);
    }
  }
Class Mul extends sub
      {
      void mulc)
        {
        system.out.println (a*b);
        } }
```

```
class Div extends Mul
  {
  void div()
    {
  system.out.println(a/b);
  }  }
}

class  Inheritance
    {
 Public static void main (string [] args)
   {
   Div d = new Div();
    d.add();
    d.sub();
    d.mul();
    d.div();
    }
}
```

Method name are different & Applying inheritance so it is linked.
* so we use reference object. to call Method

o/p

```
25
5
150
1
```

Hierarchial Inheritance

```
class Animal
    {
  void eat()
    {
    system.out.println ("Eating");
    }
  }
class Dog extends Animal
  {
  void barK()
    {
    system.out.println ("barking);
    }
  }
```

```
class cat extends Animal
  {
  void Meow
    {
  system.out. println ("Meowing");
    }
  }
class Inheritance 1
    {
  Public void main (string [] args);
      cat c = new cat();
        c.meow();
        c.eat();
      dog d = new dog();
        d.barK();
        d.eat();
        }
    }
```

o/p

Meowing
Eating
baring
Eating

super keyword

1) The super keyword is used for accessing parent class instance members (instance variable and instance methods) and constructors.

2) The super keyword can be specified either in child class constructors, (or) child class instance methods.

3) The super keyword cannot be specified in child class static methods.

super() ⟶ it is a Method also

This Keyword

```
class Abhishek
  {
  int x=20; — // instance variable
  Void m1()
    {
    int x=10 —— // local variable
    system.out.println(this.x)
    }
  Public Static Void main (string [])args)
    {
    Abhishek a = new Abhishek();
        a.m1();
    }
  }
```

To print instance variable of a class use this keyword

Super Keyword to refer parent class data members variable

```
class Kotha
    {
    int x=30; //instance variable
    }
Class Abhishek extends Kotha
  {
  intx=20; //instance variable
  Void m1();
      int x=10; // local variable
      system.out.println(x);
      System.out.println(this.x);
      system.out.println(super.x);
  }
Public static void main (string args[])
    {
    Abhishek a=new Abhishek();
        a.m1();
    }
}
```

directly prints local variable

same class of instance variable

Parent class of instance var

super to refere Parent class Methods

class methods

```
class Kotha
  {   m1
void ~~main~~()
  {
system.out.println ("Hello from Parent");
}}

class Abhishek extends Kotha
  {
void m1()
super.m1();
system.out.println ("Hello from child");
}
public static void main (string[] args)
  {
Abhishek a1 = new Abhishek();
a1.m1();
}}
```

→ to print in Parent class instance methods

Method names are same, so we not use a1 reference object to call it.

o/p

Hello from Parent

Hello from child

super() // Method is a must be in a first line of Method or constructor

super() to refer Parent class constructor

```
class Kotha
  {
Kotha()
  {
system.out.println ("parent");
  }
}
class Abhishek extends Kotha
  {
Abhishek()
  {
super();
System.out.println ("child");
```

```
public static void main (String [] args)
    Abhishek a = new Abhishek();
    }
}
```

o/p

parent
child

Abstract class in Java

1) A class that is declared using "abstract" keyword is known as abstract class.

2) It can have abstract methods (methods without body) as well as concrete methods (regular methods with body)

3) An abstract class can not instantiated, which means you are not allowed to create an object of it.

4) A class derived from abstract class must implement all those methods that are declared as abstract in the parent class.

Abstract Method in Java

1) Abstract method has no body (only declaration no definition).

2) Always end the declaration with a semicolon (;).

3) It must be overriden. An abstract class must be extended and in a same way abstract method must be overriden.

4) A class has to be declared abstract to have abstract methods.

Important Points

1) An abstract class may also have concrete (complete) methods

2) For design purpose, a class can be delared abstract even if it does not contain any abstract methods

3) Reference of an abstract class can point to its sub-classes thereby achieving run time polymorphism

4) A class must be compulsorily labeled abstract, if it has one or more abstract methods.

5) It can have constructors and static methods also

```
abstract class Animal // Abstract class -> must extended
{
    abstract void sound(); // Abstract Method -> only declaration, no def
    void display() // Concrete Method
    {
        System.out.println("I am from animal class");
    }
}
```

method of class have abstract must be Extended

~~Animal() { s.o.p("constructor"); }~~ // constructor.

```
class Dog extends Animal //inheritance
{
    void sound() // Method overriding
    {
        sysystem.out.println("Barking");
    }
    public static void main(String[] args)
    {
        Animal a = new Dog();
        a.sound();
        a.display();
    }
}
```

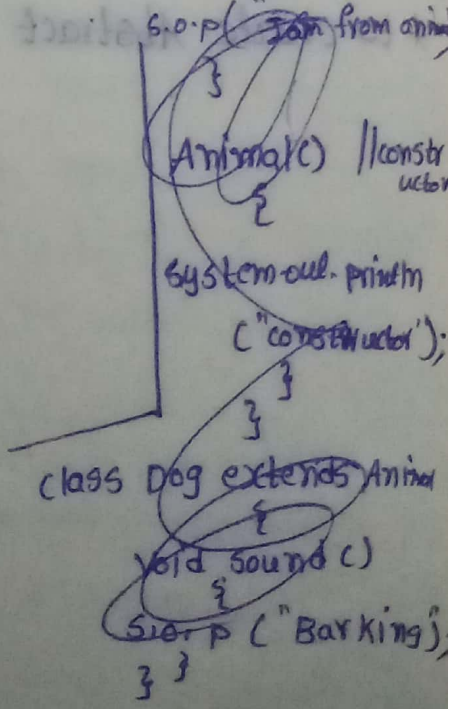
abstract class reference a

o/p

~~constructor~~

Barking

I am from animal class



* Public we do not use abstract keyword + final keyword at a time in Method / Classes Also.

2) Abstract + Static keyword we dont use at a time

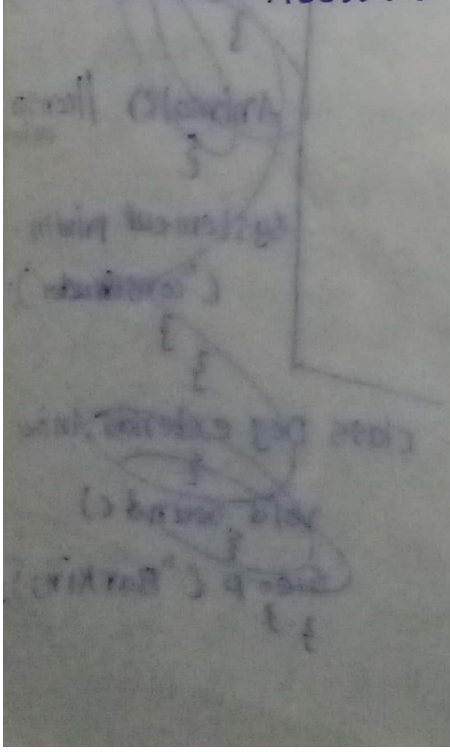
```
abstract class Animal        /* using constructor in Abstract class */
{
  void display()
  {
  system.out.println("I am from animal class")
  }
  Animal()
  {
  system.out.println("constructor");
  }
}
class Dog extends Animal
{
 Public static void main(string args[])
  {
  Animal a = new Dog();
   a.sound();
```

O/P

constructor
I am from Animal class

} — executed constructor & Method.

* to create Abstract class inside abstract / with abstract Method



Strings in Java

2 ways to print
1) using new
2) " "

```
class ArrayDemo
{
Public static void main (string args[])
{
string s1 = new String ("Hello Abhishek");
string s2 = "welcome to Java";
system.out.println (s1);
system.out.println(s2);
}
}
```

using new to print strings

" " directly creates string using double qu
it is stord in String constant Pool

using new it is store in heap

o/p
Hello Abhishek
welcome to Java

String In Java using methods

length method

```
class string Demo
{
public static void main(string[] args)
{
string s1 = new string ("Hello Abhishek");
string s2 = "welcome to PTIT;
System.out.println(s1);
System.out.println(s2);
system.out.println(s1.length());
system.out.println(s2.length());
}}
```

o/p
Hello Abhishek
welcome to PTIT
14
15

char metho

```
C)
s.o.p(s1.charAt
s.o.p(s2.charAt(4)
}
}
```

o/p
e
o

concat Method

```
class string Demo
    {
Public static void main (string args[])
 {
string s1 = new string ("Hello surya");
string s2 = "welcome to PTIT";
 system.out.println(s1);
 system.out.println(s2);
 system.out.println(s1.concat(s2));
 system.out.println(s2.concat(s1));
    }
  }
```

~~s1.equals(s2)~~
~~s1.equals Ignore case(s2)~~

o/p

```
Hello surya
welcome to PTIT
Hello surya welcome to PTIT
welcome to PTIT Hello surya
```

equals Method

```
class string Demo
    {
  Public static void main (string[]args)
   {
  string s1 = new string ("abc");
  string s2 = "ABC";
  system.out.println(s1);
  system.out.println(s2);
  system.out.println(s1.equals(s2));
  system.out.println(s1.equalsIgnoreCase(s2));
        }
    }
```

→ it prints true if only case sensitive and no. of letters are same

→ it print same value a true

o/p

```
abc
ABC
false
true
```

* if in case to create string and change the value of string we use string Buffer

* String Buffer → to modify strings.

1) using string Buffer to store strings, append, insert, replace,

→ in string Buffer, string builder we should create only using new keyword.

```
class stringBufferDemo
{
public static void main (string[] args)
{
stringBuffer sb = new stringBuffer ("string");
System.out.println (sb);
System.out.println (sb.length());
}
}
```

Hello Abhishek

o/p

Hello Abhishek
14

Append Method

Append → it is used to add existing string → The word is added to after string Buffer

```
class stringBuffer
{
Public static void main (string args[])
{
StringBuffer sb = new stringBuffer ("Hello Abhishek");
system.out.println(sb)
system.out.println (sb.append ("kotha"));
}
}
```

this word directly added to last.

o/p

Hello Abhishek kotha

Delete

to delete group of letter

```
system.out.println(sb);
system.out.println (delete(1,3));
System.out.println (sb.deleteCharAt(0));
```

o/p

Hello Abhishek
Hlo Abhishek
ello Abhishek

substring

```
system.out.println(sb);
system.out.println(sb.substring(1));
system.out.println(sb.substring(1,4));
  }
}
```

→ It prints After the first letter

→ it prints After first letter before of 4th letter.

o/p

```
Hello Abhishek
ello Abhishek
ell Abhishek
```

insert

```
system.out.println(sb);
system.out.println(sb.insert(6,"Kotha"));
  }
}
```

we like to add the word in ~~the~~ by based on index letter

o/p

```
Hello Abhishek
Hello Kotha Abhishek
```

replace

```
class
    system.out.println(sb);
    system.out.println(sb.replace(0,4,"Kotha"));
  }
}
```

o/p 1 2 3 4

```
Hello Abhishek
Kotha Abhishek
```

reverse

```
    system.out.println(sb);
    system.out.println(sb.reverse());
  }
}
```

o/p

```
Hello Abhishek
kehsihbA olleH
```

delimeter

```
import java.util.stringTokenizer;
Public class stringDemo
    {
    Public static void main (string[]args)
     {
    string str = new string Tokenizer (str, "a");
     while (t1.has more tokens);
        {
       sysytem.out.println (t1.nextToken());
        }
      }
   }
```

o/p

```
J
V
Tutori
ls in Telugu by koth
Abhishek.
```

* if in case character also printed in saperate token using third constructorr (string str, string delim, boolean flag)

```
import Java.util.stringTokenizer;
public class stringDemo
      {
    public static void main (string[] args)
      {
    string str = "Java Tutorials in Telugu by Katha Abhishek";
    stringTokenizer t1 = new stringTokenizer (str, "a", true);
     while (t1.has/more token()):
       {
     system.out.println (t1.next tokens ());
        }
   } }
```

o/p

```
J                    ls in Telugu by koth
a                    a
V                    Abhishek
a
Tutori
a
```

Interface

1) An interface is a collection of Abstract Methods.
2) An interface can contain both variables and methods.
3) An interface can not be instantiated.
4) In order to Acess the member of interface we need to inherit the interface into a class using implements Keyword.
5) In the subclass we have to override all the methods of an interface then it is called as Implementation class.
6) In the sub class we have to overrite all abstract methods.
7) A class can implement any number of interface.

```
Interface MyInterface //interface
{
public static final int X = 15;
public abstract void m1(); // abstract method
}
class InterfaceDemo implements MyInterface //interface
{
public void m1()  //abstract Method
{
System.out.println("welcome");
}
public static void main (String[] args)
{
InterfaceDemo id = new InterfaceDemo();
id.m1();
System.out.println(id.X)  or)  S.O.P(X);
```

by default variable have public static final

by defaut Methods have public, abstracted

must / should use Public to extend it

variable by default in static so directly call it

class -> class -> extends (single class)
class -> interface -> implements (any number)
interface -> interface -> extends (any num)

```
interface MyInterface
  {
  public static final int X=15;
  public abstract void m1();
  }
interface MyInterface2
  {
  public static final int Z=20;
  public abstract void m3();
  }
interface MyInterface1 extends My Interface, MyInterface2
    {
    public static final int Y=20;
    public abstract void m2();
    }
class InterfaceDemo implements MyInterface1
    {
    public void m1()
      {
      system.out.println("welcome");
    }
    public void m2()
      {
      system.out.println("Abhishek");
      }
      public void m3()
        {
        system.out.println("PTIT");
        }
  Public static void main (string args[])
      {
      InterfaceDemo id = new InterfaceDemo();
      id.m1();
      system.out.println(X);
        id.m2();
      system.out.println(Y);
        id.m3();
        system.out.println(Z);
      }
```

o/p

welcome
15
Abhishek
20
PTIT
20

default method

```
Interface MyInterface        // Interface
 {
 public static final int x=15;
 Public abstract void m1();   // abstract Method
 }
interface MyInterface2
    {
    public static final int z = 20;
    default void m3()          // default Method
          {
          system.out.println ("from default");
          }
    }
interface MyInterface1 extends My Interface, My Interface2
       {
       Public static final int y: 20;
       Public abstract void m2();          // abstract Method
       }
class Interface Demo implements My Interface1
        {
        Public void m1()
           {
           system.out.println ("welcome");
           }
      public void m2()
        {
        system.out.println ("Abhishek");
        }
 Public static void main (string args [])
        {
        Interface Demo id = new InterfaceDemo();
            id.m1();
           System.out.println(x);
            id.m2();
            System.out.println(y);
              id.m3();
              system.out.println (z);
              }
    }
```

o/p

welcome
15
Abhishek
20
from default
20

* using string Buider also to modify same as string Buffer and Applying length, APPend, insert, replace ---

* using string Builder and string buffer to modify string.

* difference of string Buibler and string buffer

* using string Buffer we create string it is synchorized uses only one member / only 1 thread at a time

* string Builder (Not- synchorized) it can acess by many threads at a same time.

* one or more more Acess the strings to use string Builder.

```
class string Builder
    {
    public static void main (string []args)
      {
    string Builder sb = new string Builder ("welcome to PTIT).
    system.out.Println (sb);
      } system.out.println ( sb.reverse());
    }
```

olP

welcome to PTIT

TITP ot emoclew.

string tokenizer

The Java.util.string Tokenizer class allows you to break a string into tokens. It is a simple way to break strings

string Tokenizer (string str)	creates string Tokenizer with specified string.
string Tokenizer (string string delim)	creates string Tokenizer with specified string and delimeter

string Tokenizer (string str, string delim, boolean flag)

boolean hasMoreTokens()	checks if there is more tokens available
string nextToken()	returns the next token from the string Tokenizer object
int countTokens()	returns the total number of tokens

```
1. String Tokenizer st = new string Tokenizer ("Java Tutorials by Kotha
                                                Abhishek");
2. while (st.hasMoreTokens()) {
3. System.out.println (st.nextToken());
4.  }
```

Example program

```
import Java.util.StringTokenizer;
 Public class stringDemo
      {
      Public static void main (string [] args)
       {
      string str = "Java Tutorials in Telugu by Kotha Abhishek";
      stringTokenizer t1 = new stringTokenizer (str);
      while (t1.hasMoreTokens())
           {
        system.out.println (t1.nextToken());
           }
      }
}
```

O/P

```
Java
Tutorials
in
Telugu
by
Kotha
Abhishek
```

```
Public static void main(String[] args)
{
InterfaceDemo id = new InterfaceDemo();
    id.m1();
    System.out.println(x);
    System.out.println(Y);
    System.out.println(Z);
```

o/p

welcome
15
20
20

Packages

1) A package as the name suggests is a pack (group) of classes, interfaces and other packages.
2) In Java we use Package to organize our classes and interfaces.
3) we have two types of Packages in Java built in Packages and the Packages we can create (also known as defined Packages)

*) Reuseability
*) Better organization
*) Name conflicts.

Predefined Packages

1) Core Packages : Java.util, Java.lang, Java.io : → starts with Java it is core
2) extended Packages : Javax.swing, Javax.sql → x is there it is extended
3) third Party Packages (vendor) : Start Any names

```
Package package1;
class PackageDemo
{
Public static void main (string[]args)
{
system.out.println("welcome to Packages");
}
}
```

package words must be small letters

-d c:

stored the directory

compiling

c:\users\desktop> Javac -d. PackageDemo.Java

c:\users\desktop> Java package1.PackageDemo

c:\users\user\Desktop> Java package1.PackageDemo

o/p: welcome to packages,

Sub Packages

main package.

subpackage of youtube

subpackage of Java

```
Package youtube.Java.package1;
class packagedemo1
{
Public static void main (string[]args)
{
system.out.println("welcome to sub packages");
}
}
```

compilation

c:\users\user\Desktop> Javac -d. PackageDemo1.Java

c:\users\user\Desktop> Java youtube.Java.package1.PackageDemo1

o/p

welcome to subpackages.

importing and

```
Package youtube.Java.package1;

public class packageDemo1
        {
    Public void m1()
        {
        system.out.println("hii from Package");
        }
    Public static void main (string [] args)
        {
        system.out.println("welcome to Java Packages");
        }
    }
```

This package accesed by seperate class are in below

o/p welcome to Java Packages

*Accessing

```
import youtub.Java.package1.packageDemo1;

class Abhishek
    {
    public static void main (string [] args)
        {
        Package Demo1 P = new package Demo1();
        P.m1();
        }
    }
```

compilation

Javac Abhishek.Java

Java Abhishek

o/p

hi from package

Exception handling

1) The term exception means exceptional condition, it is a problem that may arise during the execution program

2) A bunch of things can lead to exceptions, including Programmer errors, hardware failures, files that need to be opened cannot be found, resource exhaustion e.t.c.

3) Java provides a robust and object orienled way to handle exception scenarios, known as Java Exception handling.

4) When the exception occurs in a method, the process of creating the exception object and handling it over to runtime environment is called "throwing the Exception".

5) once runtime recieves the exception object, it tries to find the handler for exception. Exception handler is the block of code that can process the exception object. The handler is said to be "catching the exception".

6) Note that Java Exception handling is a framework that is used to handle runtime errors only, compile time errors are not handled by Exception handling.

checked exception (user) ----> declare (throws)
---> handle (try, catch, throw, finally)

5 Keywords for Exception handling.

1) Try
2) catch
3) throw
4) throws
5) finally

1) The "try" Keyword is used to specify a block where we should place exception code. The try block must be also followed by catch or finally. It means, we can't use try block

2) The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

3) The "finally" block is
the program. It is executed wheather an exception is handled or not.

4) The "throw" key word is used to throw an exception.

5) The "throws" keyword is used to declare exceptions. It doesn't throw one exception. It specifies that there may occur an exception in the method. It is alway used with method signature

```
class ExceptionDemo
  {
public static void main (string [] args)
 {
system.out.println ("Today our topic is Exception Handling");
  try
   {
   int a = 10
    int b = 0;
    int c = a/b;
   system.out.println(a);
   system.out.println(b);
   system.out.println(c);
    }
catch (Arithematic Exception ae)
    {
    ae.printstackTrace(); // exception name, reason, line, method, class.
    }
system.out.println ("welcome to programming Tutorials In Telugu");
system.out.println("by Kotha Abhishek");
} }
```

ae.p to

O/P

Today our topic is Exception Handling
welcome to programming Tutorials In telugu
by Kotha Abhishek
Java.lang.Arithemalic Exception: / by zero
at ExceptionDemo.main (ExceptionDemo.Java:8)

tostring Method

```
class ExceptionDemo
    {
    public static void main(String[]args)
    {
    System.out.println("Today our topic is Exception Handling");

    try
     {
     int a=10;
     int b=0;
     int c=a/b;
    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
     }
    catch(Arithematic Exception ae)
       {
       system.out.println(ae.toString());   //exception name, reason
       }
    System.out.println("welcome to programming Tutorials in Telugu");
     system.out.println(" by Kotha Abhishek");
      }
   }
```

o/p

Today our topic is Exception Handling
Java.lang.ArithematicException:/ by zero
welcome to programming Tutorials in Telugu
by Kotha Abhishek.

getMessage Method

```
  }
catch (Arithematic Exception ae)
   {
  system.out.println(ae.getMessage());   // reason
   }
system.out.println("welcome to programming Tutorials in Telugu");
system.out.println(" by Kotha Abhishek");
} }
```

o/p Today our topic is Exception Handling → printed only reason
/by zero

user defined Me

```
}
catch (Arithematic Exception ae)
  {
  system.out.println("you can not divide a number with 0");
  }
system.out.println("welcome to programming Tutorials In Telugu");
system.out.println("by Kotha Abhishek");
  }
}
```

o/p

Today our topic is Exception Handling
you can not divide a number with 0
welcome to programming Tutorials In Telugu.
by Kotha Abhishek.

try, catch, final

```
class ExceptionDemo
  {
  public static void main (String[]args)
  {
  system.out.println("Today our topic is Exception Handling");
  try
  {
  int a=10;
  int b=0;
  int c = a/b;
  system.out.println(a);
  system.out.println(b);
  system.out.println(c);
  }
  catch (Arithematic Exception ae)
    {
    system.out.println(ae.t
    e.printStackTrace();
    }
  finally
    {
    system.out.println("from finally block");
    }
  system.out.println("welcome to programming Tutorials In Telugu");
  system.out.println("by Kotha Abhishek");
    }
}
```

o/p

Today our topic is E.H
Java.lang.Arithematic Exception
at ExceptionDemo.main(E.:8)
From finally block
welcome to programming..
by Kotha Abhishek

single try block Multiple catch blocks

↳ sing try block multiple

```
class ExceptionDemo
  {
  public static void main (string [] args)
   {
   try
    {
    int arr [] = {2,5};   → index 0,1
    arr [2] = 5;          → but index 5 not possible
    arr [3] = 3/0;        → first error will be printed
    }
  catch (Arithematic Exception ae)
     {
     ae. print stack Trace();
     }
  catch (Array Indexout of Bounds Exception aioob)
     {
     aioob . printstack Trace();
     }
  }
}
```

arr [3] = 3/0.
arr [2] = 5 ;

o/p

Java lang. Arithematic E
at Exception Demo. main

o/p

Java. lang. Array Indexout of Bounds Exception : 2
at Exception Demo. main (Exception Demo. Java : 6)

* single try multiple final blocks are not possible.
single try have only single final block.

Nested try block

having more than one Exception and print the both exception by using Nested try block

```
class Exceptiondemo
  {
  public static void main ( string [] args)
    {
    try
    {
    int arr[] = {2,5};
      try
       {
       arr [2]= 3/0;
       }
       catch ( Arithematic Exception ae)
```

```
{
ae.printStackTrace();
}
arr[3]=7;
}
catch(ArrayIndexOutOfBoundsException a)
{
a.printStackTrace();
}
}
}
```

o/p

```
Java.lang.ArithmeticException: / by zero
        at ExceptionDemo.main(ExceptionDemo.java:4)
Java.lang.ArrayIndexOutOfBoundsException: 3
        at ExceptionDemo.main(ExceptionDemo.java:13)
```

Don't Know the Exception

```
class ExceptionDemo
{
public static void main(String[] args)
{
try
{
int arr[]={2,5};
arr[2]=7;
}
catch(~~Arithmetic Exception~~ ArrayIndexOutOfBoundsException a)
{
a.printStackTrace();
}
catch(Exception e)
{
e.printStackTrace();
}
}
}
```

→ don't know what Exception it is
Exception is super class to all Exceptions
we use Exception the execution is stopped

o/p

```
Java.lang.ArrayIndexOutOfBoundsException: 2
        at ExceptionDemo.main(ExceptionDemo.java:6)
```

Throws & Throws

1) The Java throw keyword is used to explicitly throw an exception from a method or any block of code.
2) we can throw either checked or unchecked exception in Java by throw keyword.
3) The throw keyword is mainly used to throw custom exception
4) we can define our own set of conditions or rules and throw an exception explicitly using throw keyword.
5) For example, we can throw Arithemetic Exception when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword.

```
class ExceptionDemo
  {
  static void validate (int age)
   {
   if (age<18)
    {
     throw new ArithematicException ("You are not eligible for vo
    }
    else
     {
      system.out.println("you are eligible for vote");
     }
   }
  public static void main (string args[])
   {
   validate (17);
   system.out.println(" program sucessfully completed");
   }
}
```

custo Excep

o/p

```
Exception in thread "main" java.lang.ArithematicException : You a
        at ExceptionDemo.validate (ExceptionDemo.java:4)
        at ExceptionDemo.main (ExceptionDemo.java:12)
```

```
validate (35);
System.out.println ( Program sucessfully completed");
  }
}
```

o/p

you are eligible for vote

Program sucessfully completed.

Throws

```
class ExceptionDemo
  {
  static void display() throws ArithematicException
    {
    int a= 3/0;
    System.out.println (a);
    }
  Public static void main (string args[])
    {
    try
      {
      display();
      }
    catch (ArithematicException ae)
      {
      ae.printstackTrace();
      }
    system.out.println ("sucessfully printed");
    }
}
```

o/p

```
Java.lang.ArithematicException : / by zero
        at ExceptionDemo.display (ExceptionDemo.Java :3)
        at ExceptionDemo.main (ExceptionDemo.Java :9)
sucessfully printed.
```

* throws ---> only for checked Exception (IoException, FNFE, CNFE, SQLException)

not eligible for vote

User-defined Exception

1) Create the new exception class extending Exception class
2) Create a public constructor for a new class with string type of parameter.
3) pass the string parameter to the super class.
4) declare the Exception at the Method level.
5) Create try block inside that create a new exception and throw it based on some condition.
6) write a catch block and use some predefined exceptions
7) write the optionally finally block.

```
class EligibilityException extends Exception
{
public ElizibilityException (string str)
  {
   super (str);
  }
public static void main(string args[]) throws EligibilityExce
{
System.out.println("Enter your age");
try
 {
Java.util.Scanner sc = new Java.util.Scanner(System.in
int age = sc.nextInt();
if(age<18)
  {
throw new EligibilityException("You are not eligible fo
  }
else
 {
  System.out.println("You are eligible for vote");
 }
}
catch(EligibilityException Demo ed)
  {
  ed.printStackTrace();
}  }  }
```

o/p

```
C:\Users\user\Desktop> Javac EligibilityException.Java
C:\Users\user\Desktop> Java EligibilityException
Enter your age
99
you are eligible for vote
C:\Users\user\Desktop> Java EligibilityException
Enter your age
9
EligibilityException: you are not eligible for vote
        at EligibleLityException.main(EligibilityExcptn.Java:12)
```

te");

Multithreading in Java

Multitasking

1) Multitasking is a process of executing multiple tasks simultaneously.
2) We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:
   *) process-based multitasking (multiprocess)
   *) Thread-based multitasking (multithreading)

1) Process-based Multitasking (multiprocessing):

i) Each process has an address in memory. In other words, each process allocates a seperate memory area.
ii) A process is heavy weight.
iii) cost of communication between the process is high.
iv) switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based multitasking (multithreading):

1) Threads share the same adress space.
2) A thread is lightweight.
3) cost of communication between the thread is low.

* The process many disadvantages
* so we use Thread based multitasking (multithreading)

multithreading

1) multithreading in Java is a process of executing multiple threads simultaneously.

2) A thread is a light weight sub-process, the smallest unit of processing.

3) It is a seperate path of execution.

Life cycle of a Thread (Thread states):

1) The life cycle of the thread in Java is controlled by JVM. The Java thread states are as follows;

*) New : The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

*) Runnable : The thread is in runnable state after invocation of start() method, but the thread schedular has not selected it to be runing thread.

*) Running : The thread is in runing state if the thread schedular has selected it.

*) Non-Runnable (Blocked): This is the state when the thread is still live, but is currently not eligible to run.

* Terminated : A Thread is in terminated or dead state when its run() method exits.

Achieve multithreading in Java

1) In Java language multithreading can be achieve in two different ways.
   * using thread class
   * using Runnable interface

using thread class (extending Thread class)

1) create any user defined class and make that one of a derived class of thread class
2) override run() method of Thread class
   (It contains the logic of perform any operation)
3) create an object for user-defined thread class and attached that object to predefined thread class object.
4) call start() method of thread class to execute run() method.

multithreading → simple program

```
class ThreadDemo extends Thread        → user defined
                                       → pre defined
    {
    public void run()                  → override run
    {
    System.out.println("run method is executed by JVM");
    }
    Public Static Void main (String args[])
    {
    ThreadDemo td = new ThreadDemo();  → object of user-def
    Thread t = new Thread (td)         → object of predefi
        t.start();
        }
}
```

o/p

run method is executed by JVM

Example program using Runnable Interface

```
class MultiDemo implements Runnable
  {
 public void run()
 { system.out. println ("Thread is Running");
 }
 public static void main (string args[])
  {
  multiDemo m1 = new multiDemo ();
  Thread t1 = new Thread (m1);
      t1.start();
    }
  }
```

o/p

Thread is Running.

current thread method

It gives information of current thread

```
class MultiDemo implements Runnable
  {
  public void run()
   {
  Thread t = Thread.currentThread(); }
  System.out.println(t);
  }
  Public static void main (String args[])
    {
   MultiDemo m1 = new multiDemo();
     Thread t1 = new Thread (m1);
```

user-defined Th

current Thread information stored in t variable

```
t1.start();
Thread t = Thread.currentThread();
System.out.println(t);
}
}
```

} Main Thread

o/p

```
Thread [main, 5, main]
Thread [Thread-0, 5, main]
```

multiple objects - multiple Threads

```
class MultiDemo implements Runnable
{
String name;
multiDemo (string name1)
{
name = name1;
}
public void run()
{
for(int i=01; i<=10; i++)
{
system.out.println (name+":"+i);
}
}
Public static void main (string args[])
{
MultiDemo m1 = new multiDemo ("Thread 1");
multiDemo m2 = new multiDemo ("Thread 2");
Thread t1 = new Thread (m1);
Thread t2 = new Thread (m2);
t1.start();
t2.start();
for(int i=0; i<=10; i++)
{
```

```
system.out
  }
 }
}
```

```
o/p
main : 0
main : 1
main : 2
- - -
main : 10
Thread 2 : 1
Thread 1 : 1
Thread 2 : 2
Thread 1 : 2
= = = - =
Thread 2 : 10
Thread 1 : 10
```

Sleep method ()

```
class MultiDemo implements Runnable
{
String name;
multiDemo (String name 1)
 {
  name = name 1;
  }
  public void run ()
  {
  for (int i = 1; i<10; i++)
  {
  try
   {
  Thread.sleep (1000);
   }
  catch (InterruptedException e)
   {
  system.out.println(e);
  }
  System.out.println (name +" : " +i);
 } }
```

```
public static void main (string args[])
{
MultiDemo m1 = new multiDemo ("Thread 1");
multiDemo m2 = new multiDemo("Thread 2");
    Thread t1 = new Thread (m1);
    Thread t2 = new Thread (m2);
=       t1.start()
        t2.start()
    for (int i=0; i<=10; i++)
      {
    System.out.println("main : "+i);
      }
   }
}
```

O/P

```
main 0
main 1
main 2
= = = =
Thread 2 : 2
Thread 1 : 2
Thread 2 : 2
Thread 1 : 2
- - . .
- - . .
Thread 2 : 10
Thread 1 : 10
```

Join Method

Java Thread Join Method can be used to pause the current thread execution untill unless the specified thread is dead.

```
class MultiDemo implement Runnable
  {
  String name;
  MultiDemo (String name 1)
```

```
{
name = name1;
}
public Void run()
{
for(int i=0; i<=10; i++)
{
try
{
Thread.sleep(1000);
}
catch (InterruPted EXception e)
{
e.printstacTrace();
}
System.out.println(name+" :"+i);
}}
public static void main (string args[])
{
MultiDemo m1 = new multiDemo ("Thread1");
MultiDemo m2 = new multiDemo ("Thread2");
Thread t1 = new Thread (m1)
Thread t2 = new Thread (m2)
t2.start()
{
try
{
t1.Join();
}
Catch (InterupledException e)
{
e.printstack Trace ();
}
t2.start();
for(int i=0
try
{
t2.Join();
}
catch(Interupled Exception e)
{
```

```
        e.printStackTrace ();
    }
    t1.start();

    for(int i=0; i<10; i++)
    {
    System.out.println("main : "+ i);
    }
  }
}
```

O/P

```
Thread 1 : 1
Thread 1 : 2
Thread 1 : 3
Thread 1 : 4
- - - -
Thread 1 : 10
Thread 2 : 1
Thread 2 : 2
Thread 2 : 3
- - - - -
Thread 2 : 10
main : 0
main : 1
- - - - -
main : 10
```

Thread-Synchronization

~~To Access single object. for more than one thread~~ To access synchronization is usefull

~~Synchronization gives access to~~

accessing single object we perform only one Thread and 1st thread executing The next Thread is in waiting state. The 1st executed then next Thread executed.

```
class multiDemo implements Runnable
{
int count;
public synchronized void run()
{
for(int i=0 i<10000; i++)
 {
 count++;
 }
}
public static void main(String args[])
{
multiDemo m1 = new multiDemo();
 Thread t1 = new Thread(m1);
 Thread t2 = new Thread(m1);
  t1.start();
  t2.start();

 try
  {
  t1.Join();
  t2.Join();
  }
catch(InterruptedException e)
 {
 e.printStackTrace();
 }
System.out.println(m1.count);
 }
}
```

→ without this key

o/p
11450
↘ be like
but our o/p is 20K

o/p

20000

is.alive Method

```
t1.start();
t2.start();
system.out.println(t1.isAlive());
try
{
 t1.Join();
 t2.Join();
}
catch (InterruptedException e)
{
e.printStackTrace();
}
system.out.println(m1.count);
 system.out.println(t1.isAlive());
}}
```

it is not executed so it is alive, true

it is ended or excaule so not alive so false

o/p

```
true
20000
false
```

setName - get Name method

```
t1.start();
t2.start();
t1.setName("My Thread 1");
t2.setName("My Thread 2");
system.out.println(t2.getName());
system.out.println(t2.getName());
try
{
t1.Join();
t2.Join();
}
catch (InterupledException e)
```

not set Name by default

o/p

```
Thread 0
Thread 1
20000
```

```
{
e.printStackTrace();
}
system.out.println(m1.count);
}
}
```

o/p
My Thread 1
My Thread 2
20000

Set priority - get priority

```
t1.setpriority(1);
t2.set priority(10);
system.out.Println(t1.get priority());
system.out.println(t2.get priority());
try
{
t1.join();
t2.join();
}
catch(InterupledException e)
{
e.printstackTrace();
}
system.out.println(m1.count);
}
}
```

} without set priority by default

o/p
5
5
20000

o/p
1
10
20000

Max = 10

t1.setpriority(Thread.MAX_PRIORITY)

Min = 1 → t2.setPriority(Thread.MIN_PRIORITY)

```
system.out.println(t1.get priority());
system.out.println(t2.get priority());
```

o/p
10
1
20000

Deamon Thread

Deamon Threads are ~~supported~~ summed in background and support h) in other threads

user threads are converted to Deamon thread but the setDemon(true) method must be used in befor of t1.start(); ~~for~~ method.

= **check it Daemon or not**

```
class MultiDemo implements Runnable
{
Public void run()
 {
  System.out.println ("welcome to Daemon Thread");
 }
Public static void main (string args [])
 {
 MultiDemo  m1 = new multiDemo();
 Thread t1  =  new Thread (m1);
    t1.start();
  system.out.println (Thread.current Thread(). is Daemon());
  }
}
```

o/p

false

welcome to Daemon Thread

convert to Daemon

```
Thread t1 = new Thread (m1);
  t1.set Daemon (true);
  t1.start();
  System.out.println (t1.is Daemon());
  }
}
```

```
t1.set Daemon(false);
t2.start();
system.out.println(t1.is Daemo
 }}
```

o/p false
welcome to Daem n

are stopped.

o/p
true → welcome to Daemon Thread

→ it means it is Daemon so after ~~it is~~ execution of others are stopped

Inter-thread communication

```
class customer1
 {
 int amount = 10000;
public synchronized void withdraw (int count)
  {
System.out.println ("Going to withdraw");
if (this.amount < amount)
  {
System.out.println ("less Balance waiting to deposit");
  try
   {
   wait();
   }
  catch (interrupted EXCEPtion e)
   {
   e.printstackTrace();
  }
 }
this.amount = this.amount - amount;
system.out.println ("withdraw completed");
}
Public synchronized void deposit (int amount)
    {
    System.out.println ("Going to deposit");
    this.amount = this.amount + amount;
    System.out.println ("Deposit completed");
    notify();
   }
}

class InterThreadDemo
 {
 Public static void main (string [] args)
  {
  customer1 c = new customer1();
```

```
new Thread()
  {
  public void run()
    {
    c.withdraw(15000);
    }
  }.start();

new Thread()
  Public void run()
    {
    c.deposit(10000);
  }
  }.start();

  }
}
```

o/p

Going to withdraw
less balance waiting to deposit
Going to deposit
deposit completed
withdraw completed → by using notify again calling withdraw and deposited money having so withdraw comple