Q. Why should we write immutable classes ?

Ans:

- State of an immutable object can't be changed once they are created.
- They are automatically syncrinized/thread safe.
- Immutable objects are good Map keys and set elements.

Creating an Immutable class:

- Class must be declared final.(so that child classes can't be created hence behaviour can't be extended)
- Data members in the class should be declared private and final.(so that we can't change their values after object creation)
- A parameterized constructer.
- No setters only getters.

Q. What is builder design pattern ?

Ans:

A builder design pattern gives a way to create complex immmutable objects.

- The client calls a constructer with all the required fields and gets a builder object.
- The client calls setter like methods to set each optional

parameter of interest.

- finally the client calls the build method to genetate the new object wich is immutable.

Q. Design Patterns ?

Creational:

- Factory
- Abstract Factory
- Singleton
- Prototype
- Builder

Structural:

- Adapter(Allows two incompactible interfaces to work together)
- Bridge
- Decorator("attach a flexible additional responsibilities to an object dynamically".)
- facade

Behavioral:

- Iterator

JVM Architecture:

- javac, command creates .class file from the source .java file.
- this .class file is the input to the classloader subsystem.
- This classloader subsystem is responsible for loading, linking and initialization.
- Loading consists of 3 class loaders,

  1. **BootStrap classloader**
  2. **Extension classloader**
  3. **Application classloader.**

- Above classloaders follow **extension deligation hierchey algorithem**.
- Linking consistes of **verify, prepare and resolve**.

Memory Areas Present inside JVM:

1. **Method Area (Class level data and static variables will be there)**
2. **Heap Area(Object and corresponding instance data will be there)**

3. **Stack Area(All local variables are stored in corrosponding stack, for each thread a separate run time stack is created.)**
4. **PC Registers (For every thread a separate pc register is created.)**
5. **Native Method stacks**

Execution Engine:

1. **Interpreter**
2. **JIT Compiler**
    - **Intermediate code generator**
    - **code optimizer**
    - **Target code generator**

3. Garbage Colletor

4. Security manager

JNI(Java native interface) is responsible to provide native method information to execution engine.

Note:

Bootstrap classloader loads classes from bootstrap classpath

i.e. rt.jar

all core java api classes are loaded by bootstrap classloader.

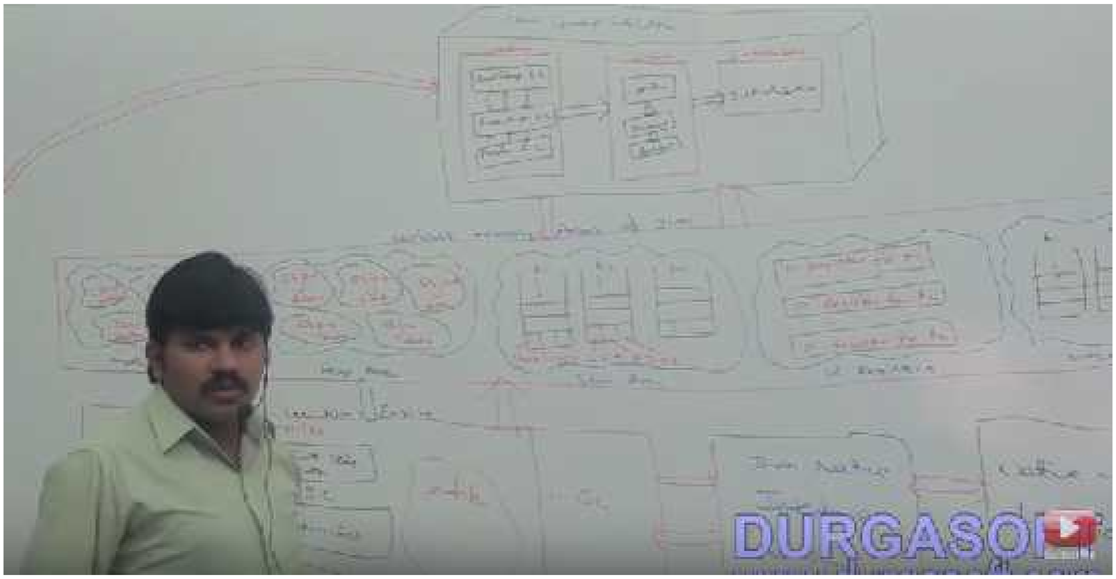
The classes present inside 'ext' folder i.e jdk, jre, lib and ext folder are loaded by extension classloader

Application classloader is responsible to load classes from application classpath.

Bootstrap classpath and classloader will get highest prority.

After loading bytecode verifier will verify is the generated byte code is proper or not and generated by valid compiler or not.

Is it virus or somethig, these things will get vified.



Garbage Collection:

1. Introduction

2. The ways to make an object elegible for GC

3. The methods for requesting JVM to run GC.

4. Finalization

In programming useless objects are garbage.

Java is considered as Robost programming language because someone is there to release the memory and chance of failing a java program is very very less.

The ways to make an object elegible for GC:

Even though programmer is not responsible to destroy useless objects, it is highely recommanded to make an object eligible for GC, if it is no longer required.

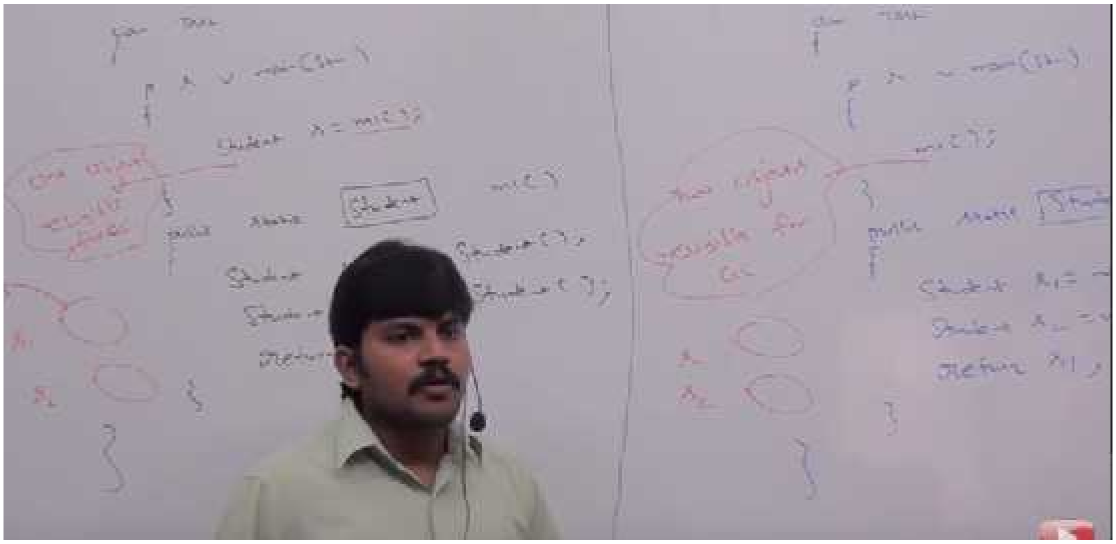
An Object is said to be eleigible for GC if and only if it does not contain any reference variable.

The following are various ways to make an object eleigible for GC,

1. Nullifying the reference variable.(An Object is eligible for garbage collection if there is no live reference)

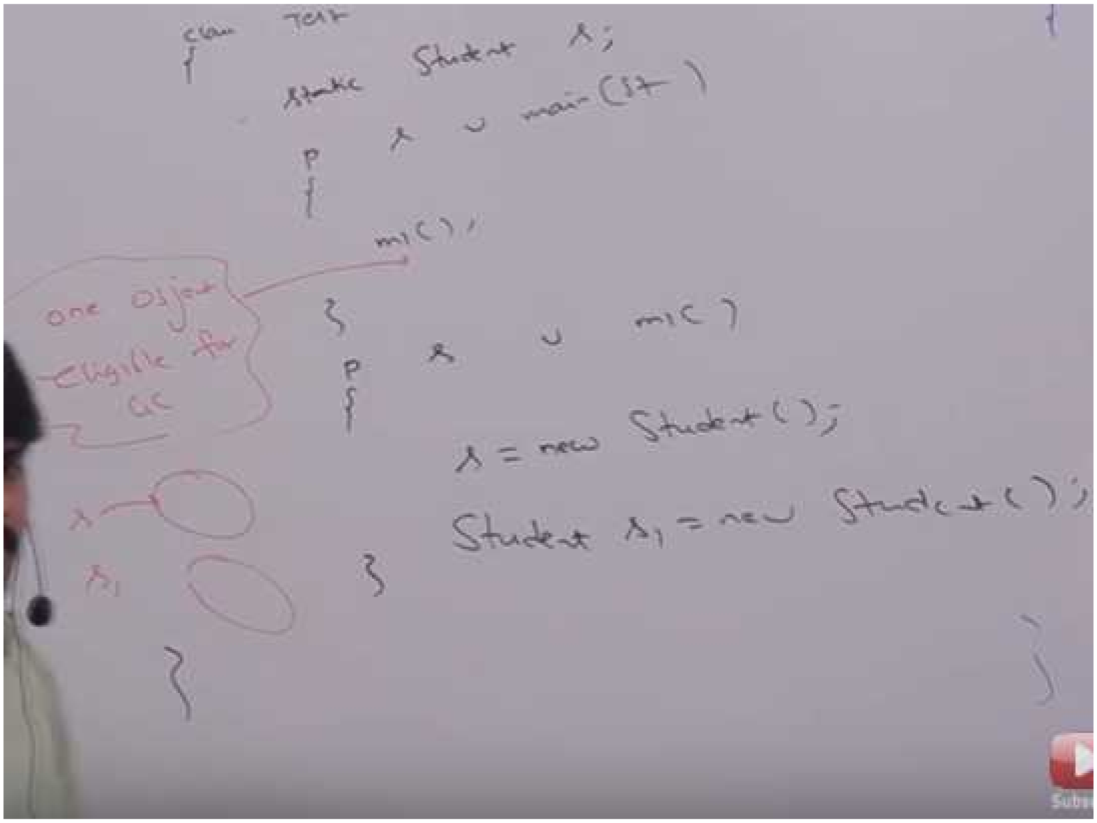
2. Re-assigning the reference variable.

3. Objects created inside a method.



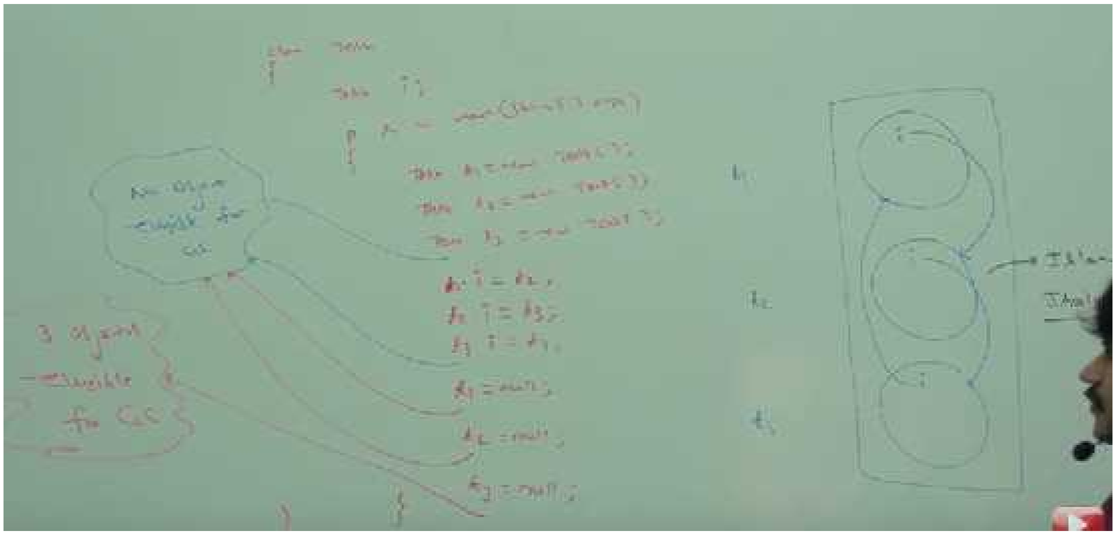
```
class Test
{
  p s v main(Str )
  {
    Student s = m1();
  }
  public static Student m1()
  {
    Student        Student();
    Student        Student();
    return
  }
}
```

```
class Test
{
  p s v main(Str )
  {
    m1();
  }
  public static Stu
  {
    Student s1 = 
    Student s2 = 
    return s1;
  }
}
```



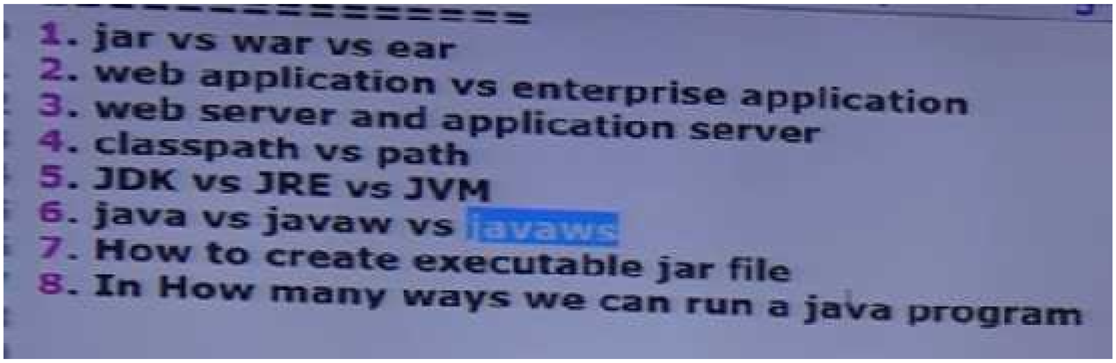
```
class Test
{
  static Student s;
  p s v main(St )
  {
    m1();
  }
  p s v m1()
  {
    s = new Student();
    Student s1 = new Student();
  }
}
```

Island of Isolation:



finalize():

This method is called by GC just before destroying an object to perform clean up activities.



1. jar vs war vs ear
2. web application vs enterprise application
3. web server and application server
4. classpath vs path
5. JDK vs JRE vs JVM
6. java vs javaw vs javaws
7. How to create executable jar file
8. In How many ways we can run a java program

JAR(java archive) vs WAR(web archive) vs EAR(Enterprise archive):

1. A group of .class files is called jar.
2. whole web project(jsps, servlets, xmls) are archived into war'
3. EAR(Servlets, jsps, Ejbs, Jms,....)

JAVA vs JAVAW vs JAVAWS:

- java command is used to run .class file.
- javaw, runs .class file but without console output.
- JAVAWS(Java web start utility.), way to distribute our aplication over the web with centralized control.

Q. How to create executable jar file ?

jar can consistes of many .class files. we need to aware which .class contains main method.

we have to write a manifest.mf file which gives the information about the jar.

jar -cvfm demo8.jar manifest.mf JarDemo.class JarDemo$1.class

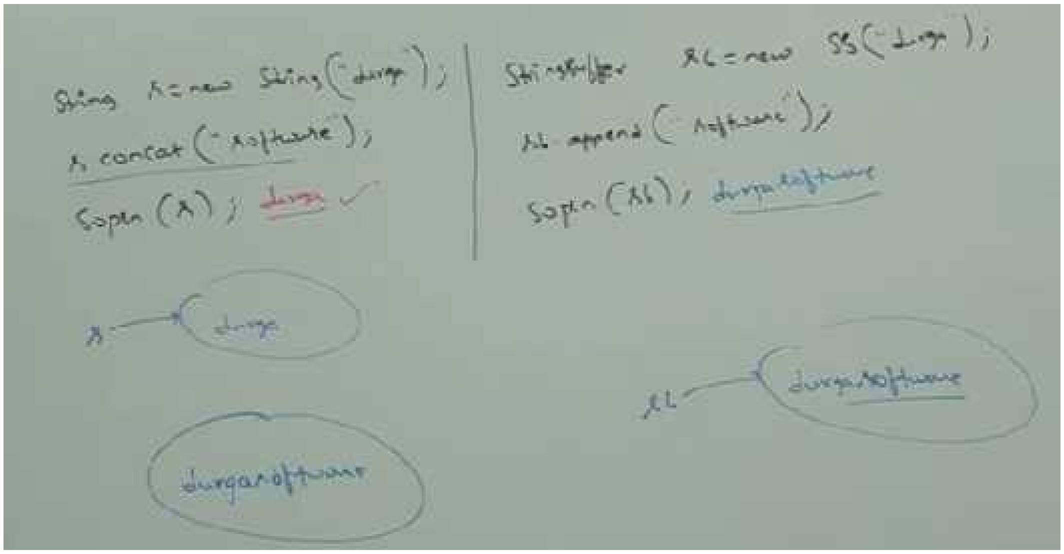
Can create through eclipse also

To run: **java -jar demo8.jar**

===

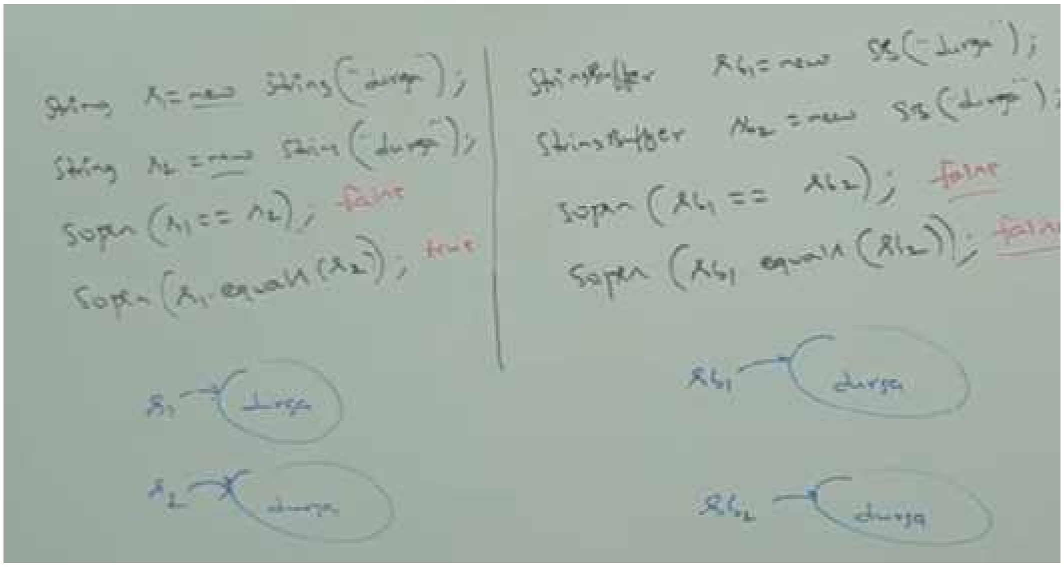
Java.lang.String:

- String is the most commonly used object in java

- 

```
String s=new String("durga");
s.concat("software");
Sopln(s); durga ✓
```

```
StringBuffer sb=new SB("durga");
sb.append("software");
Sopln(sb); durgasoftware
```

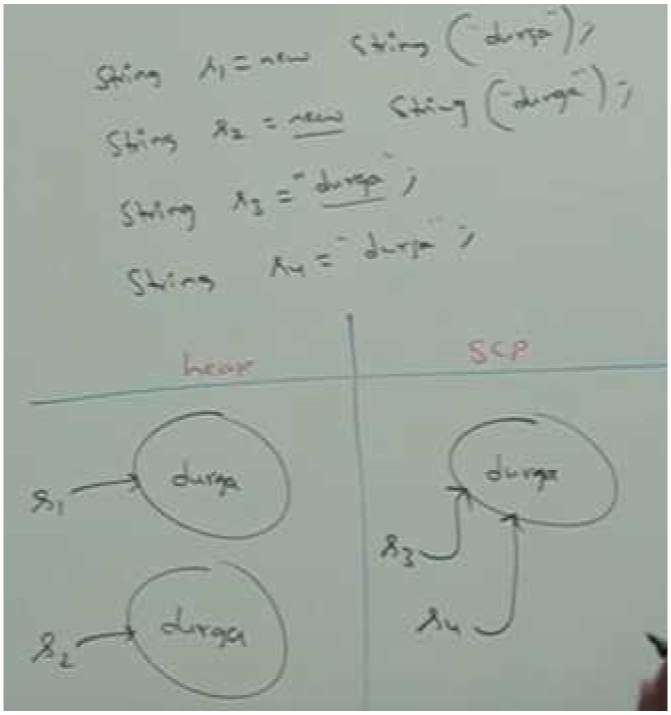
- '==', reference comparison, if two different references points to same object it returns true.
- .equals is overriden in String class to compare contents but it is not overriden in StringBuffer.
- 

```
String s1=new String("durga");
String s2=new String("durga");
Sopln(s1==s2); false
Sopln(s1.equals(s2)); true
```

```
StringBuffer sb1=new SB("durga");
StringBuffer sb2=new SB("durga");
Sopln(sb1== sb2); false
Sopln(sb1 equals(sb2)); false
```

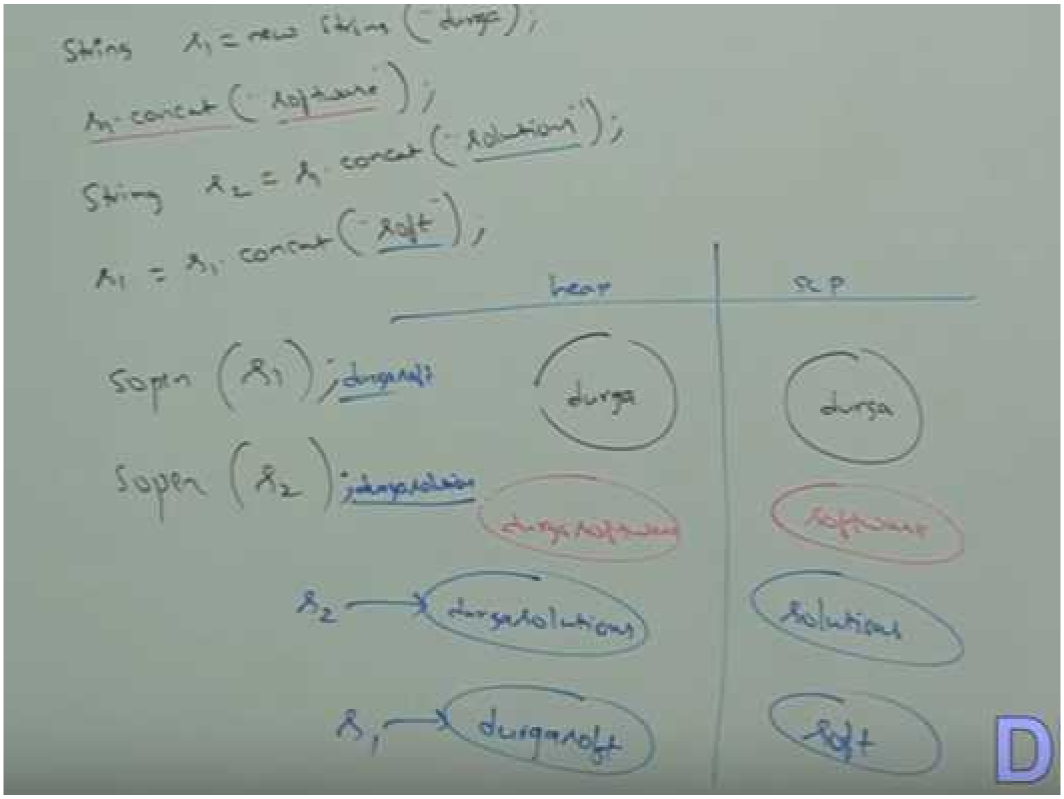
-

- Object creation in scp(string constant pool) is always optional.
- Both procedure will check if same object is there in scp or not.
- GC is not allowed in the SCP area, so all objects on SCP will be destroyed automatically at the time of JVM shutdown.
- SCP is present in Mathod area meory location

- 

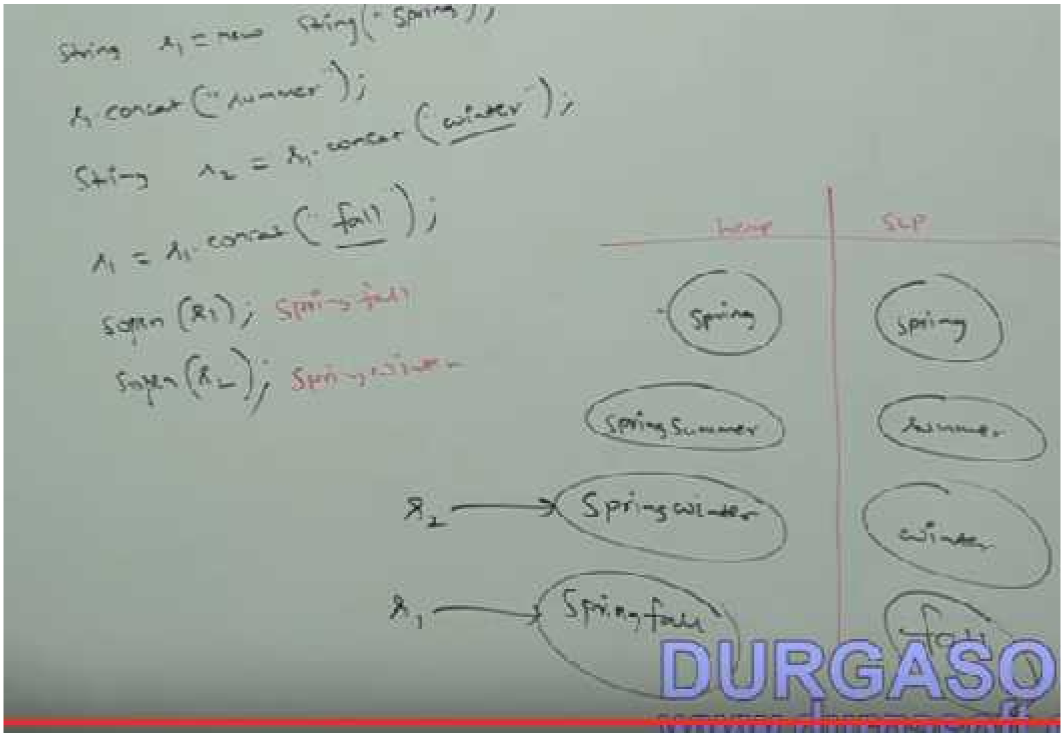
```
String s1 = new String("durga");
String s2 = new String("durga");
String s3 = "durga";
String s4 = "durga";
```

- There is no chance of two objects with same content in scp.

- 

```
String s1 = new String("durga");
s1.concat("software");
String s2 = s1.concat("solutions");
s1 = s1.concat("soft");
Sopln(s1); durgasoft
Sopln(s2); durgasolutions
```

- **Note:**

1. **For every string constant one object will be placed in scp area.**
2. **Because of some runtime operation if an object is required to create that object will be placed only in the heap area but not in scp area.**
3. 

```
String s1 = new String("spring");
s1.concat("summer");
String s2 = s1.concat("winter");
s1 = s1.concat("fall");
sopln(s1); springfall
sopln(s2); springwinter
```

Constructers of String class:

1. String s = new String();

creats an empy string object.

2. String s = new String(String literal);

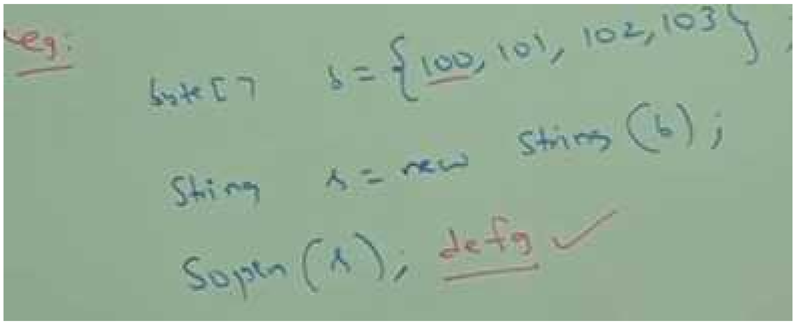
Creats a String object on the heap for the given string literal.

3. String s = new String(StringBuffer sb);

Creats an equivalent string object for the given string buffer.

4. String s = new String(char[] ch);

5. String s = new String(byte[] b);



eg:

```
byte[] b = {100, 101, 102, 103};
String s = new String(b);
Sopln(s); defg ✓
```

Generics in Java:

1.Introduction

2. Generic classes

3.Bounded types

4.Generic methods & wildcard character(?)

5.Communication with non generic code

6. Important conclusions

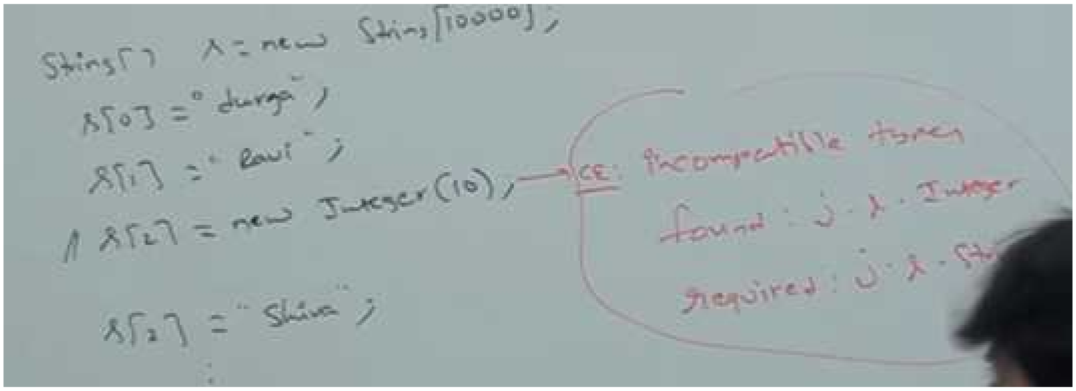
To provide compile time type safty and to resolve type casting problems.

<u>The main objectives of generics are to provide type safty and to resolve type casting prolems.</u>

Type Safty:

Arrays are type safe i.e. we can give the gurantee for the type of elements present inside array.

e.g. if our programming requirement is to hold only string type of objects we can choose string array, by mistake if we are trying to add any other type of objects we will get compile time error.



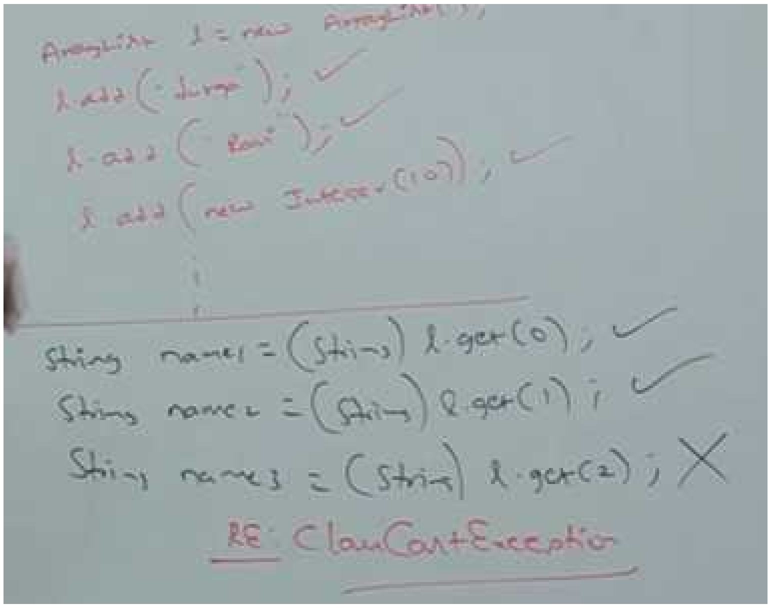
```
String[] s = new String[10000];
  s[0] = "durga";
  s[1] = "ravi";
  s[2] = new Integer(10); → CE: incompatible types
                              found : j.l.Integer
                              required : j.l.St

  s[2] = "Shiva";
     :
```

hence string array can contain only string type of objects.

due to this we can give the gurantee for the type of elements present inside array, hence Arrays are safe to use w.r.t. to type i.e. Arrays are type safe.

But collections are not type safe. i.e. we can't give the gurantee for the type of elements present inside collection.

e.g. if our programming requirement is to hold only string type of objects and if we choose ArryList, by mistake if we are trying any other type of object we won't get any compile time error but the program may fail at runtime.



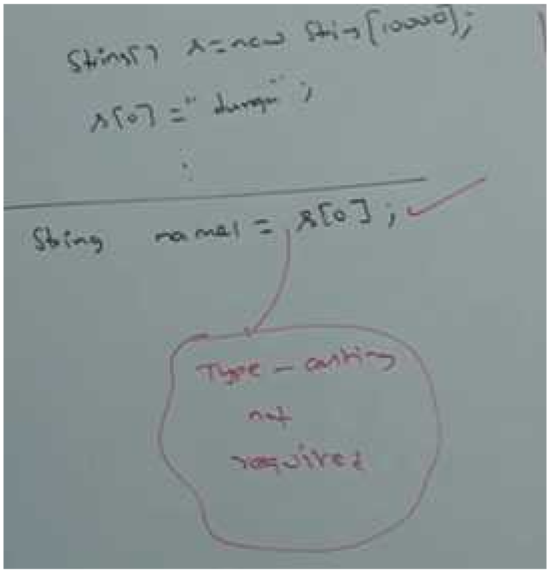
```
ArrayList l = new ArrayList();
l.add("durga"); ✓
l.add("Ravi"); ✓
l.add(new Integer(10)); ✓
      :
      :
String name1 = (String) l.get(0); ✓
String name2 = (String) l.get(1); ✓
String name3 = (String) l.get(2); ✗
        RE: ClassCastException
```

Collections are not safe to use w.r.t type.

Collections are not type safe.

type Casting:

In the case of arrays at the time of retrival it is not required to perform type casting because there is a guranee for the type of elements present inside array.

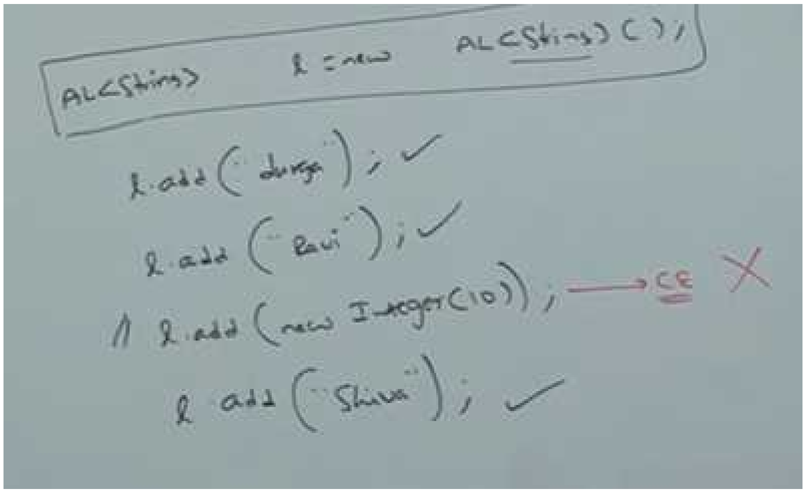


```
String[] s = new String[10000];
s[0] = "durga";
      :
String name1 = s[0]; ✓
```

Type - casting not required

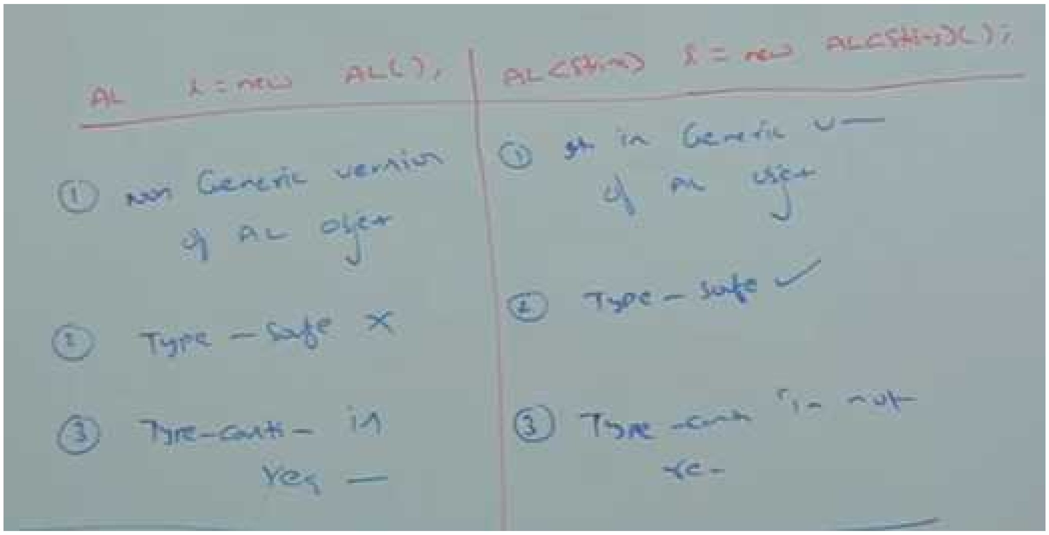
In case of collections type casting is mandatory.

So we need generics



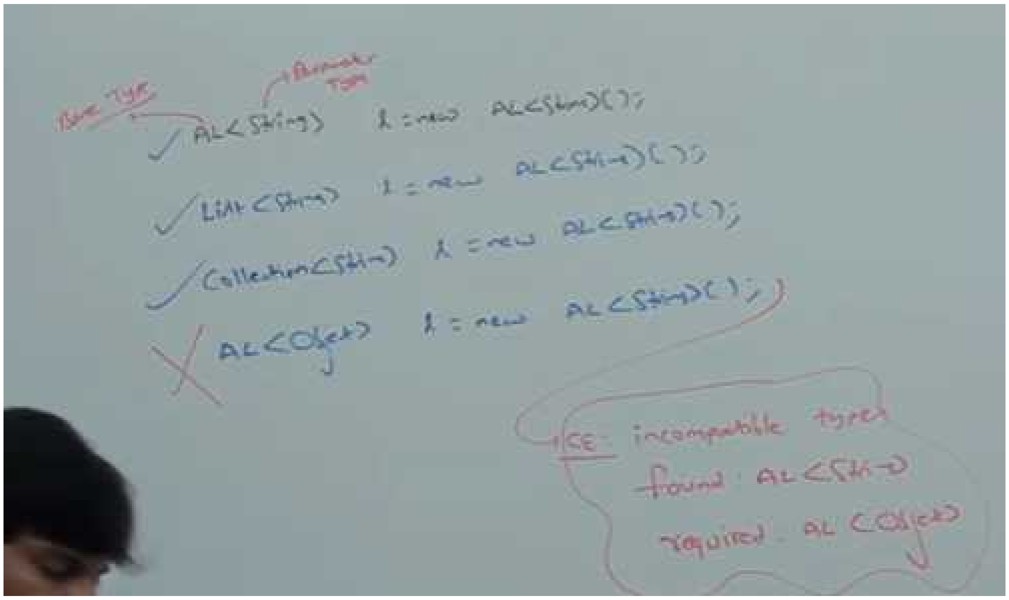
```
AL<String>   l = new   AL<String>();

l.add("durga"); ✓
l.add("Ravi"); ✓
// l.add(new Integer(10)); → CE ✗
l.add("Shiva"); ✓
```

At the time of retrival we are not required to perform type casting.



AL l = new AL();	AL<String> l = new AL<String>();
① non Generic version of AL object	① Generic version of AL object
② Type-safe ✗	② Type-safe ✓
③ Type-casting is required	③ Type-casting is not required

Uses of parent reference to hold child object is called polymorphism.



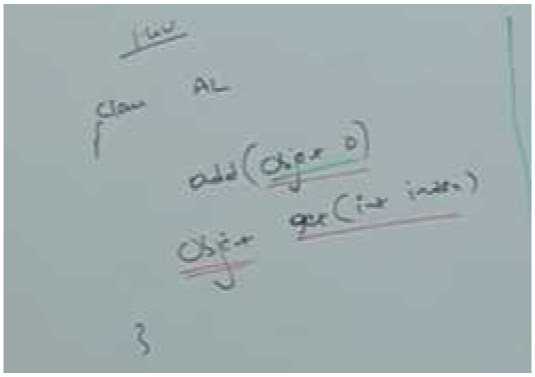
```
✓ AL<String> l = new AL<String>();
✓ List<String> l = new AL<String>();
✓ Collection<String> l = new AL<String>();
✗ AL<Object> l = new AL<String>();

CE: incompatible types
found: AL<String>
required: AL<Object>
```

polimorphism concept applicable only for the base type but not for parameter type.

For the type parameter we can provide any class or interface name but not premitives.

Generic Classes:

Until 1.4 version a non generic version of ArrayList class is declared as follows,

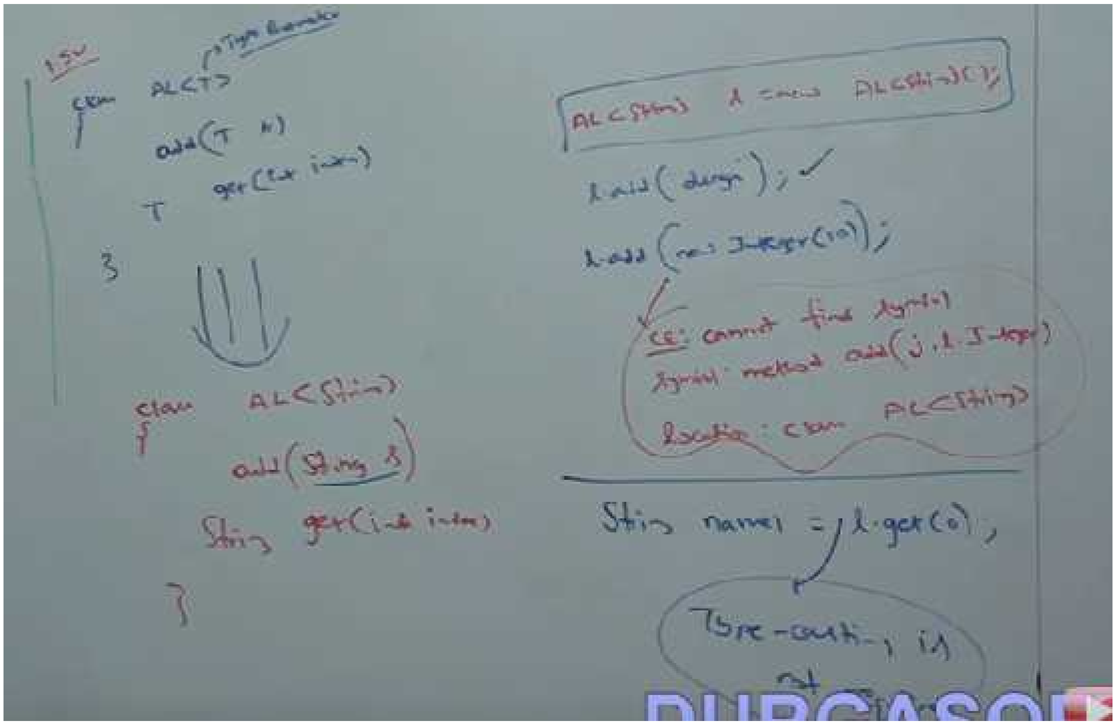


```
class AL
{
    add(Object o)
    Object get(int index)
}
```

The argument to add method is object and hence we can add any type of object to the arrayList. due to this we are missing type safty.

The return type of get method is Object. hence at the time of retrival we have to perform type casting.

But in 1.5 version a generic version of ArrayList class is declared as follows,



T is the type parameter.

Based on our runtime requirement T will be replaced with our provided type.

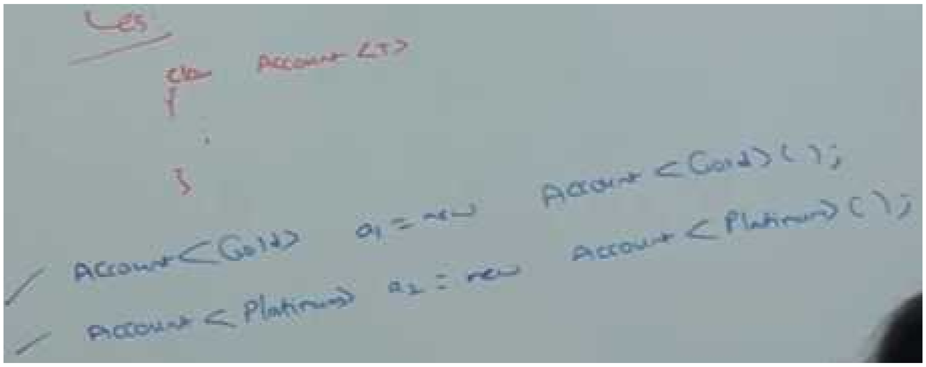
e.g. To hold only String type of Objects a generic version of ArrayList can be created as above.

In generics we are associating a type parameter to the class.

Such type of parameterized classes are nothing but generic classes or templet classes.

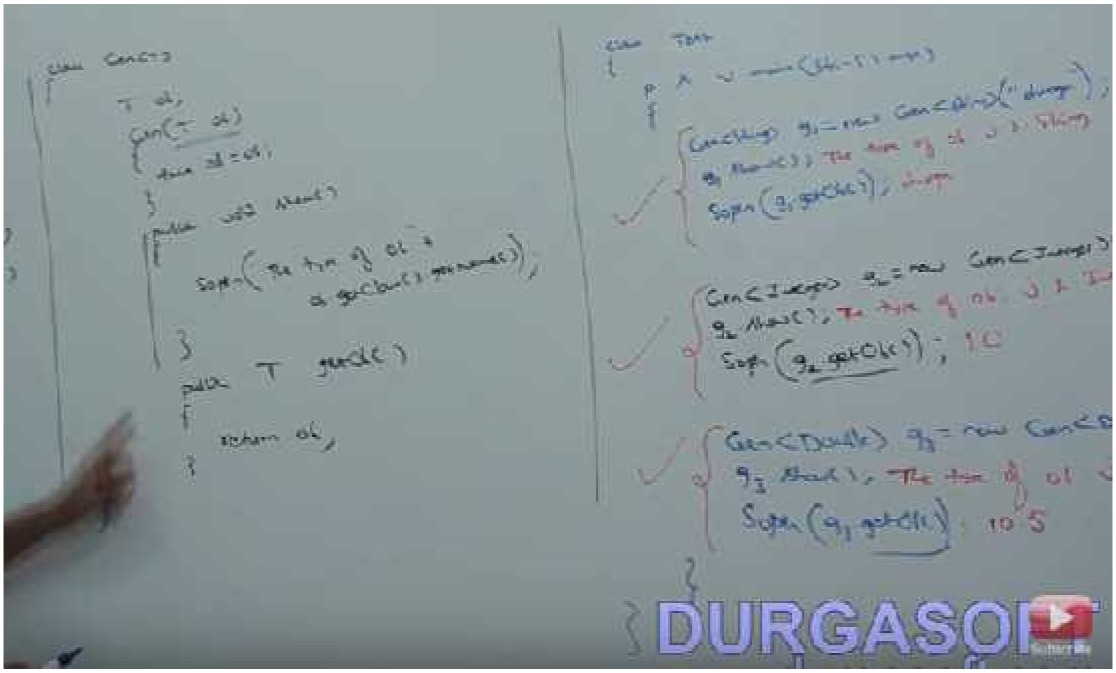
Based on our requirement we can define our own Generic classes also.

e.g.



```
Ues

class   Account<T>
{
   .
}

Account<Gold>  a1 = new  Account<Gold>();
Account<Platinum>  a2 = new  Account<Platinum>();
```

Our Own Generic Class:



```
class Gen<T>
{
    T ob;
    Gen(T ob)
    {
        this.ob=ob;
    }
    public void show()
    {
        Sopln("The type of ob is "+
                ob.getClass().getName());
    }
    public T getOb()
    {
        return ob;
    }
}
```

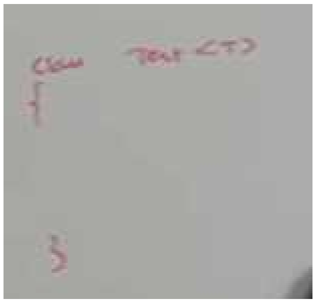
```
class Test
{
    p s v main(String[] args)
    {
        Gen<String> g1=new Gen<String>("durga");
        g1.show(); The type of ob is String
        Sopln(g1.getOb()); durga

        Gen<Integer> g2=new Gen<Integer>
        g2.show(); The type of ob is In
        Sopln(g2.getOb()); 10

        Gen<Double> g3=new Gen<D
        g3.show(); The type of ob
        Sopln(g3.getOb()) 10.5
    }
}
```

Bounded Types:

We can bound the type parameter for a particular range by using 'extends' keyword.

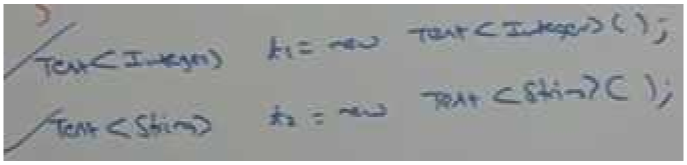
such types are called **bounded types**.

e.g.



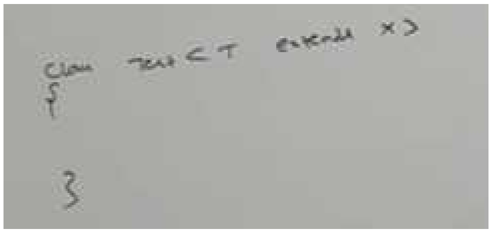
```
class Test<T>
{

}
```

At the type parameter we can pass any type and there are no restrictions and hence it is unbounded type.



```
Test<Integer> t1 = new Test<Integer>();
Test<String> t2 = new Test<String>();
```

Syntax for bounded type:

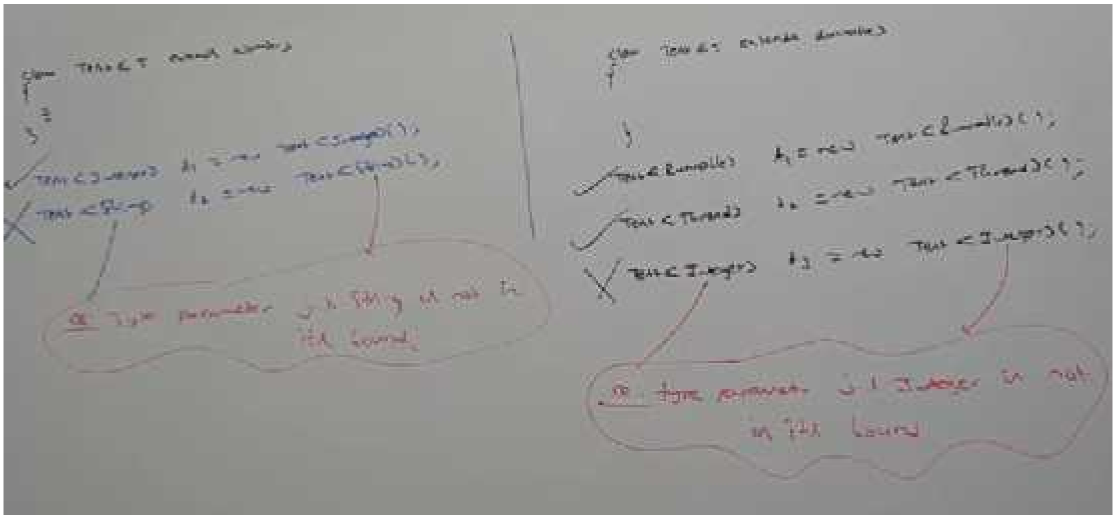


```
class Test<T extends X>
{

}
```

'x' can be either class or interface.

if 'x' is a class then as a type parameter we can pass either 'x' type or it's child classes.

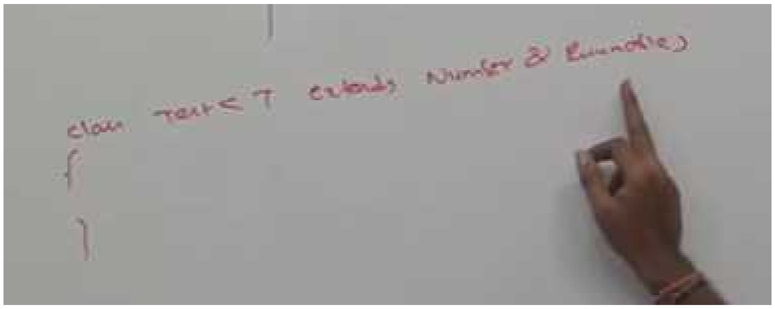
if 'x' is an interface then as a type parameter we can pass either 'x' type or it's implementation classes.



```
class Test<T extends Number>
{
}
✓ Test<Integer> t1 = new Test<Integer>();
✗ Test<String> t2 = new Test<String>();
CE: type parameter java.lang.String is not within its bound;

class Test<T extends Runnable>
{
}
✓ Test<Runnable> t1 = new Test<Runnable>();
✓ Test<Thread> t2 = new Test<Thread>();
✗ Test<Integer> t3 = new Test<Integer>();
CE: type parameter java.lang.Integer is not within its bound
```

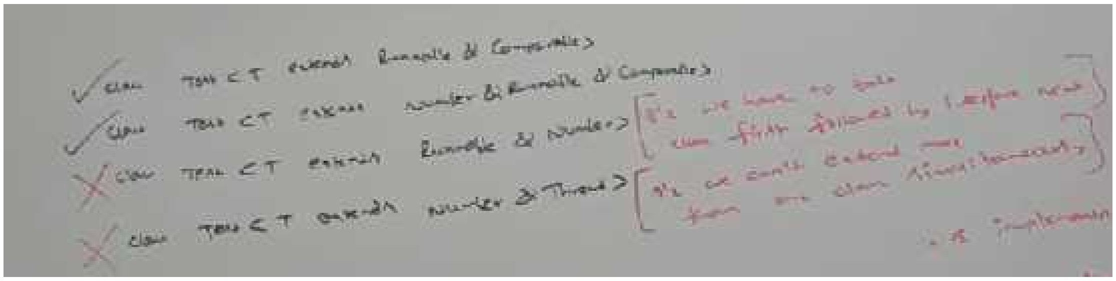
We can defeine bounded types even in combination also.

e.g.



```
class Test<T extends Number & Runnable>
{
}
```

As a type parameter we can take anything which should be child class of number and should implements runnable inteface.

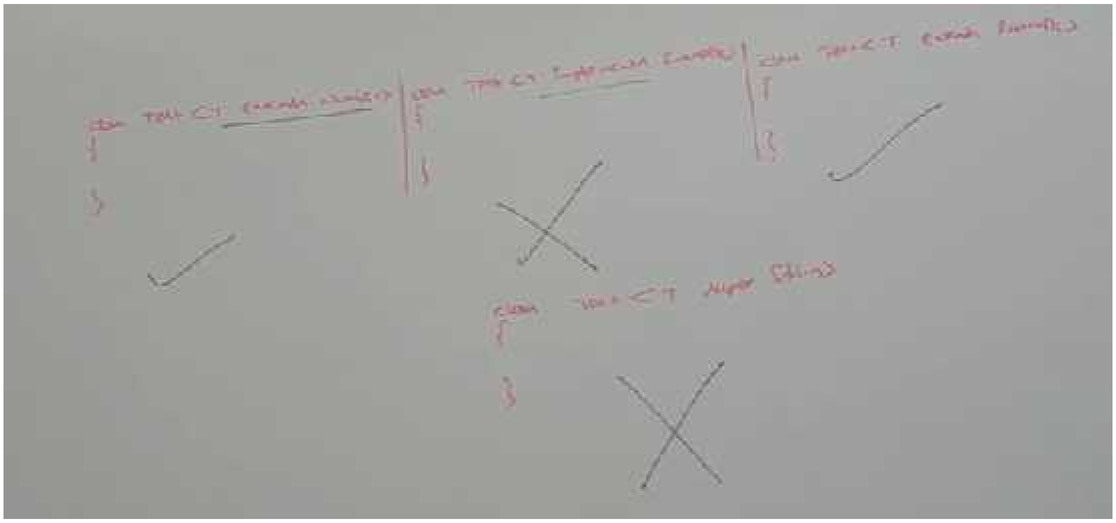


```
✓ class Test <T extends Runnable & Comparable>
✓ class Test <T extends Number & Runnable & Comparable>
✗ class Test <T extends Runnable & Number>   [ i.e., we have to take class first followed by interface next ]
✗ class Test <T extends Number & Thread>     [ i.e., we can't extend more than one class simultaneously ]
```

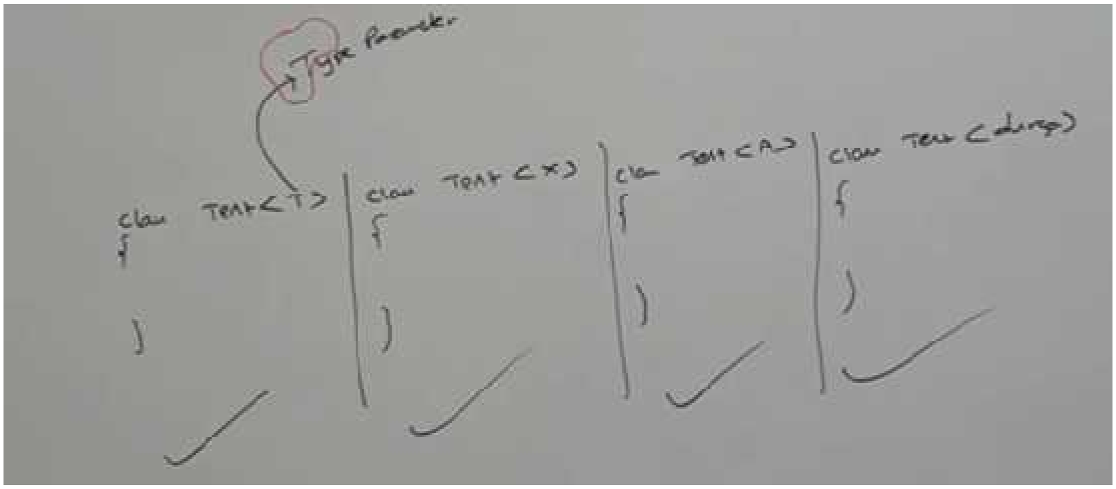
Note:

1. We can define bounded types only by using extend keyword and we can't use implements and supper keywords but we can

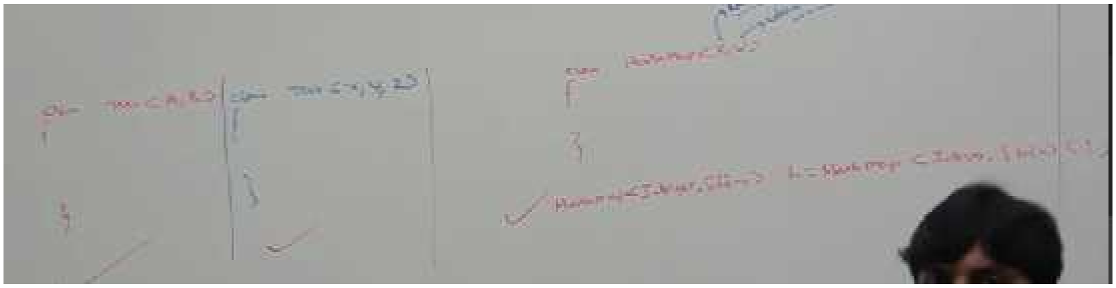
replace implements keyword purpose with extends keyword.


class Test<T extends Number>
{
}
class Test<T implements Runnable>
{
}
class Test<T extends Runnable>
{
}
class Test<T super String>
{
}

2. As a type parameter 'T', we can take any valid java identifier but it is convention to use 'T'.

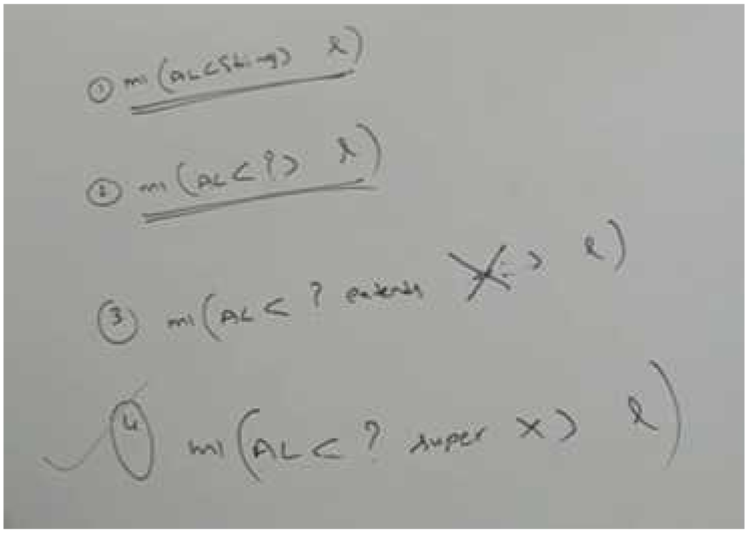


3. Based on our requirement we can declare any number of type parameters and all these type parameters should be separated with ','.



```
class Test<A,B,C>      class Test<X,Y,Z>          class HashMap<K,V>
{                      {                          {
}                      }                          }
                                                  HashMap<Integer,String> h=new HashMap<Integer,String>();
```

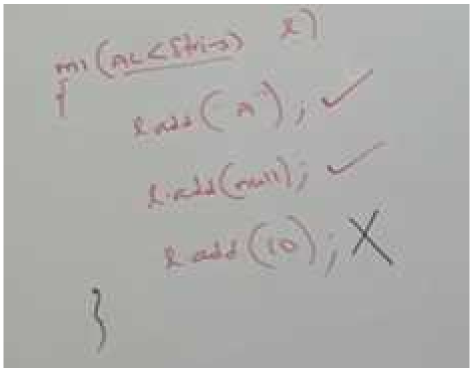
Generic Methods And WildCard characters (?):



```
① m1(AL<String> l)
② m1(AL<?> l)
③ m1(AL<? extends X> l)
④ m1(AL<? super X> l)
```

1. we can call this method by paassing ArraList of only string type. But within the method we can add only string type of objects to the list. By mistake if we are to add any other type then we will get compile time error.

e.g.

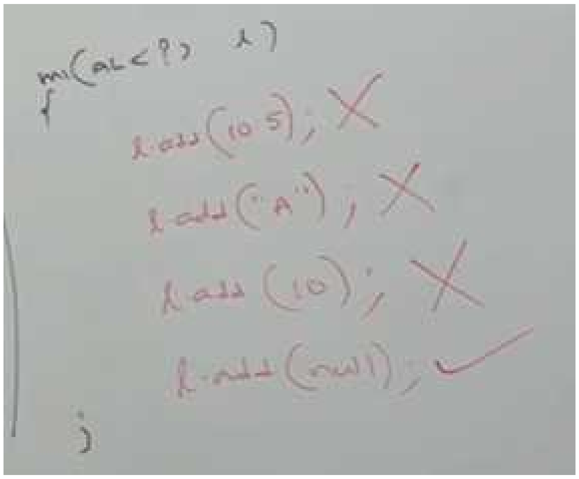


```
m1(AL<String> l)
{
    l.add("A"); ✓
    l.add(null); ✓
    l.add(10); X
}
```

2. m1(ArrayList<?> l)

We can call this method by passing ArrayList of any unknown type. But within the method we can't add anything to the list except null, because we don't know the type exactly.

Null is allowed because it is valid value for any type.



```
m1(AL<?> l)
{
    l.add(10.5); ✗
    l.add("A"); ✗
    l.add(10); ✗
    l.add(null); ✓
}
```

This type of methods are best suitable for readonly operation.

3. **m1(ArrayList<? extends x> l)**

'x ' can be either class or interface.

if 'x' is a class then we can call this method by passing ArrayList of either 'x' type or it's child classes.

if 'x' is an interface then we can call this method by passing ArryList of either 'x' type or it's implementation classes.

But within the method we can't add anything to the list except null, because we don't know the type of 'x' exactly.

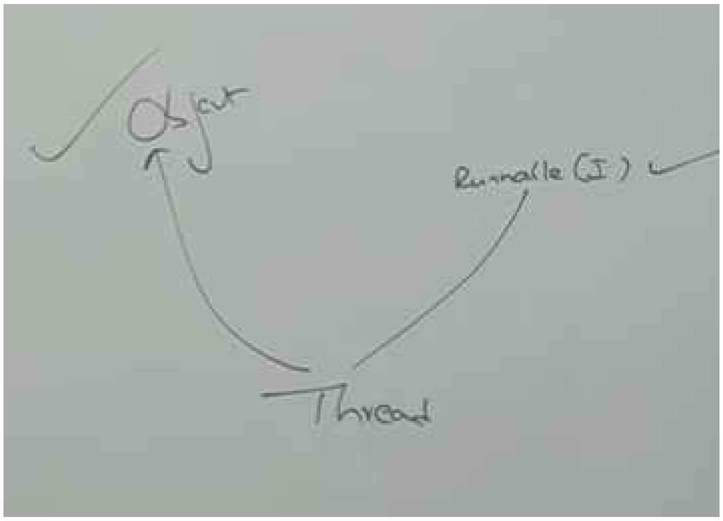
This type of methods also best suitable for readonly operations.

4. **m1(ArrayList<? super x> l)**

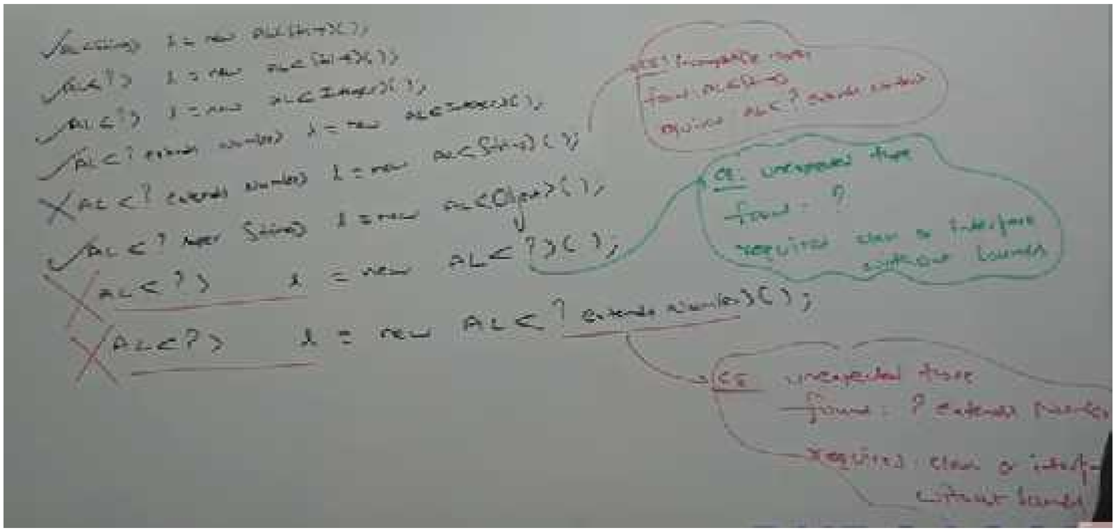
'x' can be either class or interface

if 'x' is a class then we can call this method by passing ArrayList of either 'x' type or it's super classes.

if 'x' is an interface then we can call this method by passing arrayList of either 'x' type or **super class of implementation class of 'x'.**



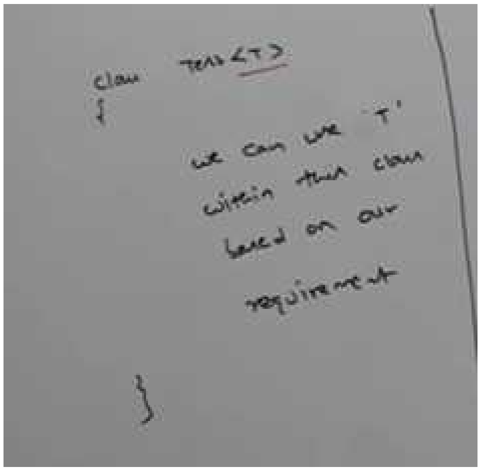
But within the method we can add 'x' type of objects and null to the list.


AL<String> l = new AL<String>();
AL<?> l = new AL<String>();
AL<?> l = new AL<Integer>();
AL<? extends Number> l = new AL<Integer>();
AL<? extends Number> l = new AL<String>();
AL<? super String> l = new AL<Object>();
AL<?> l = new AL<?>();
AL<?> l = new AL<? extends Number>();
CE: incompatible types
found: AL<String>
required: AL<? extends Number>
CE: unexpected type
found: ?
required: class or interface without bounds
CE: unexpected type
found: ? extends Number
required: class or interface without bounds

bounded types only valid on reference types.

We can declare type parameter either at class level or at method level.

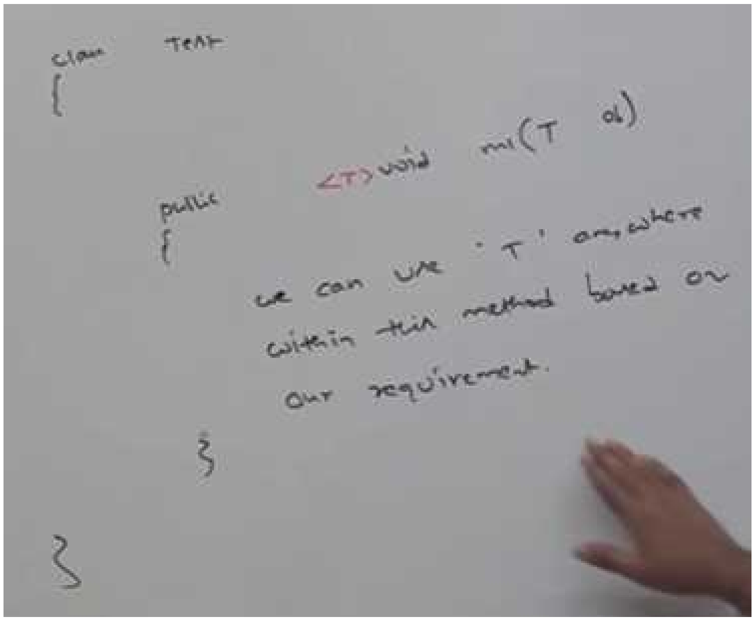
Declaring type parameter at class level:



```
class Test<T>
{
    we can use 'T'
    within this class
    based on our
    requirement
}
```

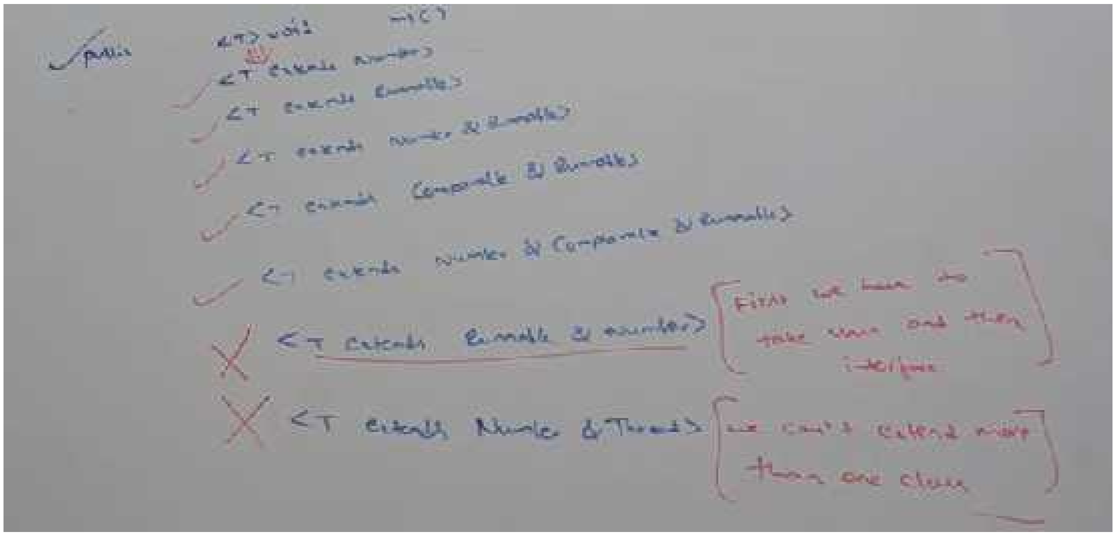
Declaring type parameter at method level:

We have to declare type parameter just befor return type,



```
class Test
{
    public <T> void m1(T ob)
    {
        we can use 'T' anywhere
        within this method based on
        our requirement.
    }
}
```

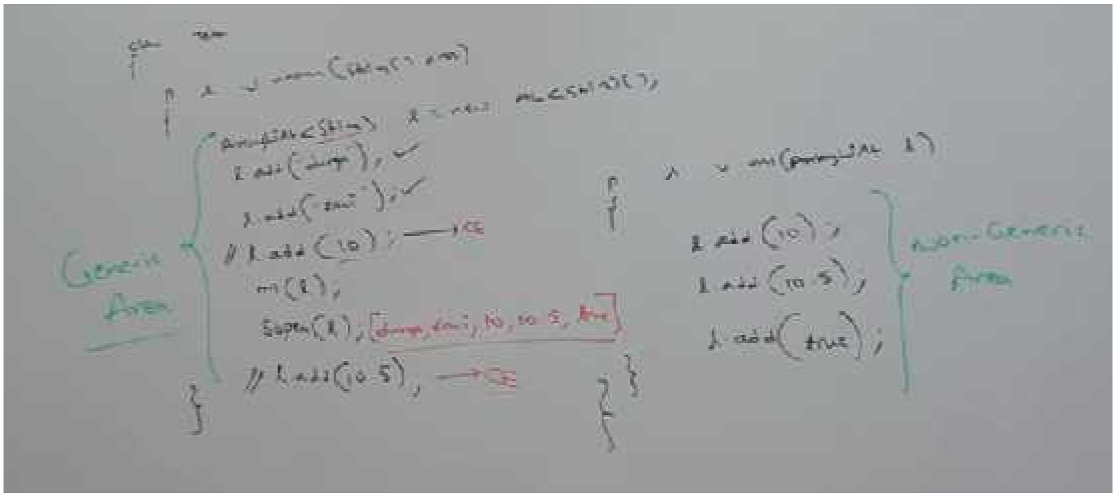
We can define bounded types even at method level also.



```
public  <T> void      m1(T)
        <T extends Number>
        <T extends Runnable>
        <T extends Number & Runnable>
        <T extends Comparable & Runnable>
        <T extends Number & Comparable & Runnable>
        <T extends Runnable & Number>   [First we have to take class and then interface]
        <T extends Number & Thread>     [we can't extend more than one class]
```

communication with non-generic code:

If we send generic object to non generic area then it starts behaving like non generic object, similarly if we send non generic object to generic area then it starts behaving like generic object, i.e. the location in which object present based on that behaviour will be defined.



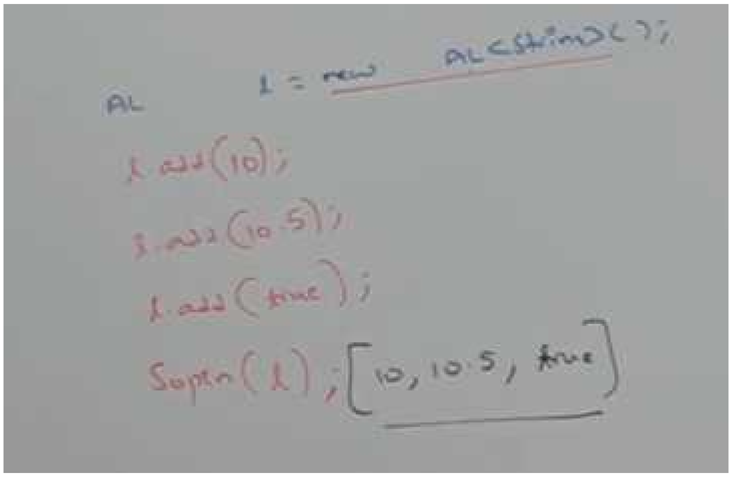
```
class Test
{
  p s v main(String[] args)
  {
    ArrayList<String> l = new ArrayList<String>();
    l.add("durga");  ✓
    l.add("ravi");   ✓
    // l.add(10);  → CE
    m1(l);
    S.o.pln(l);  [durga, ravi, 10, 10.5, true]
    // l.add(10.5);  → CE
  }
  p s v m1(ArrayList l)
  {
    l.add(10);
    l.add(10.5);
    l.add(true);
  }
}
```

Generic Area
Non-Generic Area

The main purpose of generics is to provide type safty and to resolve type casting problems.

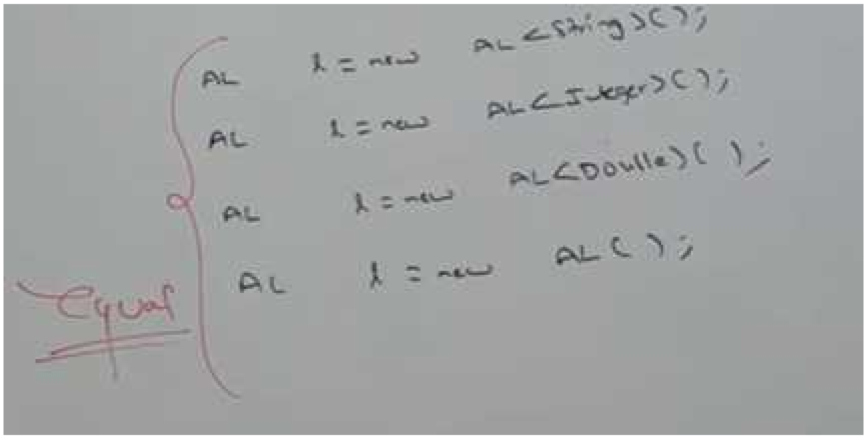
Type safety and type casting both are applicable at compile time hence generics concept also applicable only at compile time but not at runtime.

At the time of compilation at last step generic syntax will be removed and hence for the jvm generic syntax won't be available.



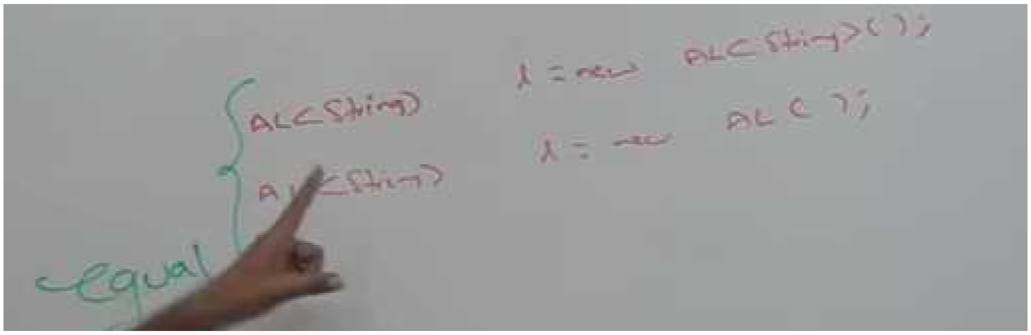
```
AL        l = new   AL<String>();
l.add(10);
l.add(10.5);
l.add(true);
Sopln(l); [10, 10.5, true]
```

Hence the following declarations are equal,

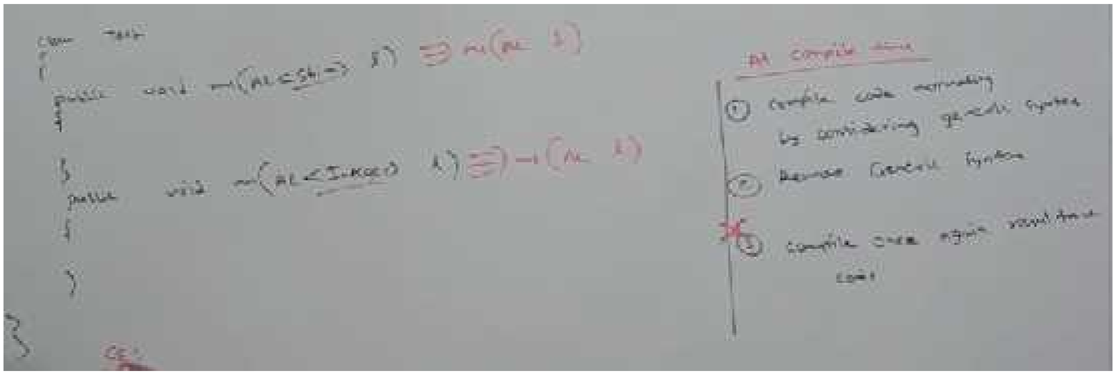


```
      AL     l = new    AL<String>();
      AL     l = new    AL<Integer>();
      AL     l = new    AL<Double>( );
Equal AL     l = new    AL( );
```

The following declarations are equal.



Another case:

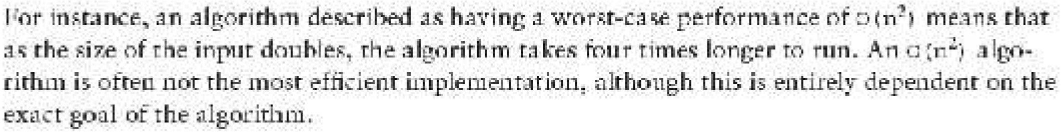


```
class Test
{
  public void m(AL<String> l) ⇒ m(AL l)
  {
  }
  public void m(AL<Integer> l) ⇒ m(AL l)
  {
  }
}
CE:
```

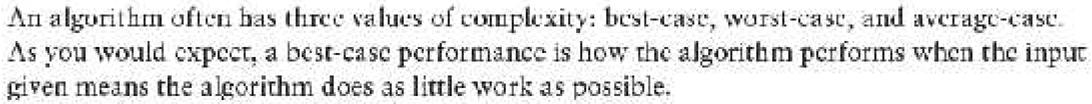
At compile time

1. compile code normally by considering generic syntax
2. Remove Generic Syntax
3. compile code again normally

Algorithems Summary:

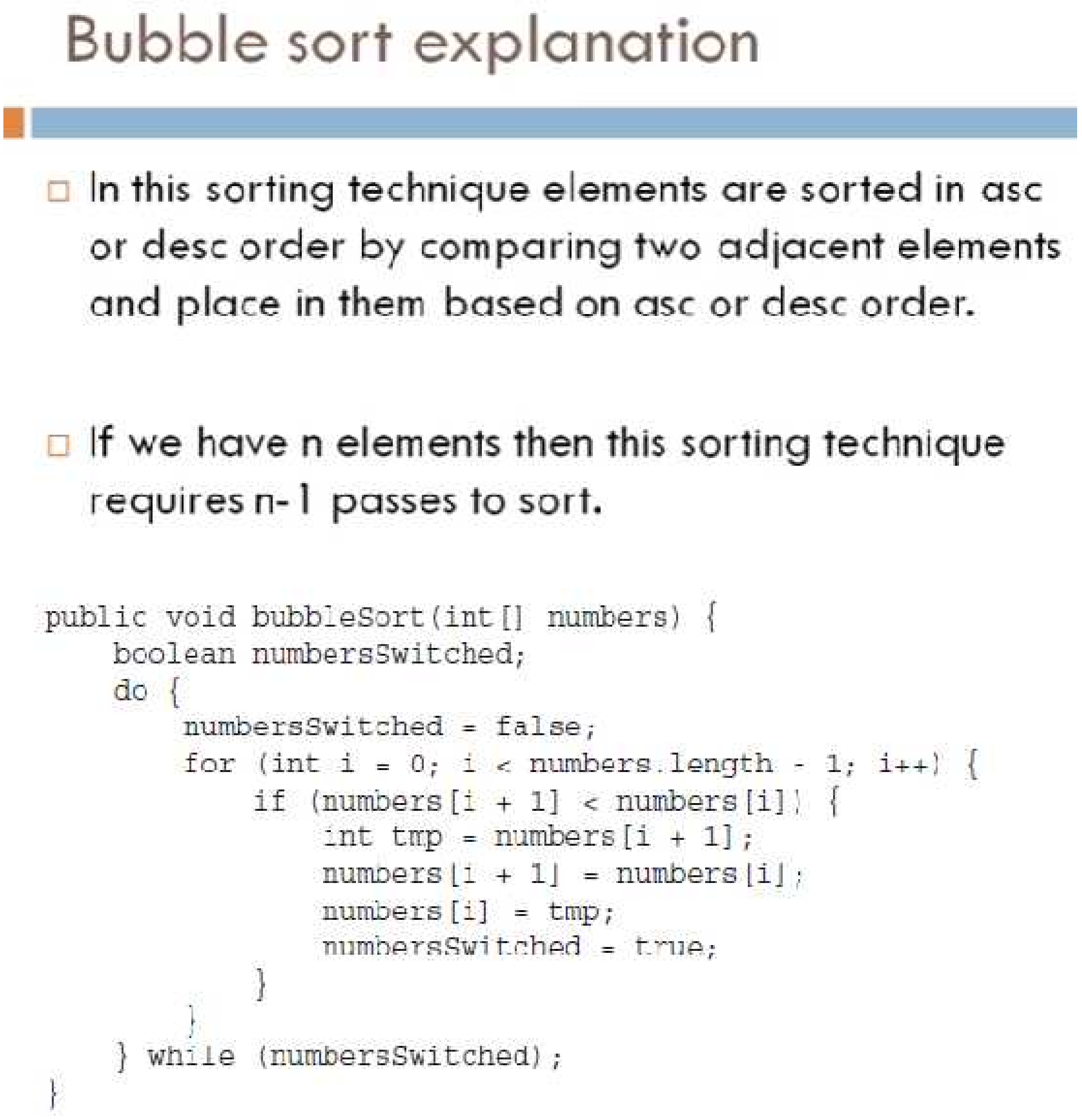


For instance, an algorithm described as having a worst-case performance of $O(n^2)$ means that as the size of the input doubles, the algorithm takes four times longer to run. An $O(n^2)$ algorithm is often not the most efficient implementation, although this is entirely dependent on the exact goal of the algorithm.



An algorithm often has three values of complexity: best-case, worst-case, and average-case. As you would expect, a best-case performance is how the algorithm performs when the input given means the algorithm does as little work as possible.

The performance descriptions here are often called the *time complexity* of the algorithm. Algorithms have a *space complexity*, too; that is, how much extra space the algorithm needs to do its work.



Bubble sort explanation

- In this sorting technique elements are sorted in asc or desc order by comparing two adjacent elements and place in them based on asc or desc order.
- If we have n elements then this sorting technique requires n-1 passes to sort.

```
public void bubbleSort(int[] numbers) {
    boolean numbersSwitched;
    do {
        numbersSwitched = false;
        for (int i = 0; i < numbers.length - 1; i++) {
            if (numbers[i + 1] < numbers[i]) {
                int tmp = numbers[i + 1];
                numbers[i + 1] = numbers[i];
                numbers[i] = tmp;
                numbersSwitched = true;
            }
        }
    } while (numbersSwitched);
}
```

Although this implementation is simple, it is extremely inefficient. The worst case, when you want to sort a list that is already sorted in reverse order, is a performance of O(n2): For each iteration, you are only switching one element. The best case is when a list is already sorted: You make one pass through the list, and because you have not switched any elements, you can stop. This has a performance of O(n).

Inner Classes:

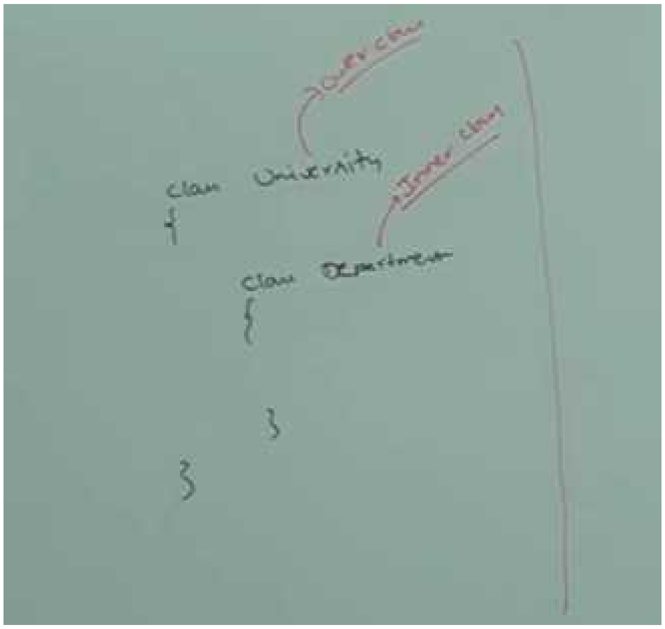
sometimes we can declare a class inside another class such type of classes are called inner classes.

Inner classes concept introduced in 1.1 version to fix GUI bugs as part of event handling but beacuse of powerful features and benifits of inner classes slowly programmers started using in regular coding also.

Without existing one type of object if there is no change of existing another type of object then we should go for inner clases.

e.g.

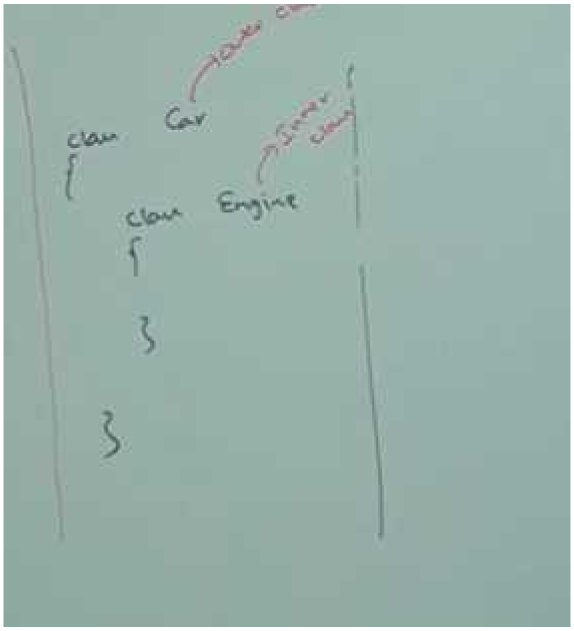
University consists of several departments, without existing university there is no change of existing department, hence we have to declare departmrnt class inside university class.



```
class University
{
    class Department
    {

    }
}
```

e,g. 2

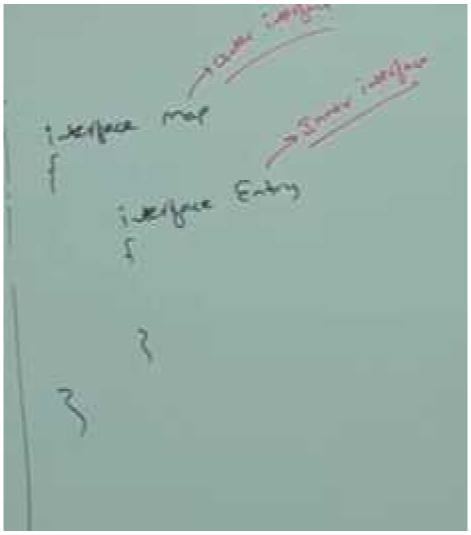
without existing car object there is no chance of existing engine object hence we have to declare engine class inside car class.



```
class Car
{
    class Engine
    {

    }

}
```

e.g. 3

Map is a group of key value pairs and each key value pair is called an entry. without existing map object there is no chance of existing entry object hence interface entry is defined inside Map interface.



```
interface map
{
    interface Entry
    {

    }
}
```

Note:

1. without existing outer class object there is no chance of existing

inner class object.

2. The relation between outer class and inner class is not IS-A relation and it is HAS-A relationship.(Composition or Aggregation)

Based on position of declaration and behaviour all inner classes are divided into 4 types,

1. Normal or Regular Inner classes.

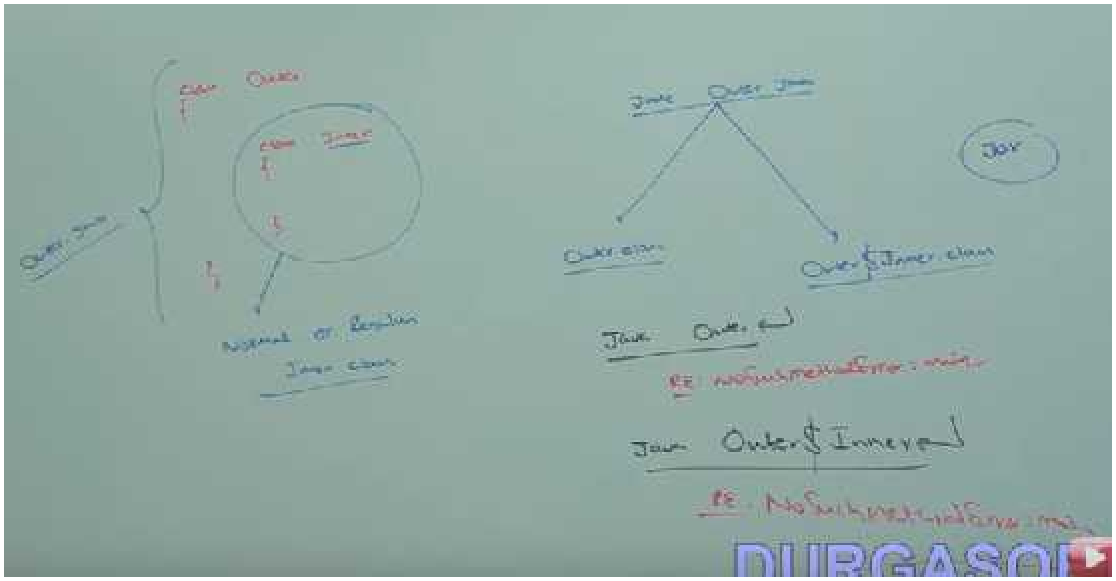
2. Method Local inner classes.

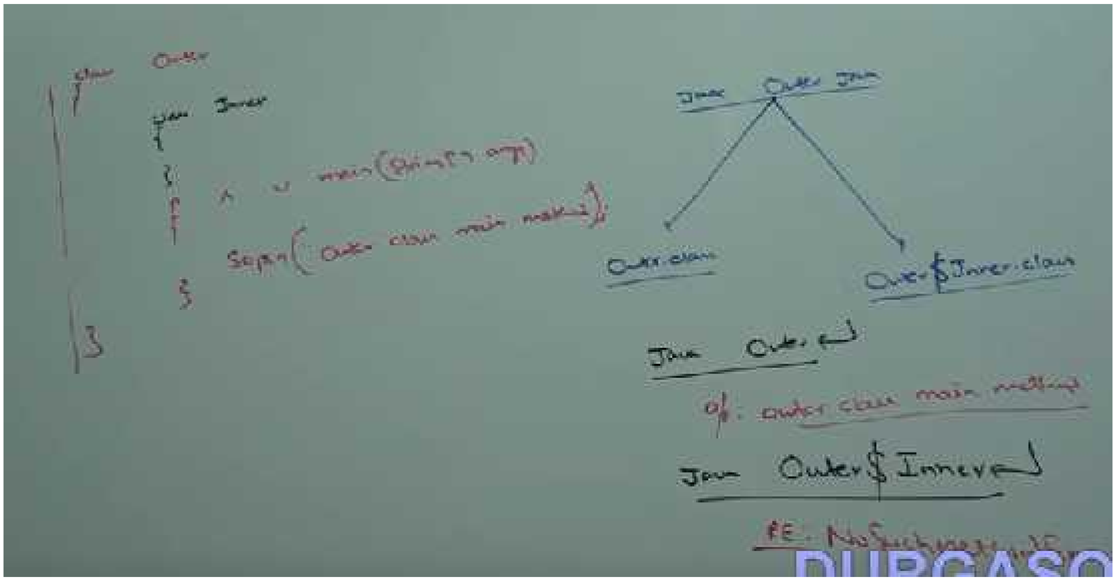
3. Annonymous Inner Classes.

4. Static Nested Classes.

Normal or regular Inner Classes:

If we are declaring any named class direcly inside a class without static modifier such type of inner class is called normal or regular inner class.





```
class Outer
{
    class Inner
    {
    }
    p s v main(String[] args)
    {
        Sopln("outer class main method");
    }
}
```

javac Outer.java

Outer.class

Outer$Inner.class

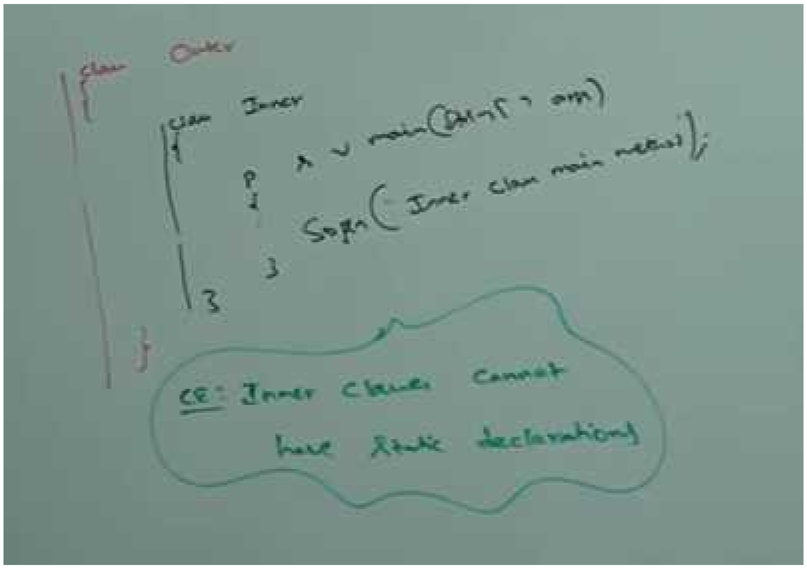
java Outer ↵

o/p: outer class main method

java Outer$Inner ↵

RE: NoSuchMethodError

Inside inner class we can't declare any static members hence we can't declare main() method and we can't run inner class directly from command prompt.

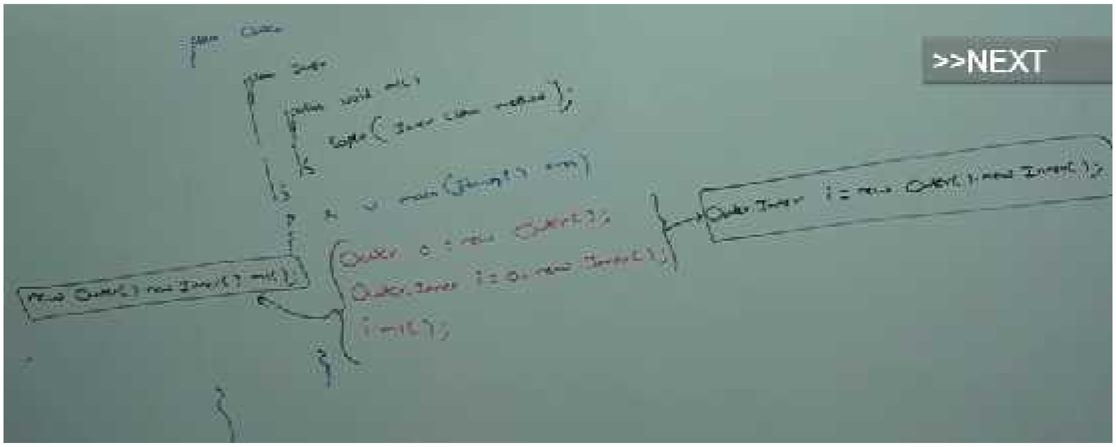


```
class Outer
{
    class Inner
    {
        p s v main(String[] args)
        {
            Sopln("Inner class main method");
        }
    }
}
```

CE: Inner classes cannot have static declarations

Case 1:

Accessing inner class code from static area of outer class:



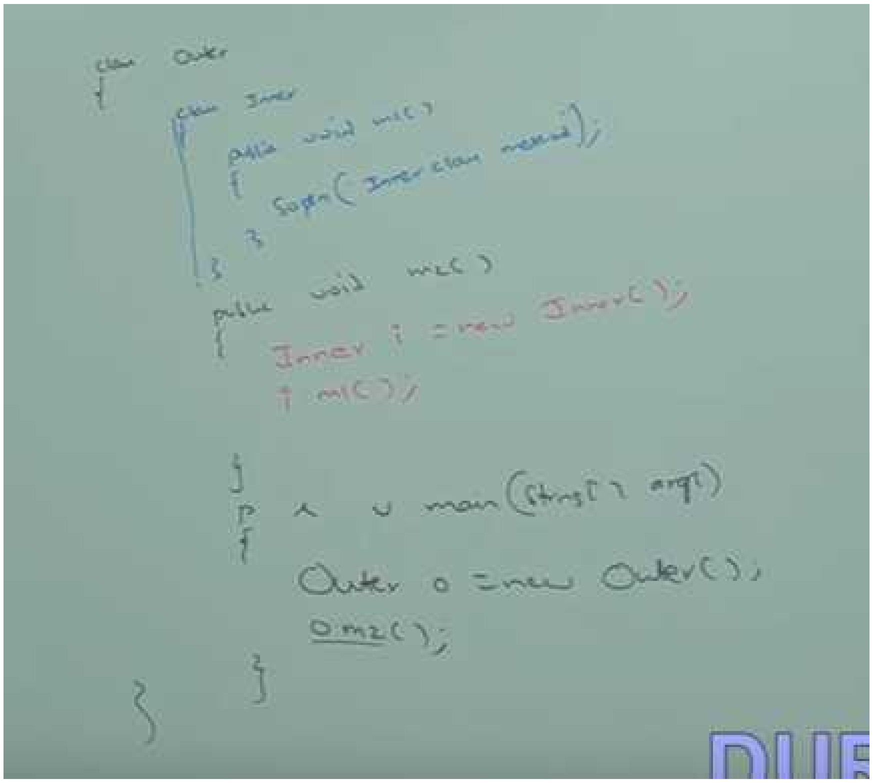
```
class Outer
{
    class Inner
    {
        public void m1()
        {
            Sopln("Inner class method");
        }
    }
    p s v main(String[] args)
    {
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();
        i.m1();
    }
}
```

```
new Outer().new Inner().m1();
```

```
Outer.Inner i = new Outer().new Inner();
```

Case 2:

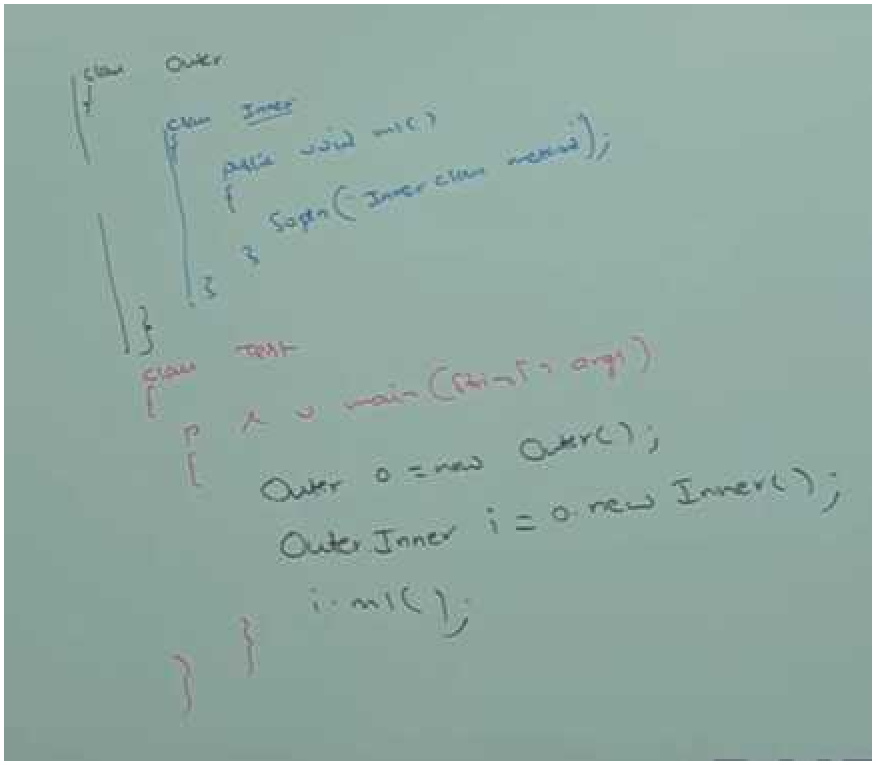
Accessing inner class code from instance area of outer class:



```
class Outer
{
    class Inner
    {
        public void m1()
        {
            Sopln("Inner class method");
        }
    }
    public void m2()
    {
        Inner i = new Inner();
        i.m1();
    }
    p s v main(String[] args)
    {
        Outer o = new Outer();
        o.m2();
    }
}
```

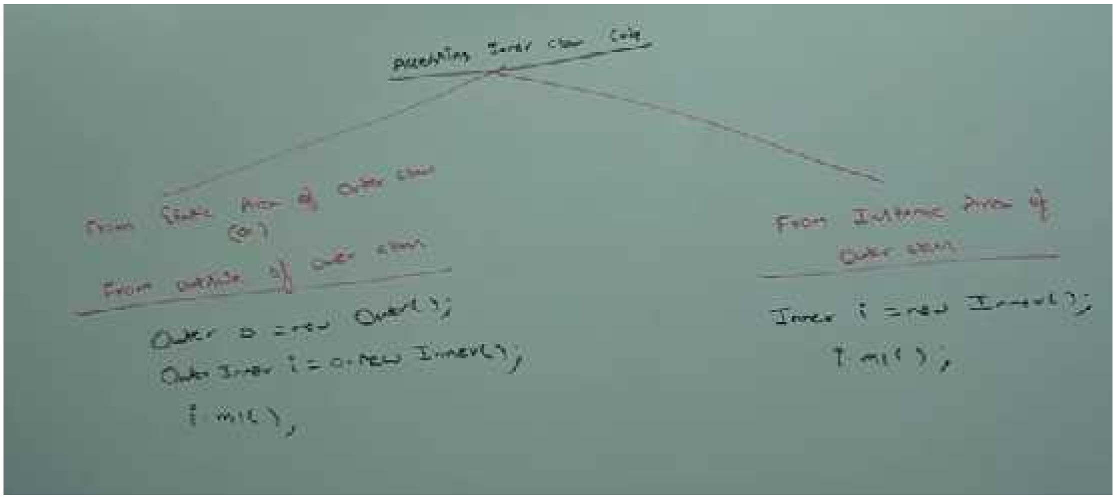
Case 3:

Accessing inner class code from outside of outer class



```
class Outer
{
  class Inner
  {
    public void m1()
    {
      Sopln("Inner class method");
    }
  }
}
class Test
{
  p s v main(String[] args)
  {
    Outer o = new Outer();
    Outer.Inner i = o.new Inner();
    i.m1();
  }
}
```

Summary:

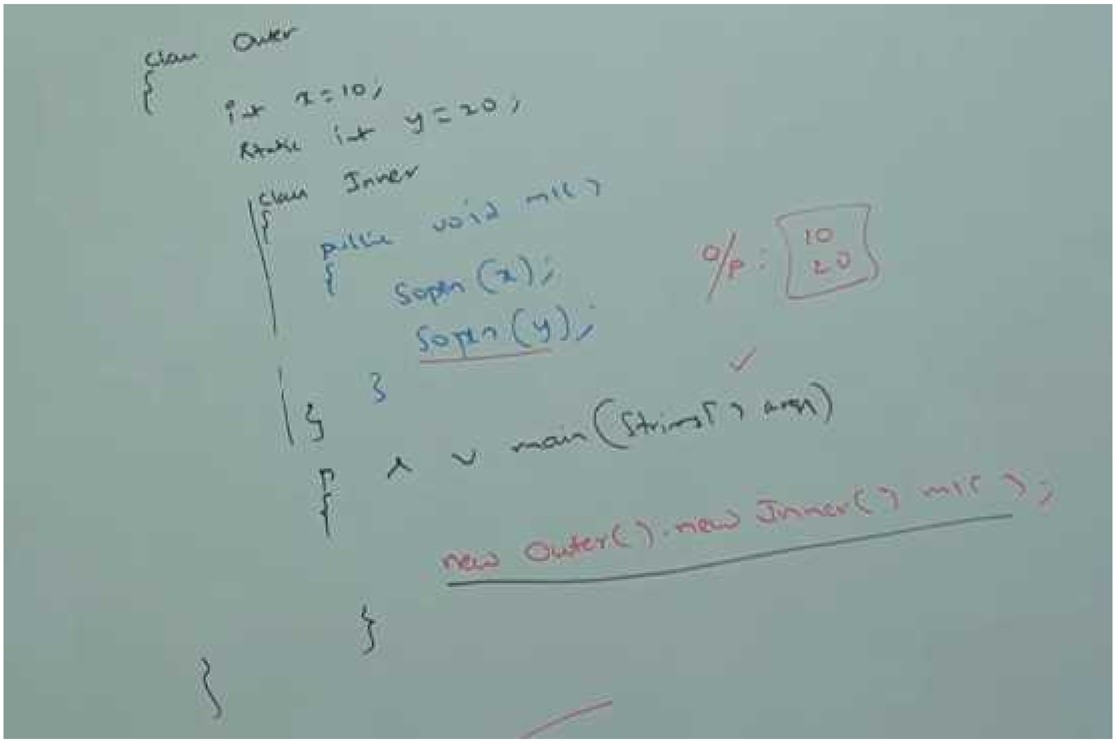


accessing Inner class code
From static Area of outer class
(or)
From outside of outer class
From Instance Area of
outer class

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
i.m1();
```

```
Inner i = new Inner();
i.m1();
```

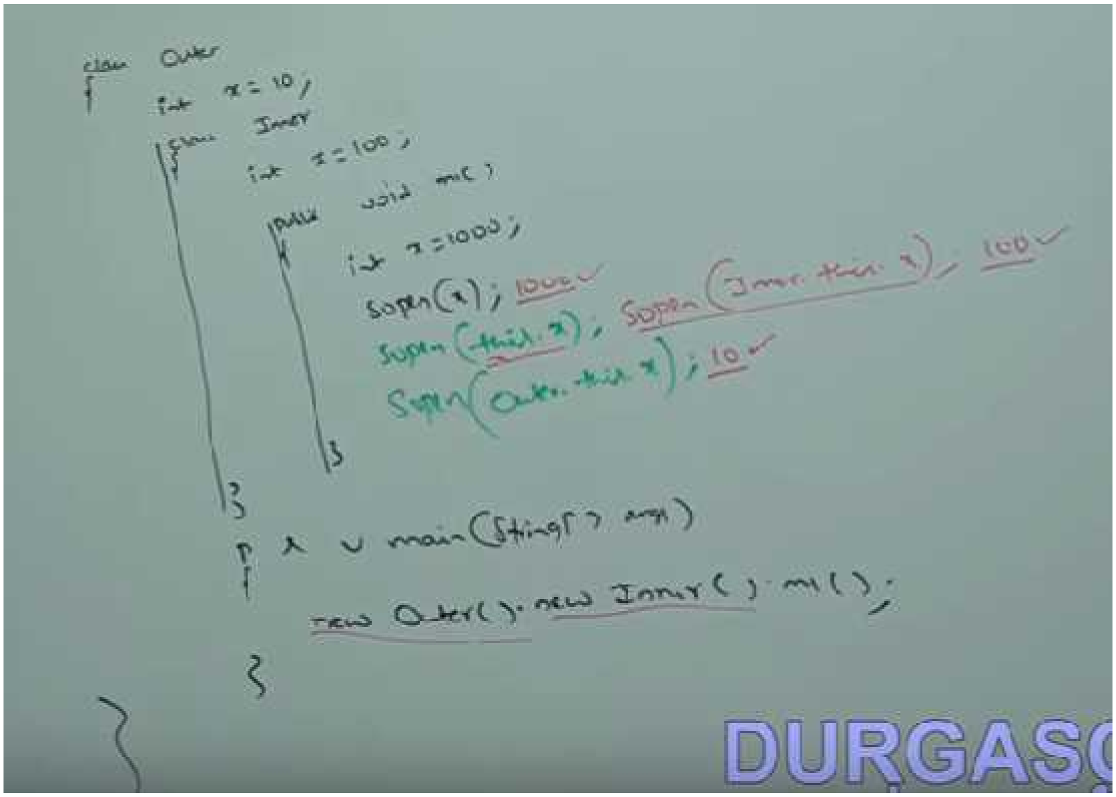
From normal or regular inner class we can access both static and non static members of outer class directly



```
class Outer
{
    int x=10;
    static int y=20;
     class Inner
     {
         public void m1()
         {
             Sopln(x);
             Sopln(y);
         }
     }
     p s v main(String[] args)
     {
         new Outer().new Inner().m1();
     }
}
```

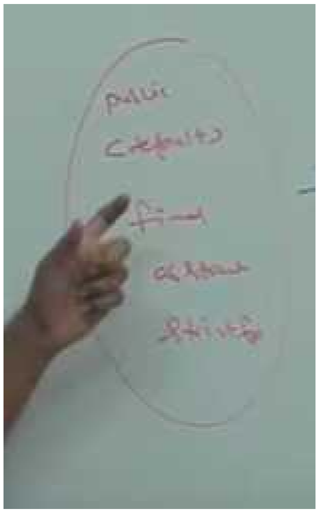
o/p : 10
      20

within the inner class 'this' always refers current inner class object. If we want to refer current outer class object we have to use 'outerclassName.this'.



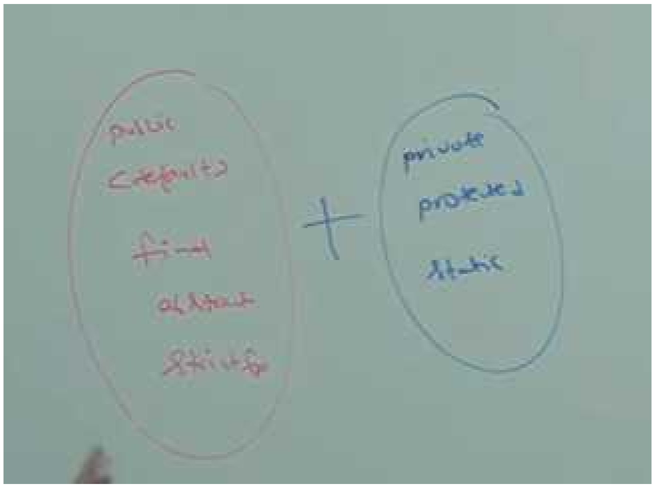
```
class Outer
{
    int x=10;
    class Inner
    {
        int x=100;
        public void m1()
        {
            int x=1000;
            Sopln(x); 1000✓
            Sopln(this.x); Sopln(Inner.this.x); 100✓
            Sopln(Outer.this.x); 10✓
        }
    }
    p s v main(String[] args)
    {
        new Outer().new Inner().m1();
    }
}
```

The only applicable modifiers for outer classes are,



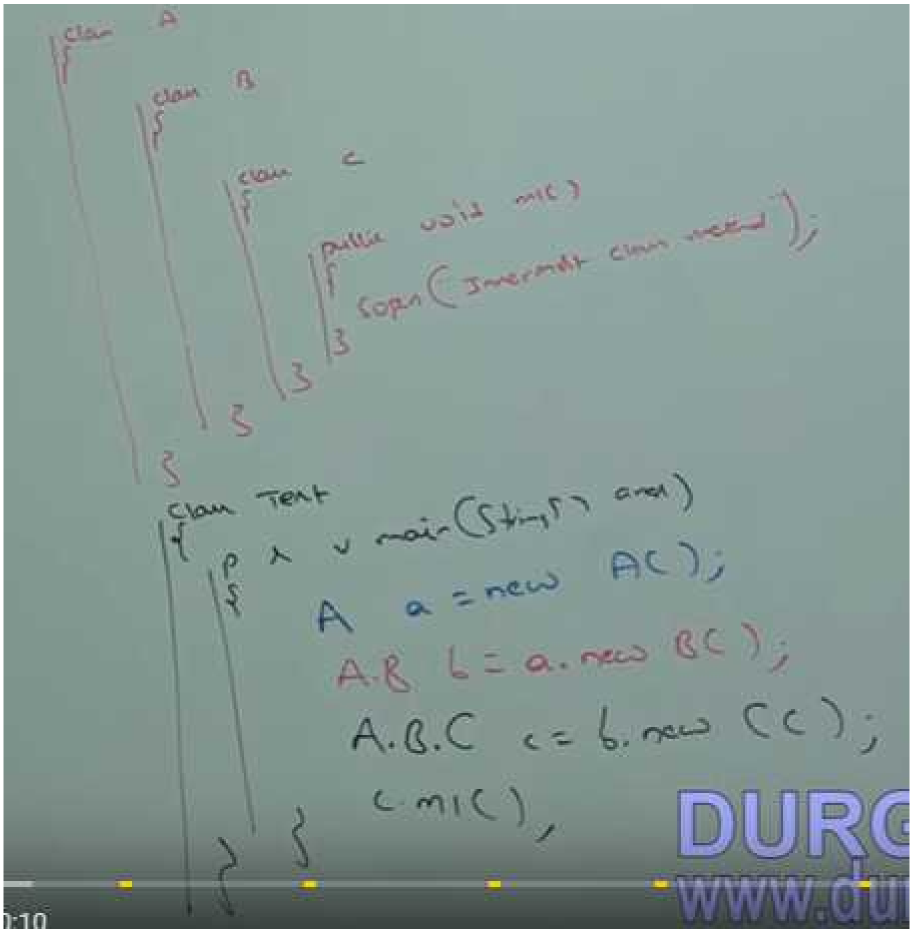
public
<default>
final
abstract
strictfp

But for inner classes applicable modifiers are,



Nesting of Inner classes:

Inside inner class we can declare another inner class i.e. nesting of inner classes is possible.



```
class A
{
    class B
    {
        class C
        {
            public void m1()
            {
                sopn("Innermost class method");
            }
        }
    }
}
class Test
{
    p s v main(String[] args)
    {
        A a = new A();
        A.B b = a.new B();
        A.B.C c = b.new C();
        c.m1();
    }
}
```

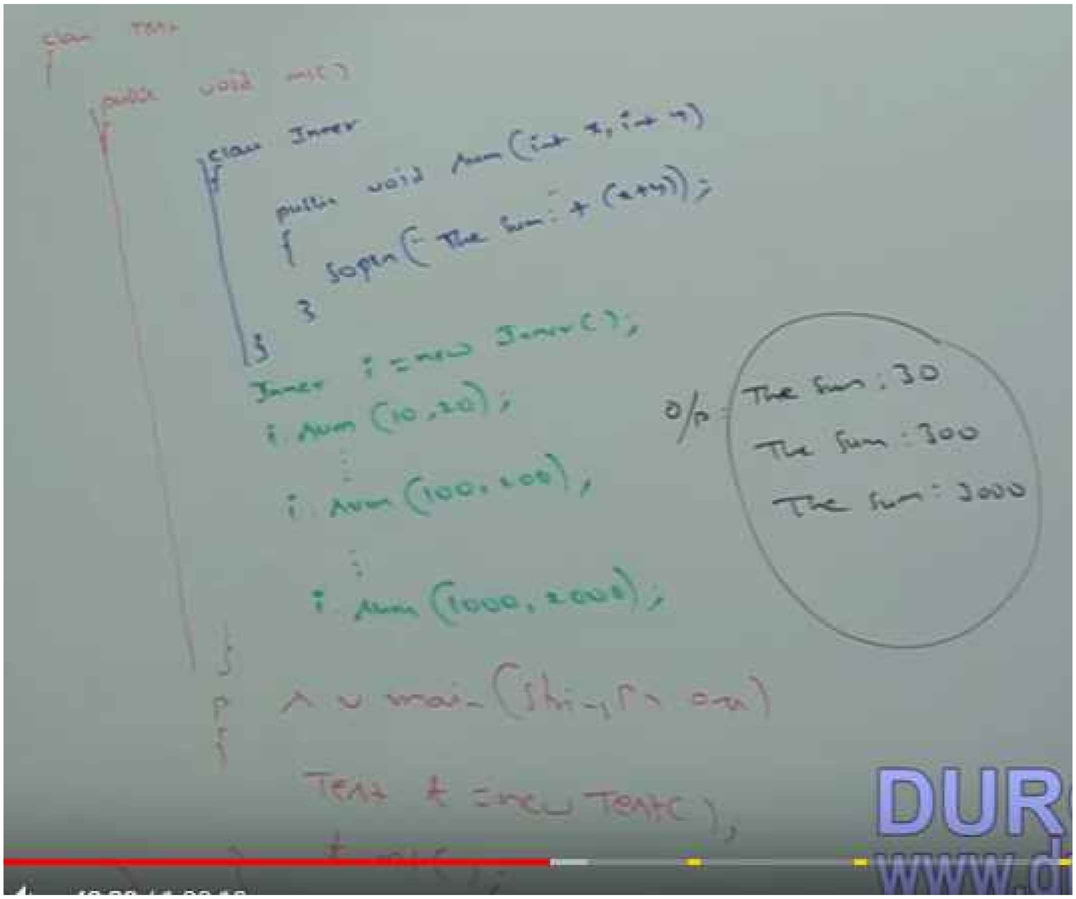
Method Local Inner classes:

Sometimes we can declare a Class inside a method. Such type of inner classes are called method local inner classes.

The main purpose of method local inner class is to define method specific repeatedly required functionality.

Method local inner classes are best suitable to meet nested method requirements.

We can method local inner classes only within the method where we declared, outside of the method we can't acces. Because of it's less scope method local inner classes are most rarely used type of inner classes.



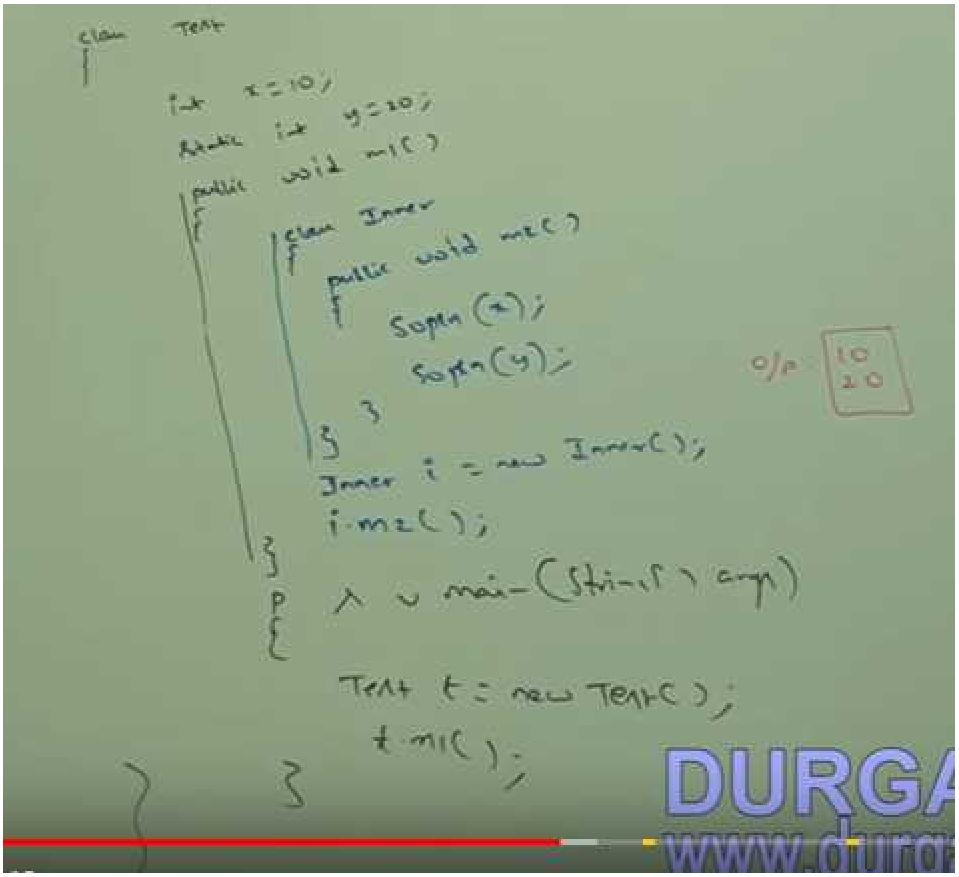
```
class Test
{
  public void m1()
  {
    class Inner
    {
      public void sum(int x, int y)
      {
        sopln("The sum:"+(x+y));
      }
    }
    Inner i=new Inner();
    i.sum(10,20);
      :
    i.sum(100,200);
      :
    i.sum(1000,2000);
  }
  p s v main(String[] args)
  {
    Test t=new Test();
    t.m1();
  }
}
```

```
o/p: The sum:30
     The sum:300
     The sum:3000
```

We can declare method local inner class inside both instance and static methods.

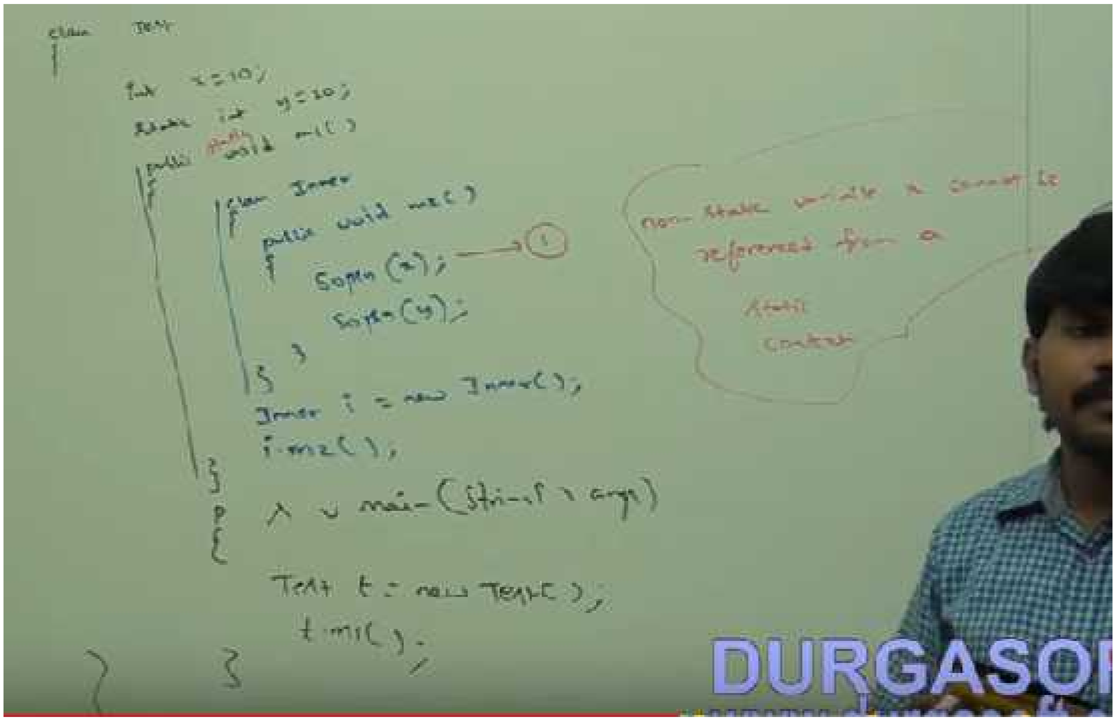
If we declare inner class inside instance method then form that method local inner class we can access both static and non-static members of outer class directly.

If we declare inner class inside a static method then we can access only static members of outer class directly from that method local inner class



```
class Test
{
    int x=10;
    static int y=20;
    public void m1()
    {
        class Inner
        {
            public void m2()
            {
                Sopln(x);
                Sopln(y);
            }
        }
        Inner i = new Inner();
        i.m2();
    }
    p s v main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}
```

o/p 10
20



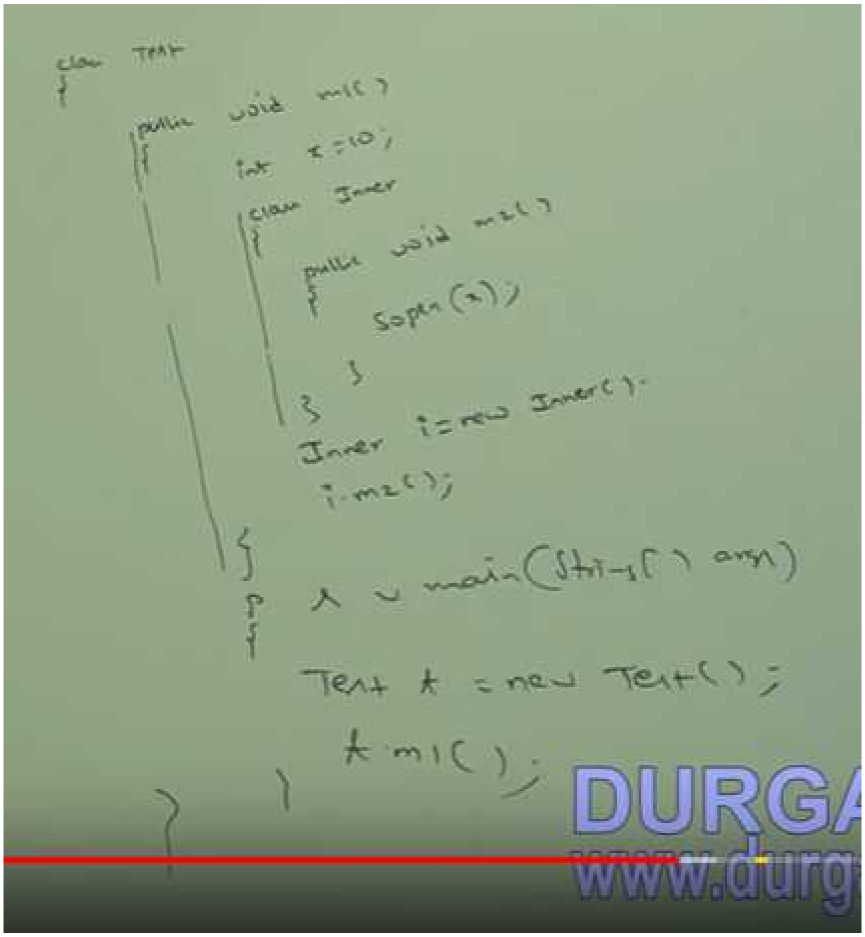
```
class Test
{
    int x=10;
    static int y=20;
    public static void m1()
    {
        class Inner
        {
            public void m2()
            {
                Sopln(x); ——→ ①
                Sopln(y);
            }
        }
        Inner i = new Inner();
        i.m2();
    }
    p s v main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}
```

non-static variable x cannot be referenced from a static context

**From method local inner class we can't acces local variabls of the method in which we declare inner class

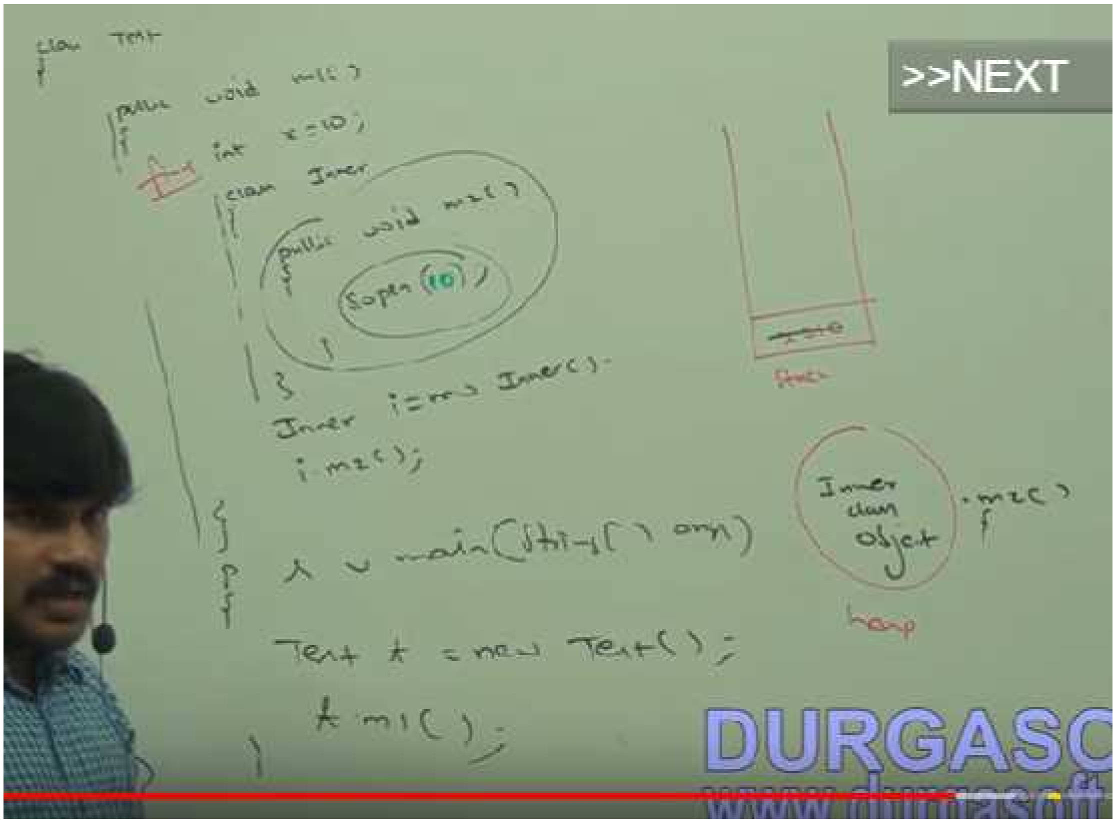
If the local variable declared as 'final' then we can access.

following code gives compile time error so make 'x' final.



```
class Test
{
    public void m1()
    {
        int x=10;
        class Inner
        {
            public void m2()
            {
                Sopln(x);
            }
        }
        Inner i=new Inner();
        i.m2();
    }
    p s v main(String[] args)
    {
        Test t = new Test();
        t.m1();
    }
}
```

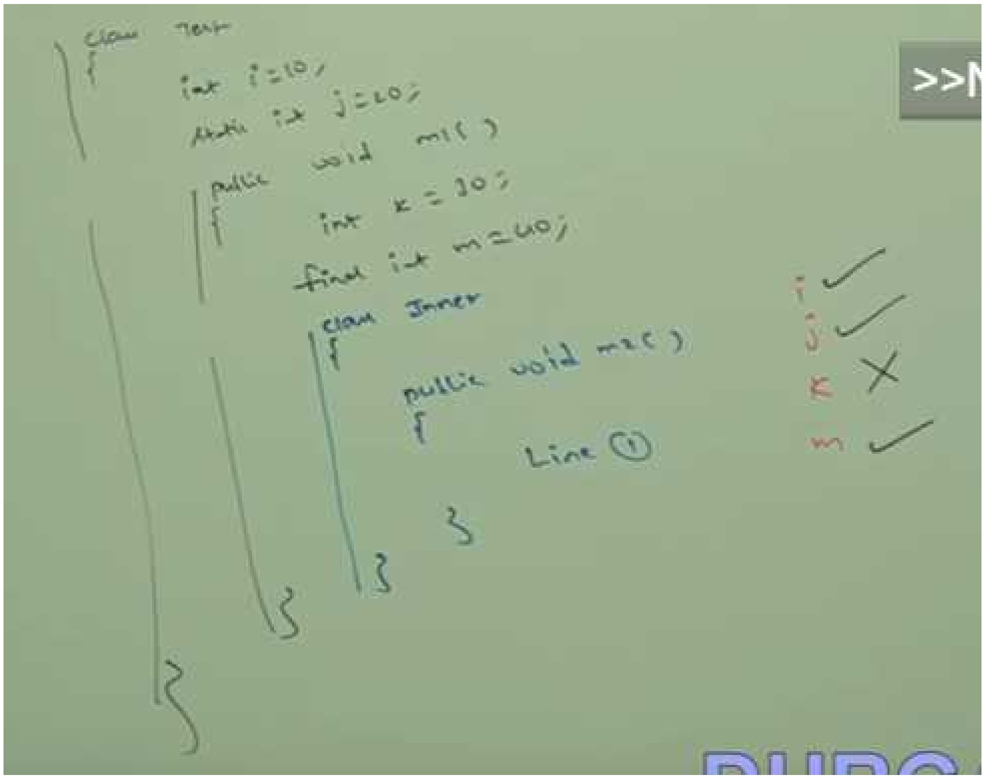
final variable values are replaced at compile time only, hence following code compiles



```
class Test
{
    public void m1()
    {
        int x=10;
        class Inner
        {
            public void m2()
            {
                Sopln(10);
            }
        }
        Inner i=new Inner();
        i.m2();
    }
    p s v main(String[] args)
    {
        Test t =new Test();
        t.m1();
    }
```

Q.1

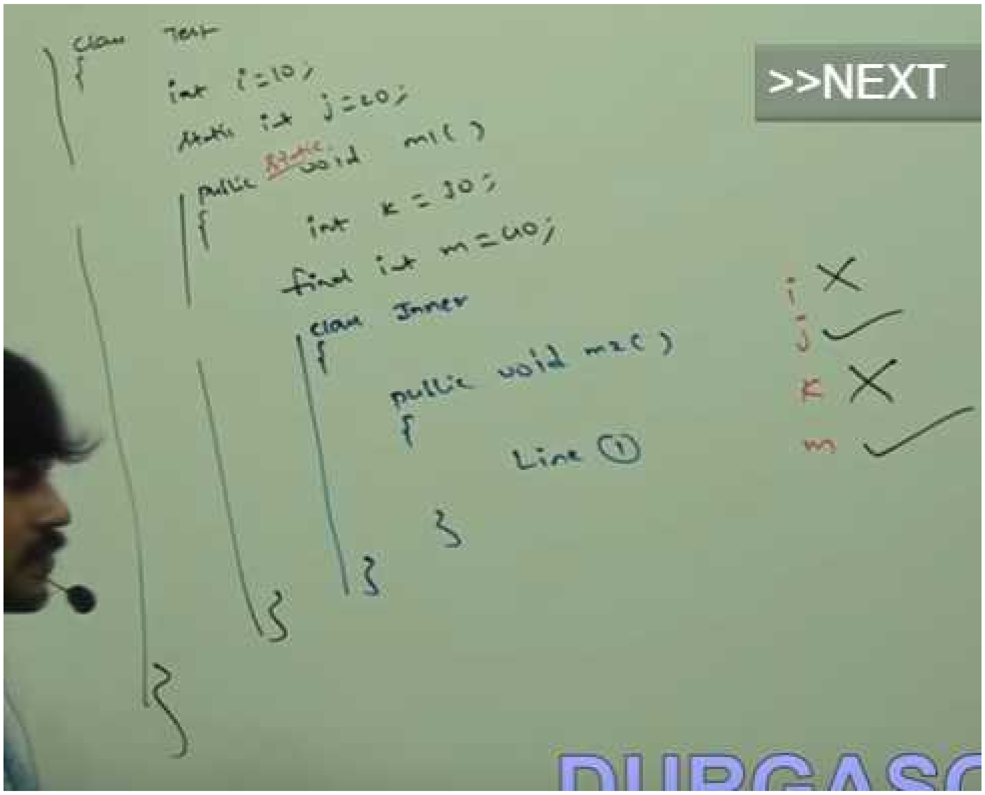
Consider the following code,



```
class Test
{
    int i=10;
    static int j=20;
    public void m1()
    {
        int k=30;
        final int m=40;
        class Inner
        {
            public void m2()
            {
                Line ①
            }
        }
    }
}
```

i ✓
j ✓
k ✗
m ✓

Q.2



```
class Test
{
    int i=10;
    static int j=20;
    public static void m1()
    {
        int k=30;
        final int m=40;
        class Inner
        {
            public void m2()
            {
                Line ①
            }
        }
    }
}
```

i ✗
j ✓
k ✗
m ✓

If we declare m1() as static then at line 1 which variables we can access directly,(see above)

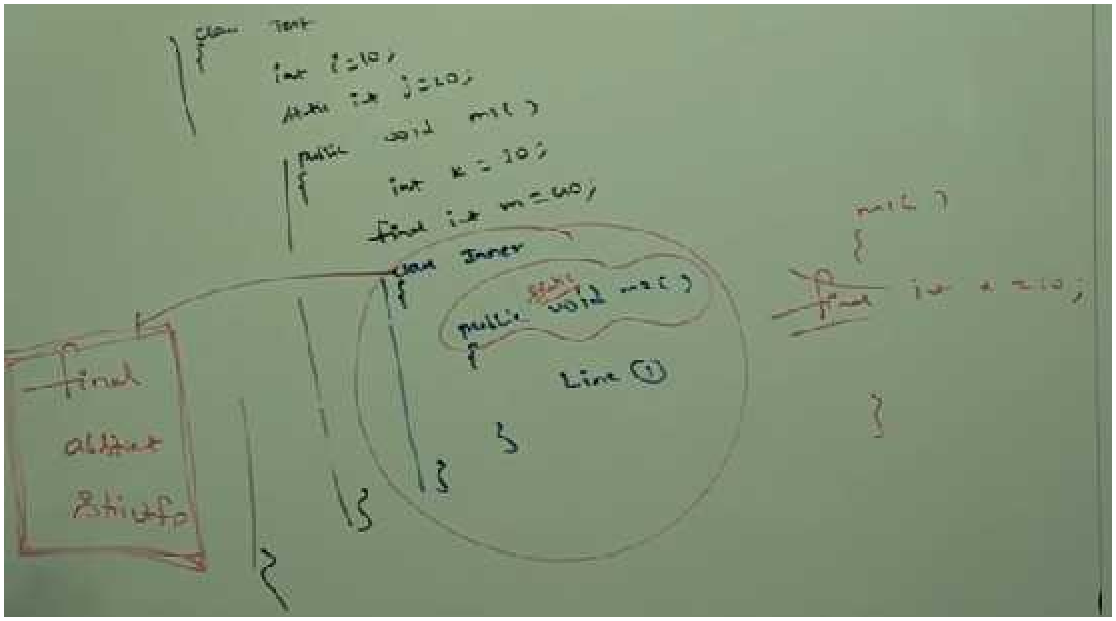
Q. 3

If we declare m2(), method as static then at line 1 which variables we can access directly ?

Ans: we will get compile time error because we can't declare static members inside inner classes.

Note:

The only appicable modifiers for method local inner classes are



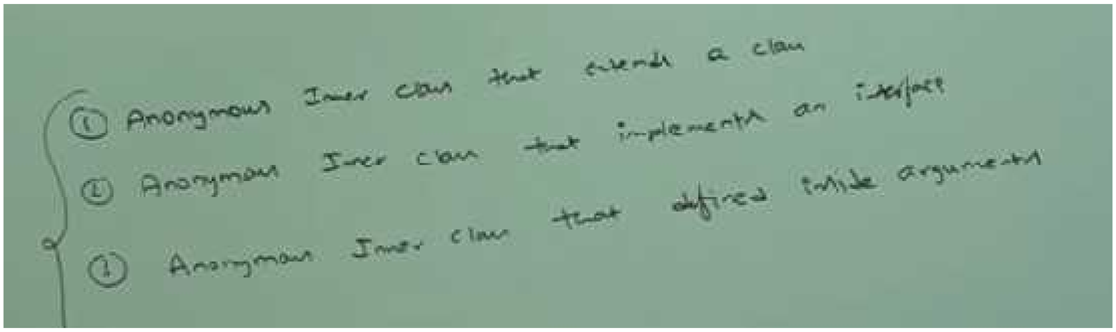
If we are trying apply any other modifier then we will get compile time error.

Annonymous Inner classes:

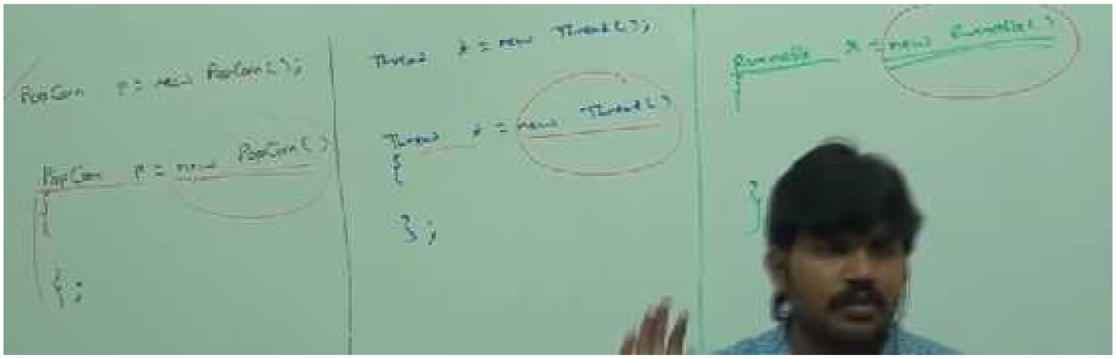
Sometimes we can declare inner class without name.Such type of inner classes are called annonymous inner classes.

The main purpose of annonymous inner classes is just for instant use(one time uses).

Based on declaration and behaviour there are three types of annonymous inner classes,

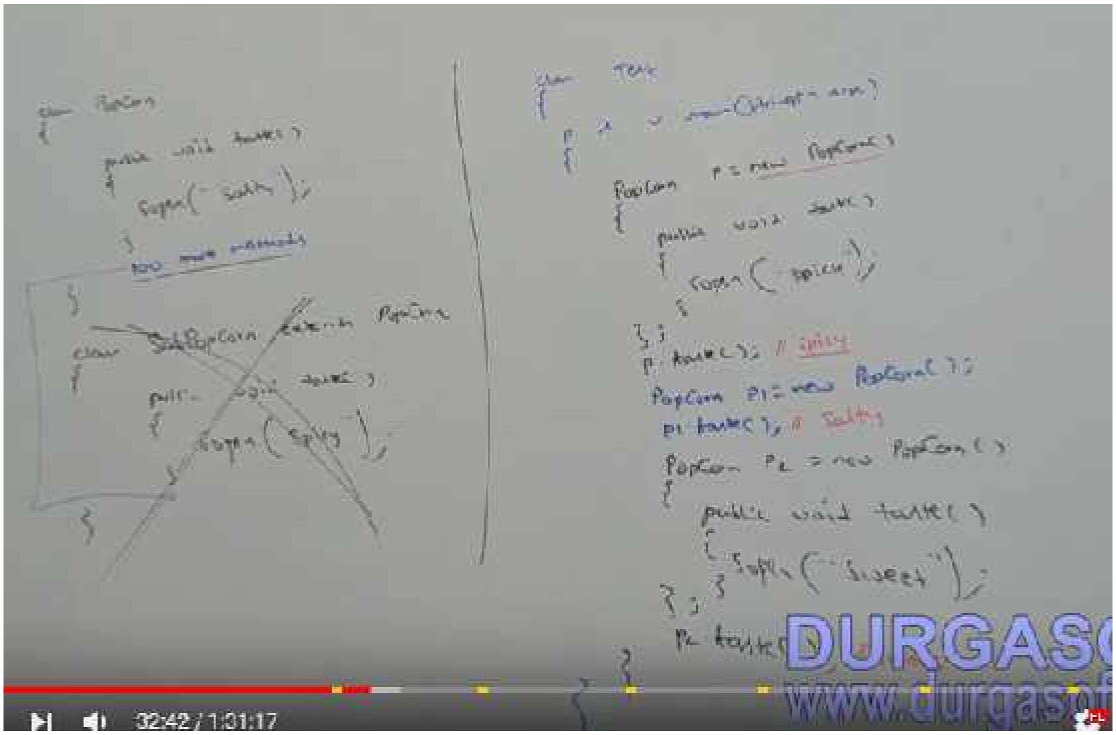


1. Anonymous Inner class that extends a class
2. Anonymous Inner class that implements an interface
3. Anonymous Inner class that defined inside arguments



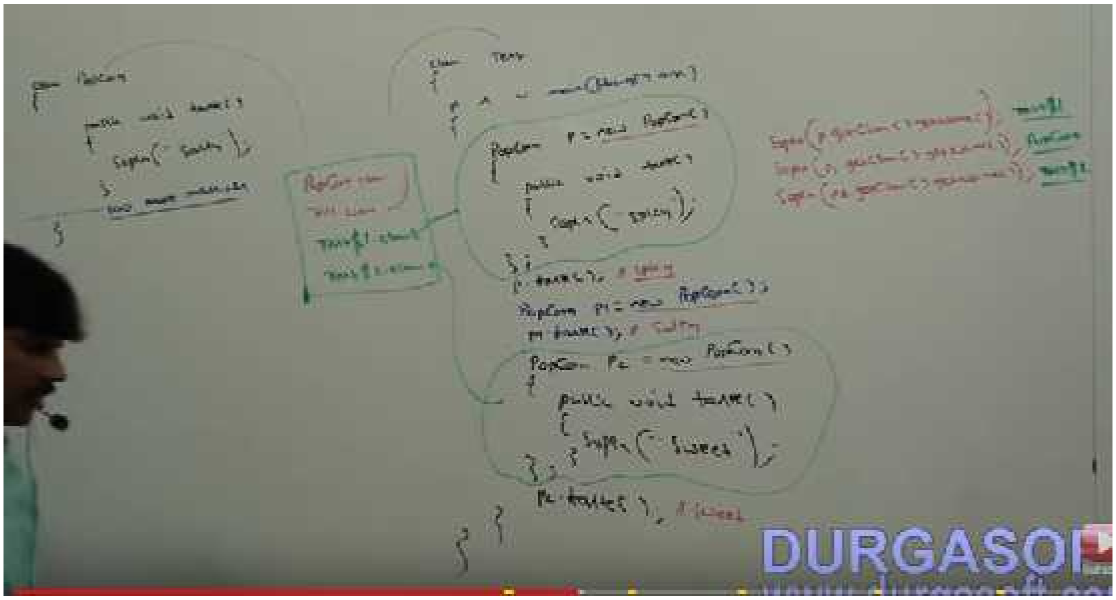
```
PopCorn p = new PopCorn();

PopCorn p = new PopCorn()
{
};

Thread t = new Thread();

Thread t = new Thread()
{
};

Runnable r = new Runnable()
{
}
```

Annonymous Inner class that Extends a class:



```
class PopCorn
{
    public void taste()
    {
        Sopln("salty");
    }
    100 more methods
}
~~class SubPopCorn extends PopCorn~~
~~{~~
    ~~public void taste()~~
    ~~{~~
        ~~Sopln("spicy");~~
    ~~}~~
~~}~~
```

```
class Test
{
    p s v main(String[] args)
    {
        PopCorn p = new PopCorn()
        {
            public void taste()
            {
                Sopln("spicy");
            }
        };
        p.taste(); // spicy
        PopCorn p1 = new PopCorn();
        p1.taste(); // salty
        PopCorn p2 = new PopCorn()
        {
            public void taste()
            {
                Sopln("sweet");
            }
        };
        p2.taste();
    }
}
```



Analysis:

1. PopCorn p = new PopCorn();

just we are creating PopCorn object.

2. PopCorn p = new PopCorn(){};

We are declaring a class that extnds PopCorn without name(Annonymous inner class)

For that child class we are creating an object with parent reference.

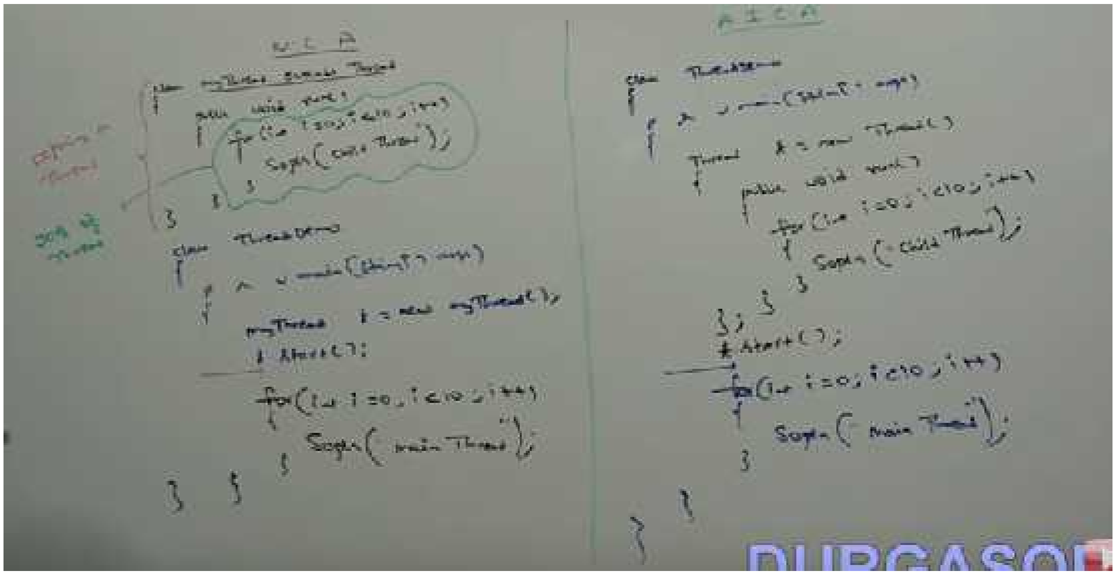
```
3. PopCorn p = new PopCorn(){
    public void taste(){
      SOP("spiecy");
    }
}
```

We are declaring a class that extens PopCorn without name(Annonymous inner class)

In that child class we are overriding taste() method

For that child class we are creating an object with parent reference.

Defining a Thread by Extending Thread class:

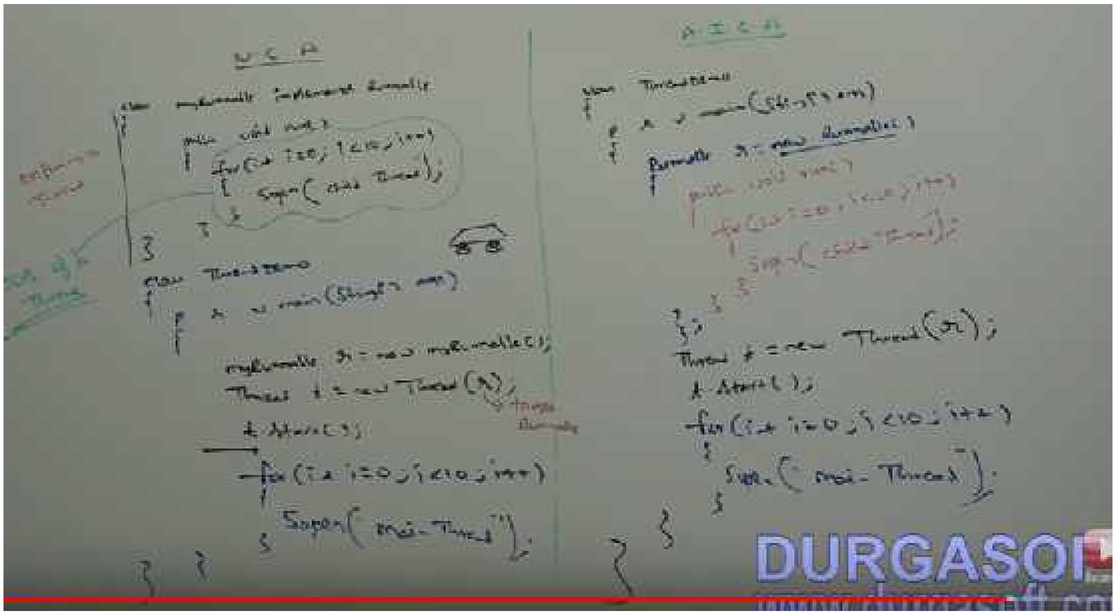


```
class myThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            Sopln("child Thread");
        }
    }
}
class ThreadDemo
{
    p s v main(String[] args)
    {
        myThread t = new myThread();
        t.start();
        for(int i=0;i<10;i++)
        {
            Sopln("main Thread");
        }
    }
}
```

```
class ThreadDemo
{
    p s v main(String[] args)
    {
        Thread t = new Thread()
        {
            public void run()
            {
                for(int i=0;i<10;i++)
                {
                    Sopln("Child Thread");
                }
            }
        };
        t.start();
        for(int i=0;i<10;i++)
        {
            Sopln("main Thread");
        }
    }
}
```

Annonymous Inner class that implements an Interface:

Defining a thread by implementing Runnable Interface,



N.C.A

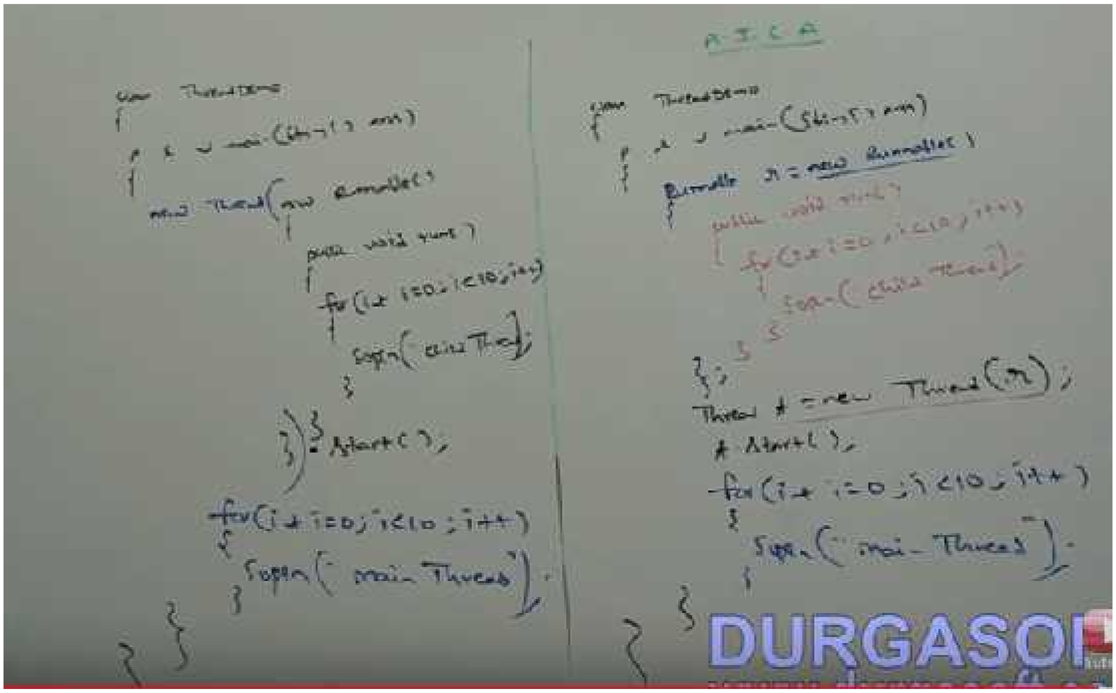
```
class MyRunnable implements Runnable
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      Sopln("child Thread");
    }
  }
}
class ThreadDemo
{
  p s v main(String[] args)
  {
    MyRunnable r = new MyRunnable();
    Thread t = new Thread(r);
    t.start();
    for(int i=0; i<10; i++)
    {
      Sopln("main Thread");
    }
  }
}
```

defining a thread
job of a Thread
target Runnable

A.I.C.A

```
class ThreadDemo
{
  p s v main(String[] args)
  {
    Runnable r = new Runnable()
    {
      public void run()
      {
        for(int i=0; i<10; i++)
        {
          Sopln("child Thread");
        }
      }
    };
    Thread t = new Thread(r);
    t.start();
    for(int i=0; i<10; i++)
    {
      Sopln("main Thread");
    }
  }
}
```

Annonymous Inner Class that defines inside Argument:



```
class ThreadDemo
{
  p s v main(String[] args)
  {
    new Thread(new Runnable()
    {
      public void run()
      {
        for(int i=0; i<10; i++)
        {
          Sopln("child Thread");
        }
      }
    }).start();
    for(int i=0; i<10; i++)
    {
      Sopln("main Thread");
    }
  }
}
```

A.I.C.A

```
class ThreadDemo
{
  p s v main(String[] args)
  {
    Runnable r = new Runnable()
    {
      public void run()
      {
        for(int i=0; i<10; i++)
        {
          Sopln("child Thread");
        }
      }
    };
    Thread t = new Thread(r);
    t.start();
    for(int i=0; i<10; i++)
    {
      Sopln("main Thread");
    }
  }
}
```

Normal Java class vs Annonymous Inner class:

1. A normal java class can extend only one class at a time of course annonymous inner class also can extend only one class at a time.

2. A normal java class can implement any number of interfaces simultaneously but annonymous inner class can implement only one interface at a time.

3. A normal java class can extend another class and can implement any number of inetfaces simultaneously.

But annonyous inner class can extend a class or can implement an interface but not both simultaneously.

4. In normal java class we can write any number of constructors simultaneously.

but in annonyous inner classes we can't write any constructor expilicitly(because the name of the class and the name of the constructor must be same but annonyous inner classes not having any name.)

Q. Where Annonyous inner classes are best suitable ?

Ans:

GUI based Event handlers

===

Preventing Thread Executions:

1. yield()

2. join()

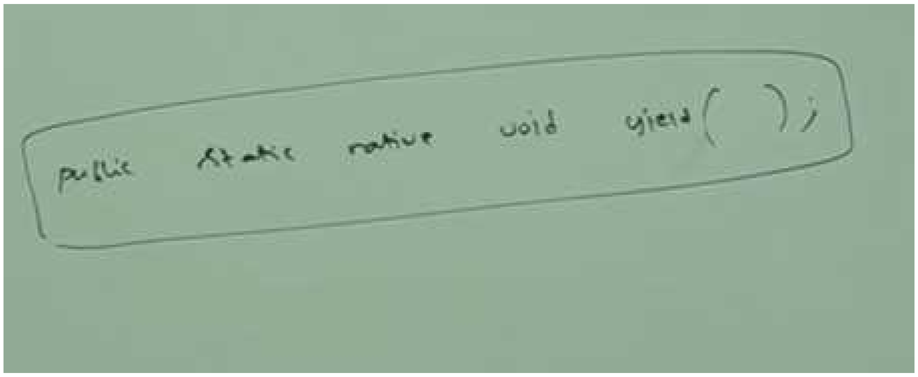
3. sleep()

yield():

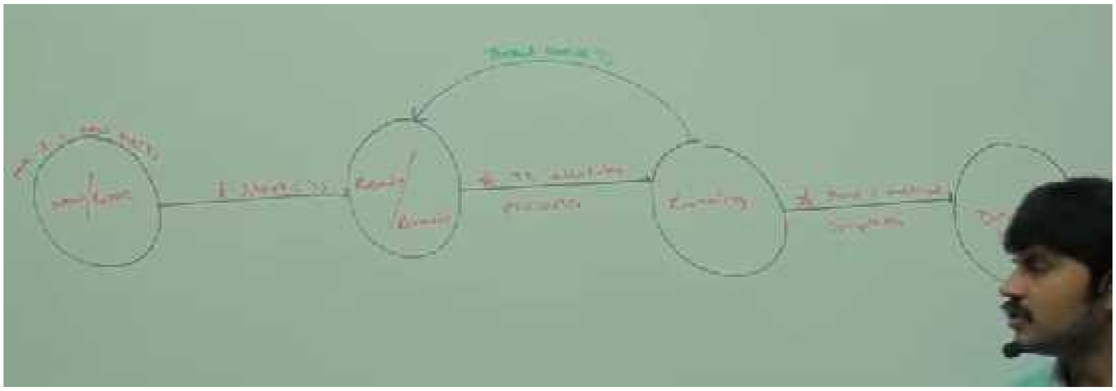
yield() causes to pause current executing thread, to give the chance for waiting threads of same priority.

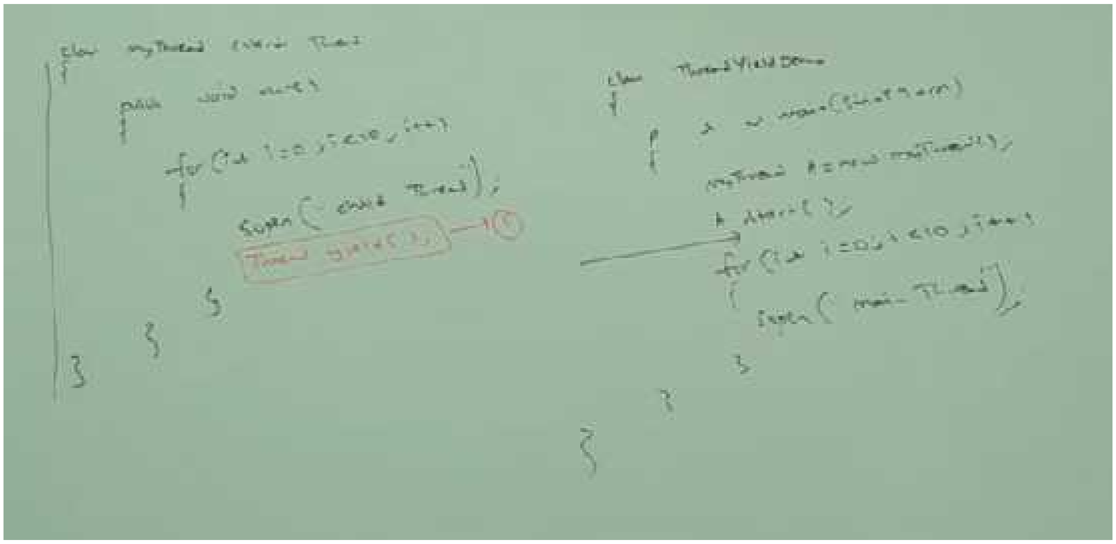
if there is no waithing thread or all waithing threads have low prority then same thread can continue it's execution.

The thread which is yielded, when it will get the chance once again it depends on thread scheduler.



```
public static native void yield( );
```





```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            Sopln("child Thread");
            Thread.yield(); —1
        }
    }
}

class ThreadYieldDemo
{
    p s v main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        for(int i=0;i<10;i++)
        {
            Sopln("main Thread");
        }
    }
}
```

Child thread always called yield method because of that main will get chance more number of times and the chance of completing main thread first is high.

Some platforms won't provide proper support for yield() method.

join():

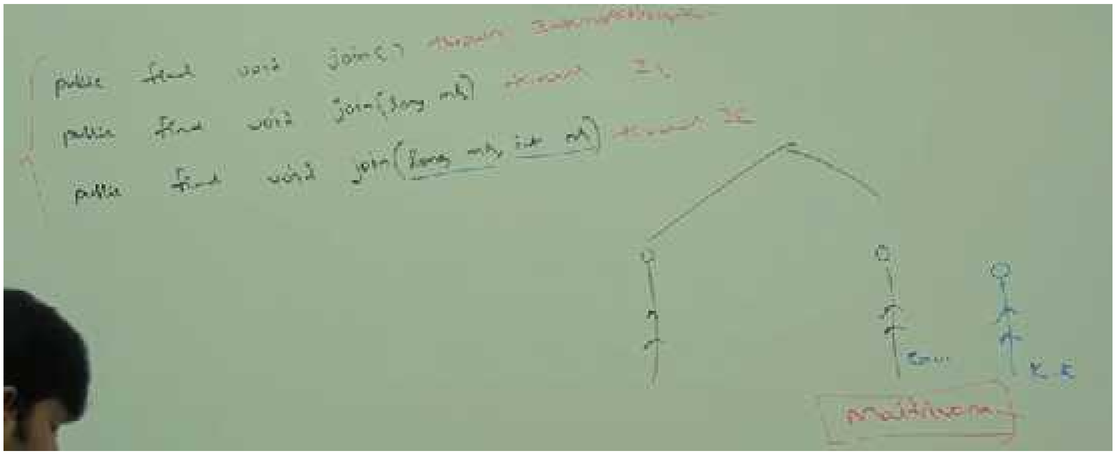
If a thread wants to wait until completeing some other thread then we should go for join(), method.

e.g.

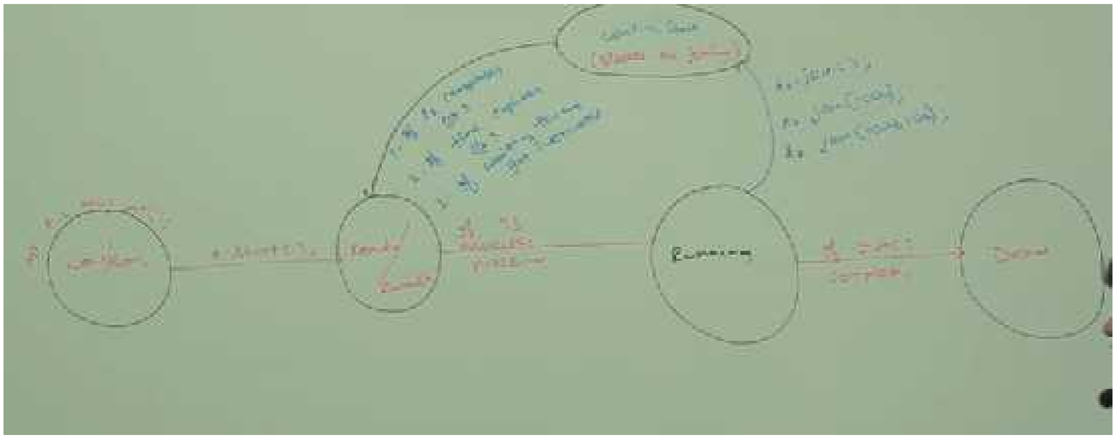
If a thread t1 wants to wait until completing t2, then t1 has to call t2.join().

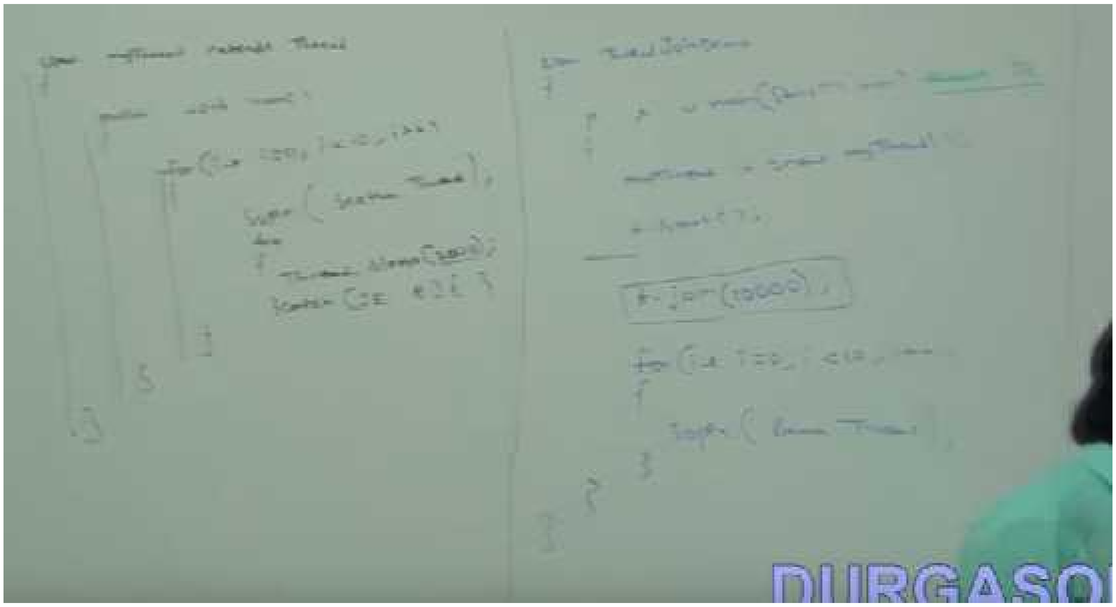
If t1 executes t2.join(), then immediately t1, will be entered into waiting state until t2 completes

Once t2 completes then t1 can continue it's execution.

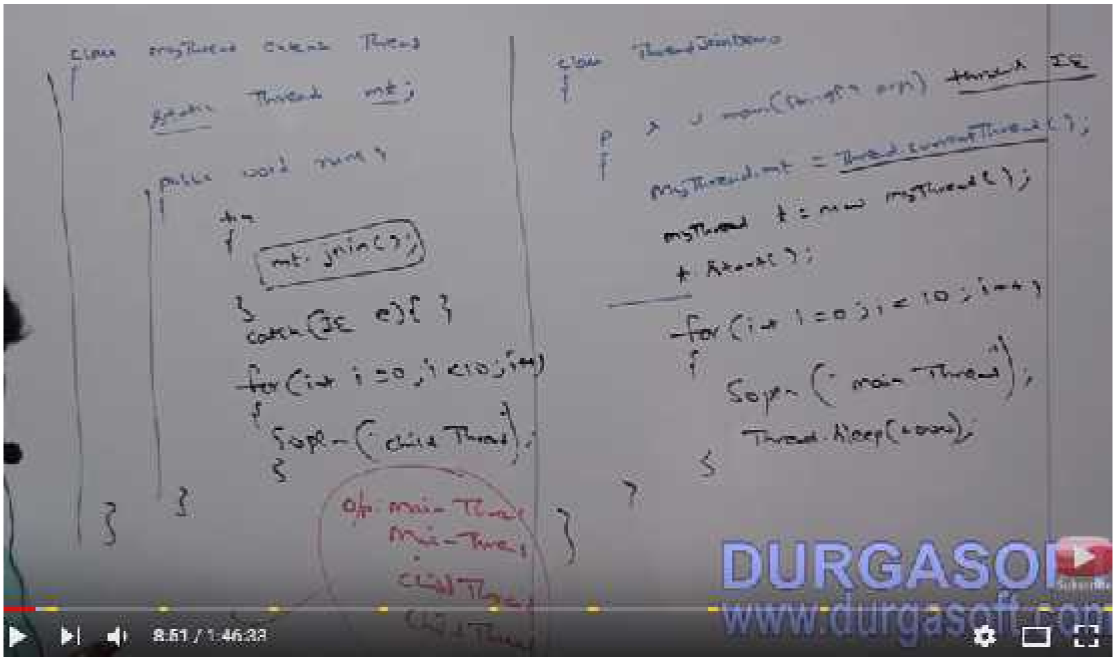


```
public final void join() throws InterruptedException
public final void join(long ms) throws IE
public final void join(long ms, int ns) throws IE
```





waiting of child thread until completing main() thread,



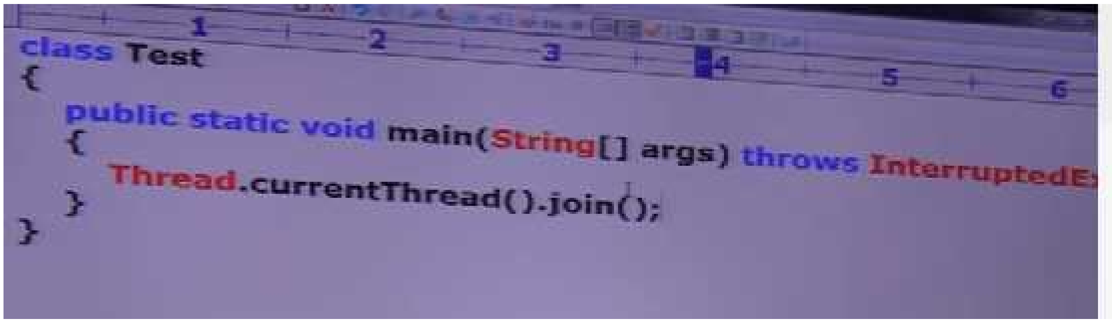
```
class myThread extends Thread
{
  static Thread mt;
  public void run()
  {
    try
    {
      mt.join();
    }
    catch(IE e){ }
    for(int i=0;i<10;i++)
    {
      Sopln("child Thread");
    }
  }
}

class ThreadJoinDemo
{
  p s v main(String[] args) throws IE
  {
    myThread.mt = Thread.currentThread();
    myThread t = new myThread();
    t.start();
    for(int i=0;i<10;i++)
    {
      Sopln("main Thread");
      Thread.sleep(2000);
    }
  }
}
```

Case 3:

If main(), thread calls join method on child thread object and child thread called join method on main thread object then both threads will wait forever and the program will be stucked(this is something like deadlock).

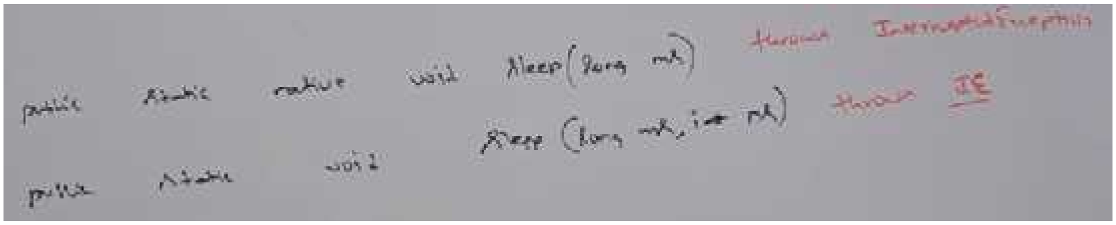
Case 4:

Another deadlock situation, program stucks



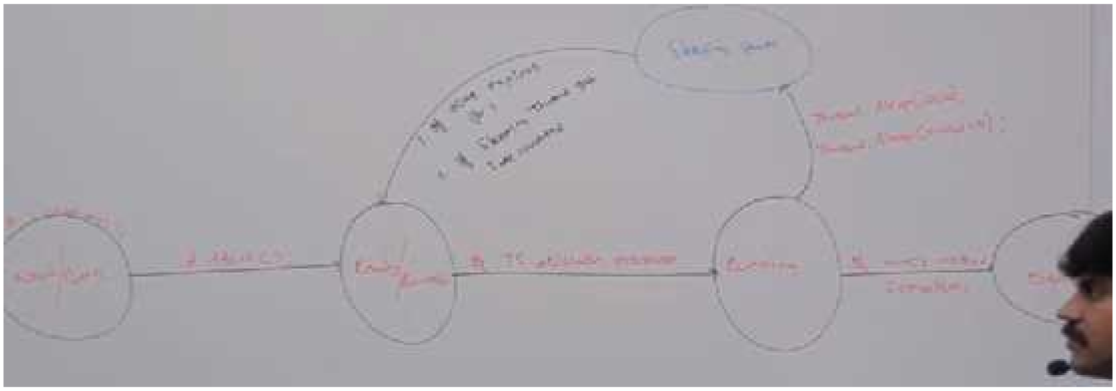
```
class Test
{
    public static void main(String[] args) throws InterruptedE
    {
        Thread.currentThread().join();
    }
}
```

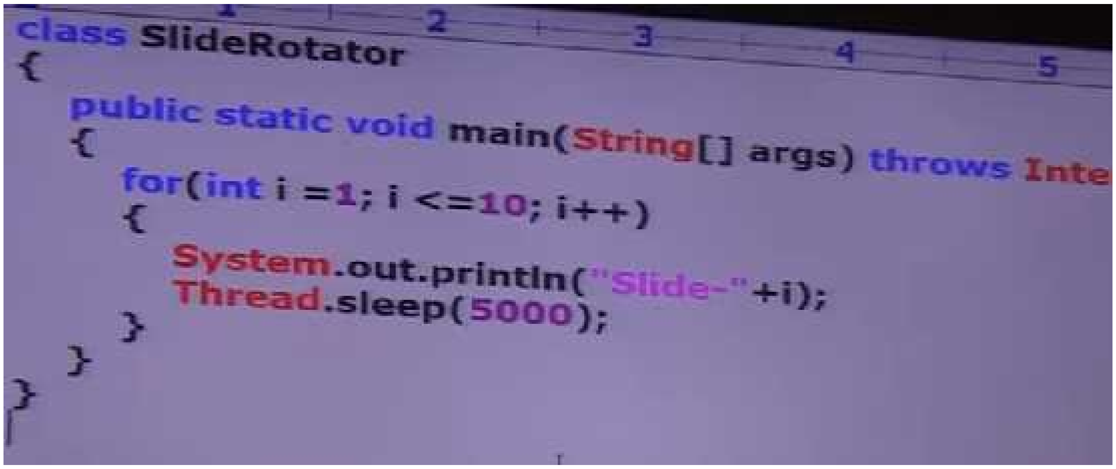
Sleep():



public static native void sleep(long ms) throws InterruptedException

public static void sleep(long ms, int ns) throws IE

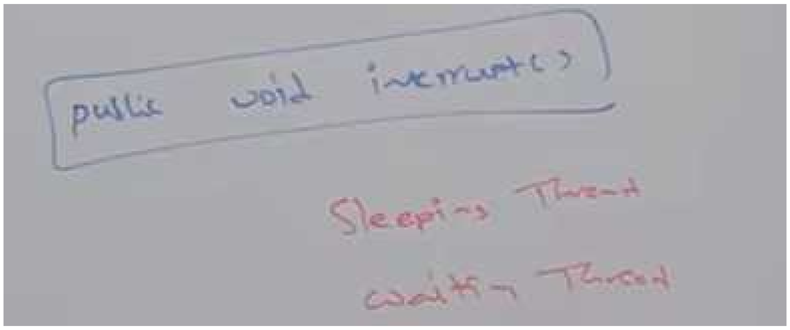




```
class SlideRotator
{
   public static void main(String[] args) throws Inte
   {
      for(int i =1; i <=10; i++)
      {
         System.out.println("Slide-"+i);
         Thread.sleep(5000);
      }
   }
}
```

Thread Interruption:

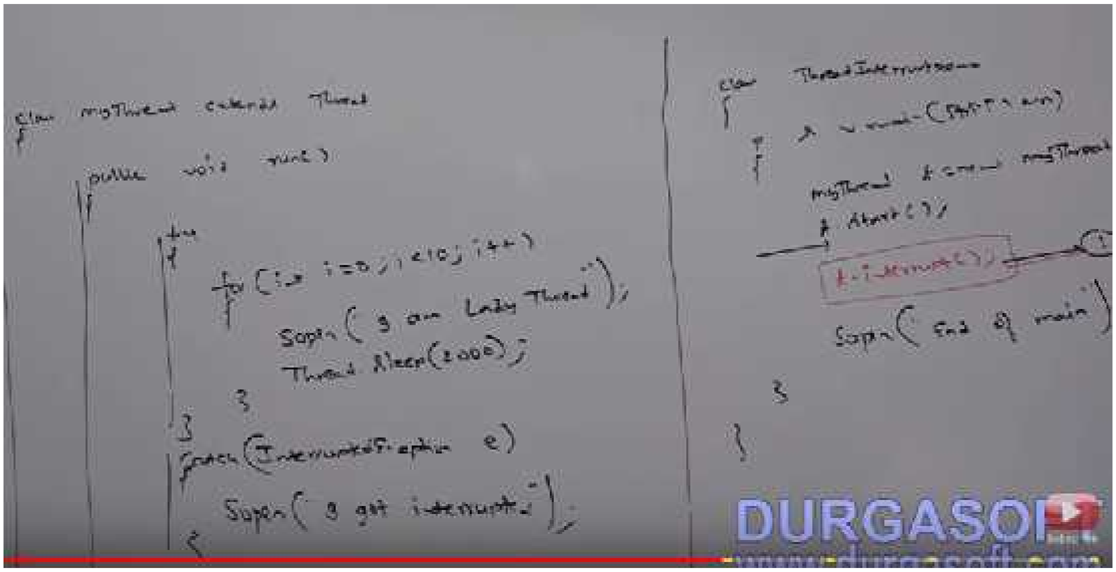
A thread can interrupt a sleeping thread or waiting thread by using interrupt() method of thread class.



public void interrupt()

Sleeping Thread

waiting Thread



```
class myThread extends Thread
{
   public void run()
   {
      try
      {
         for(int i=0; i<10; i++)
         {
            Sopln("I am Lazy Thread");
            Thread.sleep(2000);
         }
      }
      catch(InterruptedException e)
      {
         Sopln("I got interrupted");
      }
```

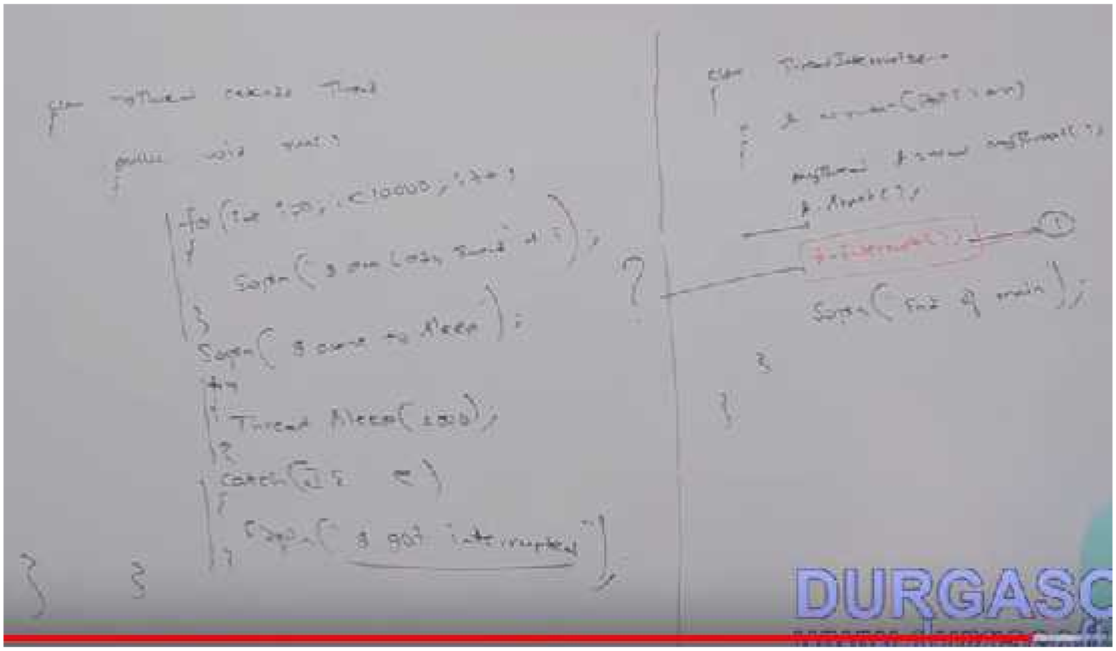
```
class ThreadInterruptDemo
{
   p s v main(String[] args)
   {
      myThread t = new myThread();
      t.start();
      t.interrupt();
      Sopln("End of main");
   }
}
```

*Note:

Whenever we are calling interrupt method, if the target thread not in sleeping state or waiting state then there is no impact of interrupt call immediately. interrupt call will be waited until target thread entered into sleeping or waiting state.

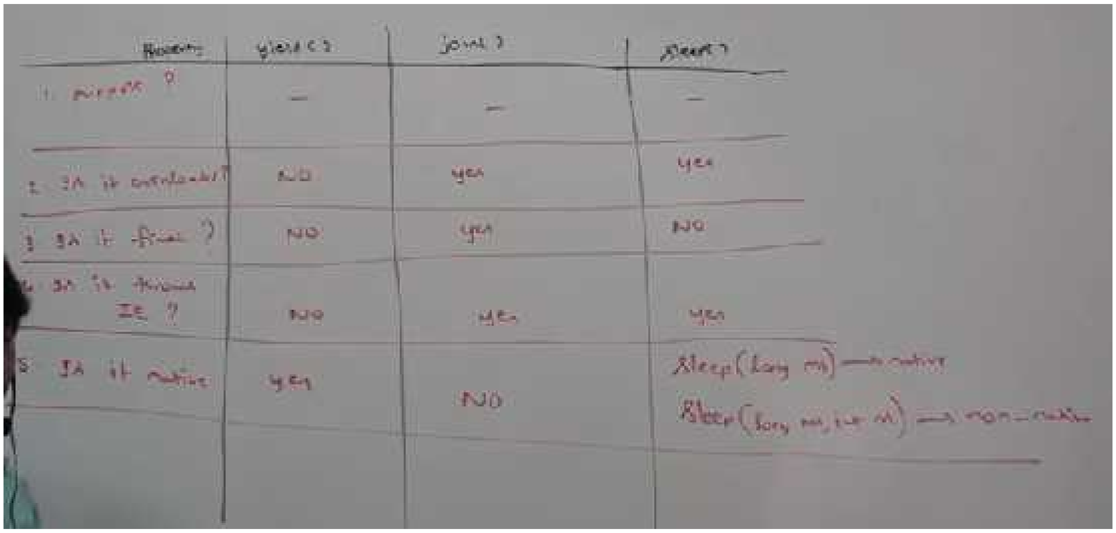
If the target thread entered into sleeping or waiting state then immediately interupt call will interrupt the target thread.

If the target thread never entered into sleeping or waiting state in it's lifetime then there is no impact of interrupt call. This is the only case where interrupt call will be waisted.



```
class MyThread extends Thread
{
   public void run()
   {
      for(int i=0;i<10000;i++)
      {
         Sopln("I am lazy Thread" + i);
      }
      Sopln("I want to sleep");
      try
      {
         Thread.sleep(1000);
      }
      catch(IE e)
      {
         Sopln("I got interrupted");
      }
   }
}

class ThreadInterruptDemo
{
   p s v main(String[] args)
   {
      MyThread t=new MyThread();
      t.start();
      t.interrupt();
      Sopln("End of main");
   }
}
```

Comparision Table:



Property	yield()	join()	sleep()
1. Purpose ?	—	—	—
2. Is it overloaded?	NO	yes	yes
3. Is it final ?	NO	yes	NO
4. Is it throws IE ?	NO	yes	yes
5. Is it native	yes	NO	sleep(long ms) → native sleep(long ms, int ns) → non-native

Syncronization:

Syncronized is a modifier applicable only for methods and blocks but not for classes and variables.

If multiple threads are trying to operate simultaneously on the same java object then there may be a chance of data inconsistancy problem.

To overcome this problem we should go for 'syncronized' keyword.

If a method or block declared as syncronized then at a time only one thread is allowed to execute that method or block on the given object so that data inconsistancy problem will be resolved.

The advantadge of syncronized keyword is we can resolve data inconsistancy problems, but the main disadvantadge of syncronized keyword is it increases waithing time of threads and creats performance problems, hence if there is no specific requirement then it is not recommanded to use syncronized keyword.

Internally syncronization is implemented by using lock. Every object in java has a unique lock.

Whenever we are using syncronized keyword then only lock conept is come into the picture.

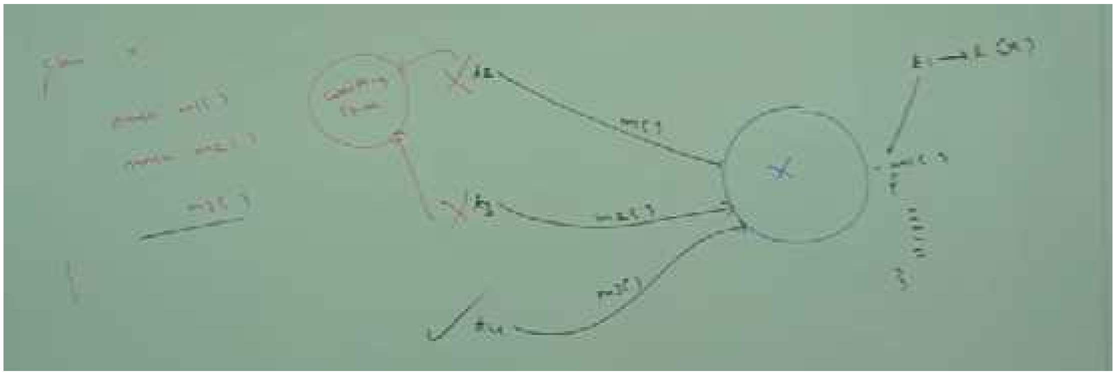
If a thread wants to execute syncronized method on the given object 1st it has to get lock of that object.

Once thread got the lock then it is allowed to execute any syncronized method on that object.

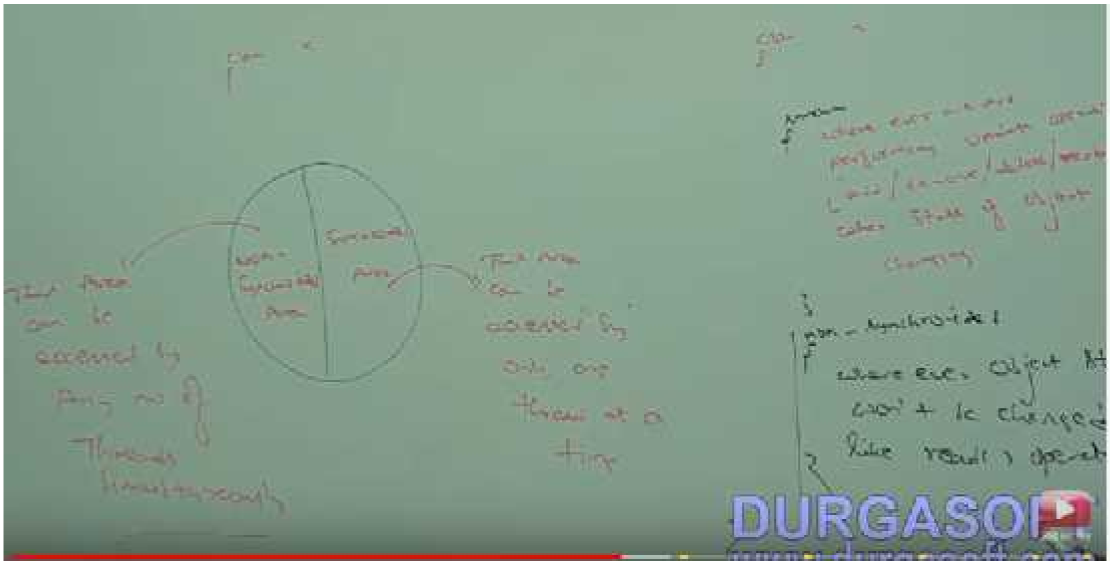
Once method execution completes automaticaly thread releses the lock.

Acquiring and relesing lock internally takes care by JVM, and programmer not responsible for this activity.

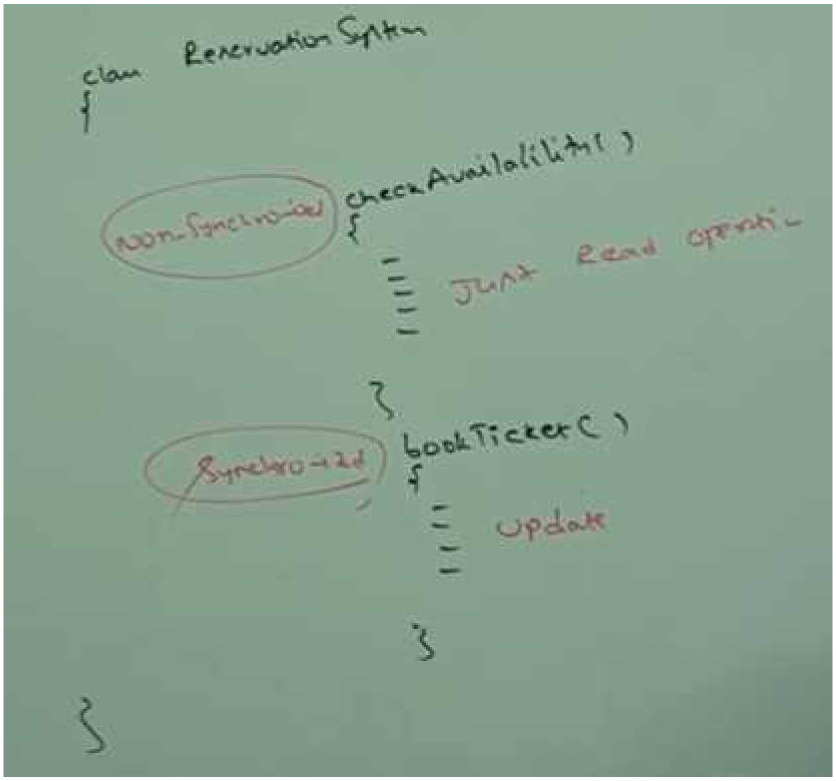
While a thread executing syncronized method on a given object the remaining threads are not allowed to execute any syncronized method simultaneously on the same object, but remaining threads are allowed to execute non syncronized methods simultaneously.



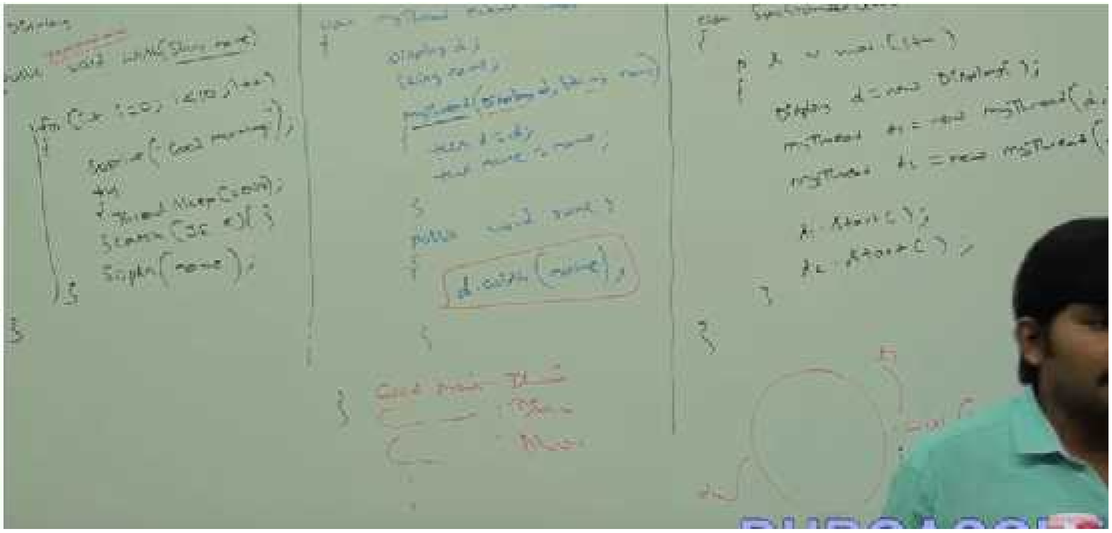
Lock concept is implemented based on Object but not based on method.



Non-Synchronized Area
Synchronized Area
This Area can be accessed by any no of Threads simultaneously
This Area can be accessed by only one thread at a time
where ever we are performing update operation (add/remove/delete/replace) i.e. where state of object changing
non-synchronized
where ever object state won't be changed like read() operation
DURGASOFT



```
class ReservationSystem
{
    Non-Synchronized  checkAvailability()
                      {
                          -
                          -   Just Read operation
                          -
                          -
                      }
    Synchronized      bookTicket()
                      {
                          -
                          -   update
                          -
                      }
}
```

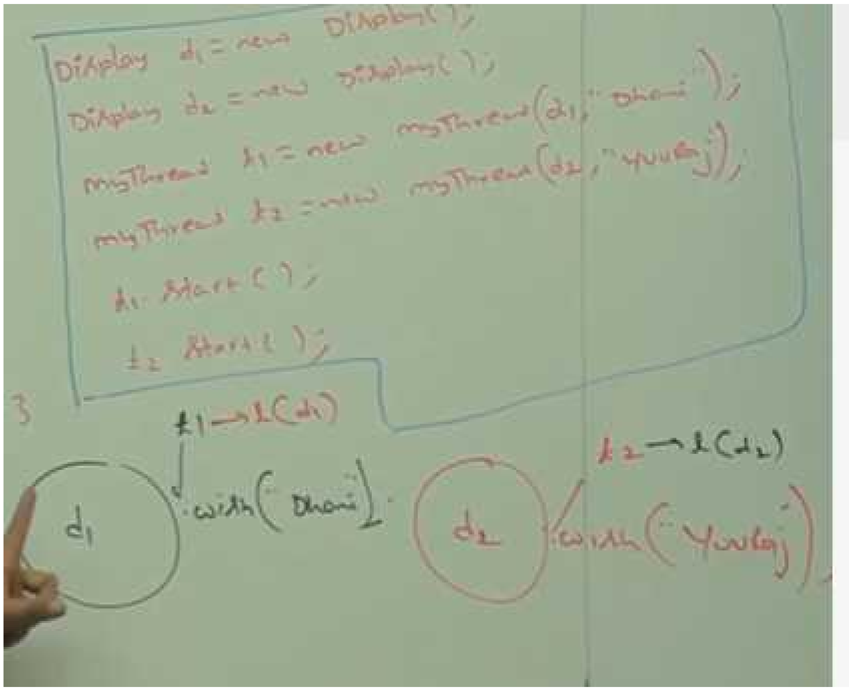


```
Display
{
  public synchronized void wish(String name)
  {
    for(int i=0; i<10; i++)
    {
      Sop("Good morning:");
      try
      { Thread.sleep(2000);
      } catch(IE e){ }
      Sopln(name);
    }
  }
}
```

```
class MyThread extends Thread
{
  Display d;
  String name;
  MyThread(Display d, String name)
  {
    this.d=d;
    this.name=name;
  }
  public void run()
  {
    d.wish(name);
  }
}
```

```
class SynchronizedDemo
{
  p s v main(String[] args)
  {
    Display d=new Display();
    MyThread t1=new MyThread(d,
    MyThread t2=new MyThread(
    t1.start();
    t2.start();
  }
}
```

Case Study:



```
Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new MyThread(d1, "Dhoni");
MyThread t2 = new MyThread(d2, "Yuvraj");
t1.start();
t2.start();
}
```

Even though wish(), method is syncronized we will get irregular output, because threads are operating on different java objects.

Conclusion:

If multiple threads are operating on same java object then syncronization is required.

if multiple threads are operating on multiple java objects then syncronization is not required.

Class level lock:

Every class in java has a unique lock, which is nothing but class level lock.

If a thread wants to execute a static syncronized method then thread required class level lock.

Once thread got class level lock then it is allowed to excecute any static syncrinized method of that class.

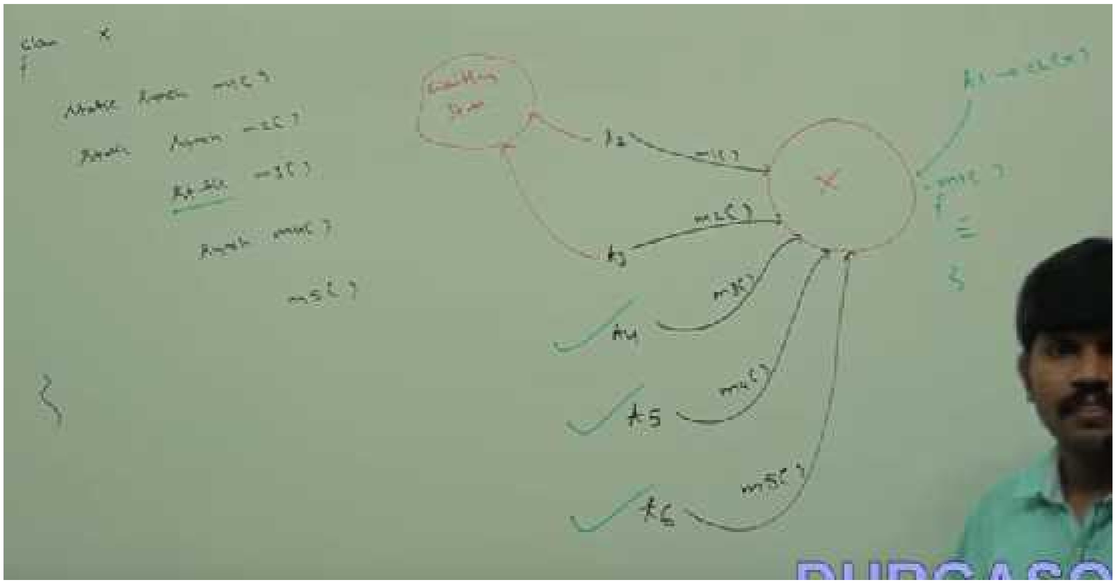
Once method execution completes automatically thread releases the lock.

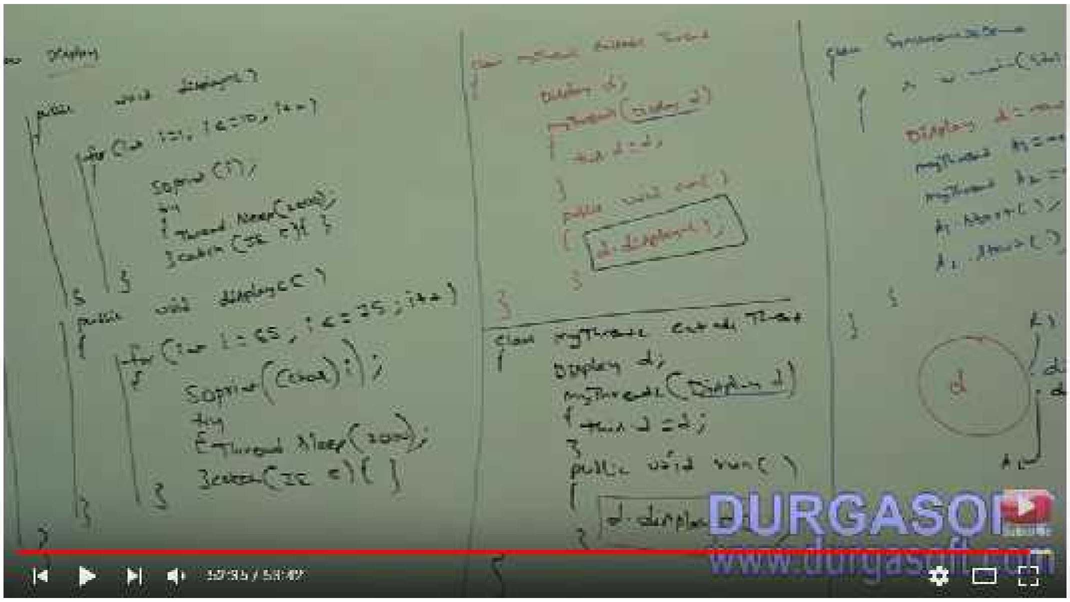
While a thread executing static syncronized method the remaining threads are not allowed to execute any static syncronized method of that class simuntaneously, but remaining threads are allowed to execute the following methods simultaneously,

1. normal static methods

2. syncronized instance methods

3. noramal instance methods



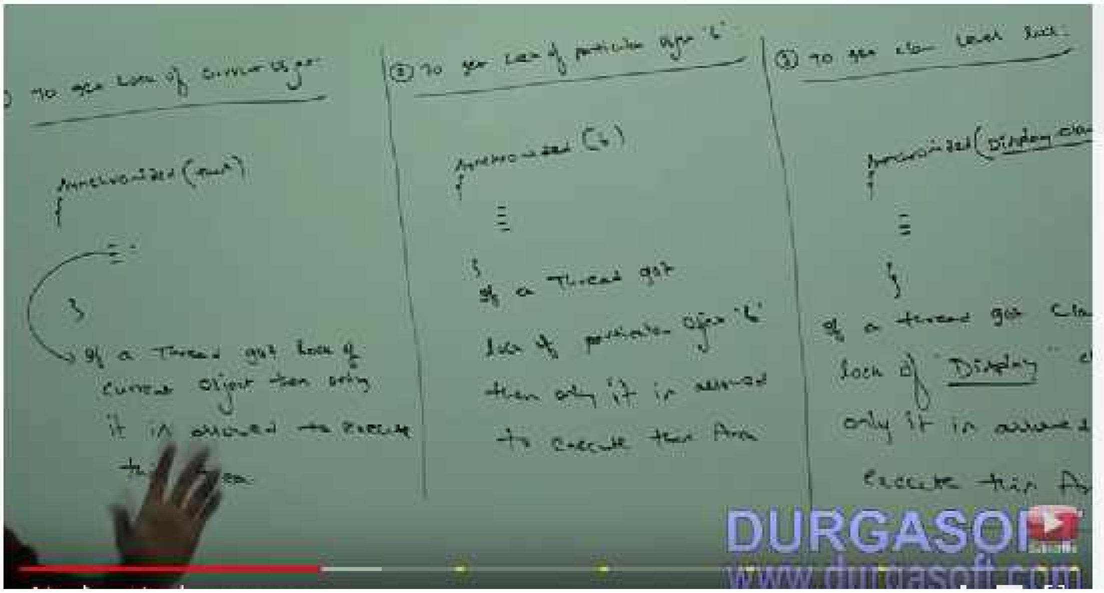


Syncronized Block:

If very few lines of the code require syncronization then it is not recommanded to declare entire method as syncronized. We have to enclose those few lines of the code by using syncronized block.

The main advantadage of syncronized block over syncronized method is it reduces waiting time of threads and improves performance of the application.

We can declare syncronized block as follows,



① To get lock of current object

```
synchronized(this)
{
  :
}
```

If a Thread got lock of current object then only it is allowed to execute this area

② To get lock of particular object 'b'

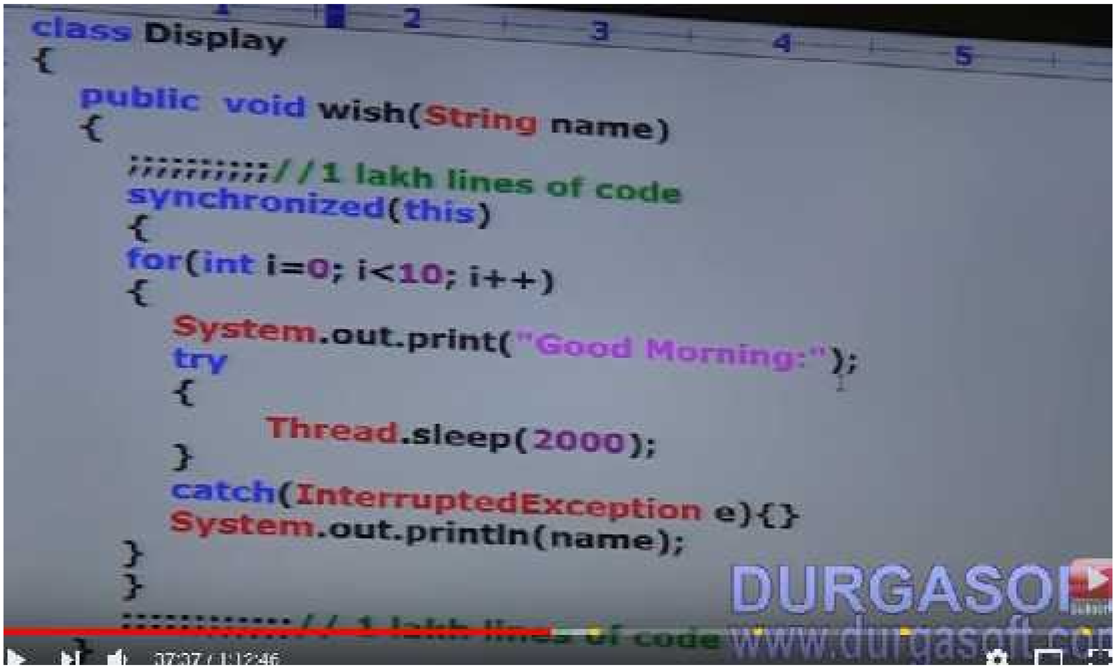
```
synchronized(b)
{
  :
}
```

If a Thread got lock of particular object 'b' then only it is allowed to execute this Area

③ To get class level lock

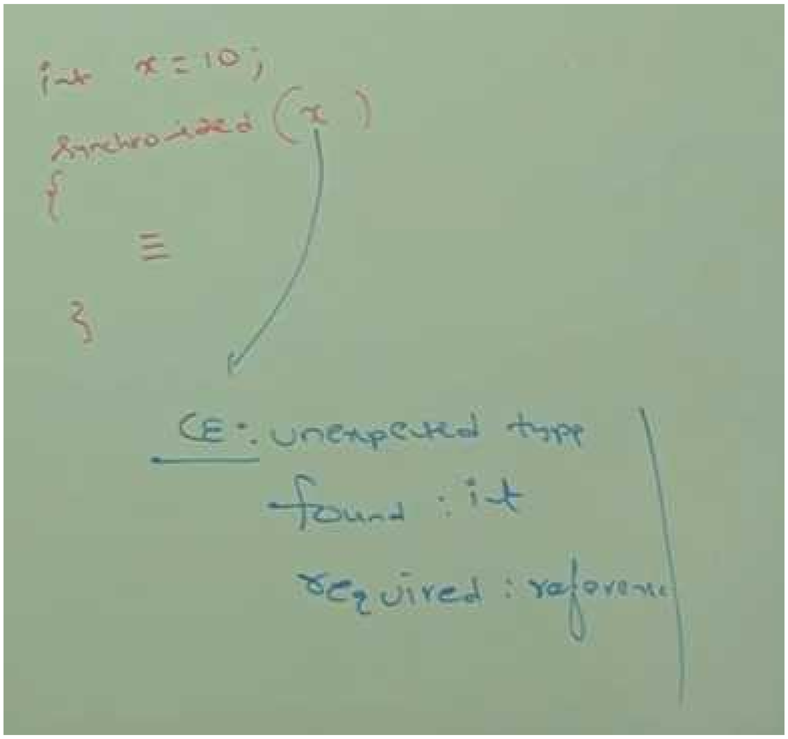
```
synchronized(Display cla
{
  :
}
```

If a thread got cla
lock of "Display" c
only it is allowed t
execute this A



```
class Display
{
    public  void wish(String name)
    {
        ;;;;;;;;;;;//1 lakh lines of code
        synchronized(this)
        {
        for(int i=0; i<10; i++)
        {
            System.out.print("Good Morning:");
            try
            {
                  Thread.sleep(2000);
            }
            catch(InterruptedException e){}
            System.out.println(name);
        }
        }
        ;;;;;;;;;;;;//1 lakh lines of code
```

Lock concept applicaple for object types and class types but not for premitives. hence we can't pass premitive type as argument to syncronized block otherwise we will get compiletime error saying unexpected type found int, required reference.



```
int x=10;
synchronized(x)
{
    ≡
}

CE: unexpected type
    found : int
    required : reference
```

FAQs:

1. What is syncronized keyword and where we can apply ?

2. Explain advantadge of syncronized ketword ?

3. Explain disadvantadge of syncronized keyword ?

4. What is race condition ?

Ans: If multiple threads are operating simultaneously on same java object then there may be a chance of data inconsistancy problem. This is called race condition.

We can overcome this problem by using syncronized keyword.

5. What is object lock and when it is required ?

6. What is class level lock and when it is required ?

7. What is the difference between class level lock and Object level lock

8. While a thread executing syncronized method on the given object is the remaining threds are allowed to ececute any other syncronized method simutaneously on the same object ?(No)

9. What is syncronized block ?

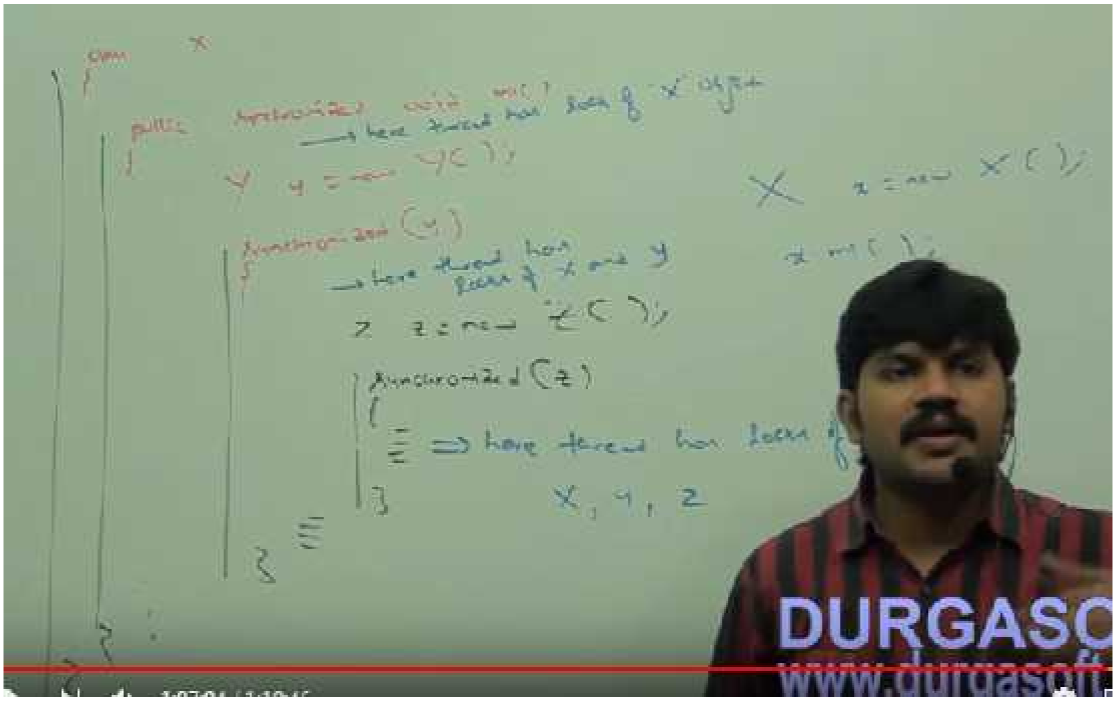
10. How to declare syncronized block to get lock of current object ?

11. How to declare syncronized block to get class level lock ?

12. What is the advantadge of syncronized block over syncronized method ?

13. Is a thread can acquire multiple locks simultaneously ?(Yes)

From different objects:


class X
{
public synchronized void m()
{
→ here thread has lock of 'X' object
Y y = new Y();
synchronized (y)
{
→ here thread has lock of X and y
Z z = new Z();
synchronized (z)
{
⇒ here thread has lock of
X, Y, Z
}
}
}
}
X x = new X();
x.m();
DURGASO
www.durgasoft

Q. What is syncronized statements ?

Ans: Interview people created terminology. The statements present in syncronized method and syncronized block are called syncronized statements.

Inter Thread Communication:

Two threads can communicate with each other by using wait(), notify() and notifyAll() methods.

The thread which is expecting updation is resposible to call wait() method then immediately the thread will entered into waiting state.

The thread whis is responsible to perform updation, after performing updation it is responsible to call notify(), method then waiting thread will get that notification and continue it's execution with those updated items.

**Wait(), notyfy(), notifyAll(), methods present in object class but not in thread class because, thread can call these methods on any java object.

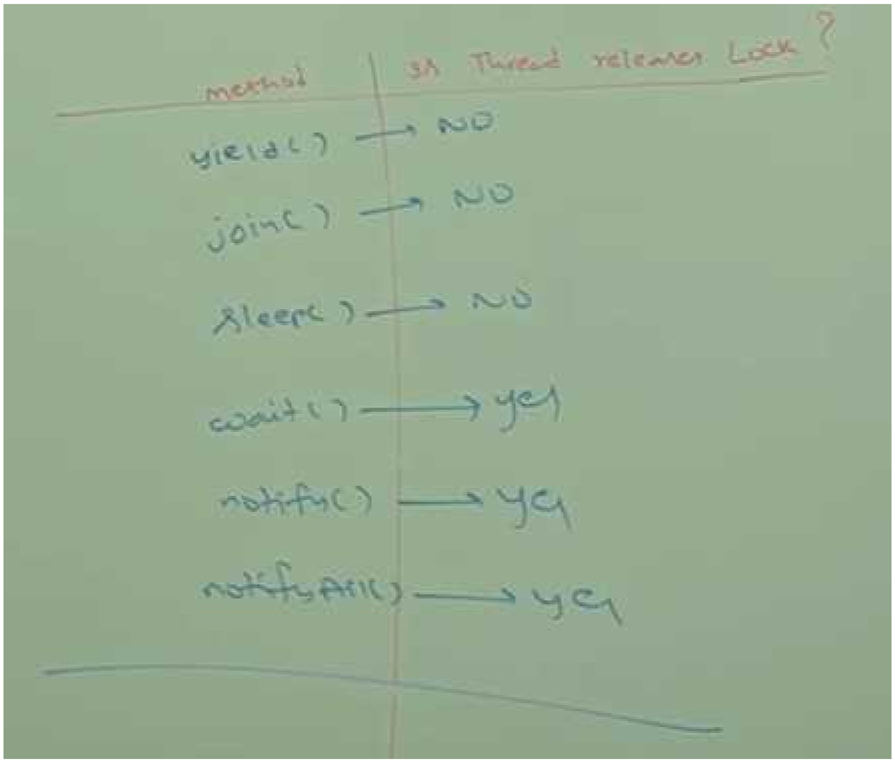
To call wait(), notify() or notifyAll(), methods on any object, thread should be owner of that object i.e. the thread should has acquired lock of that object i.e. the thread should be inside syncronized area.

Hence we can call wait(), notify() or notifyAll(), methods only from syncronized area otherwise we will get runtime exception saying IlligalmonitersateException.

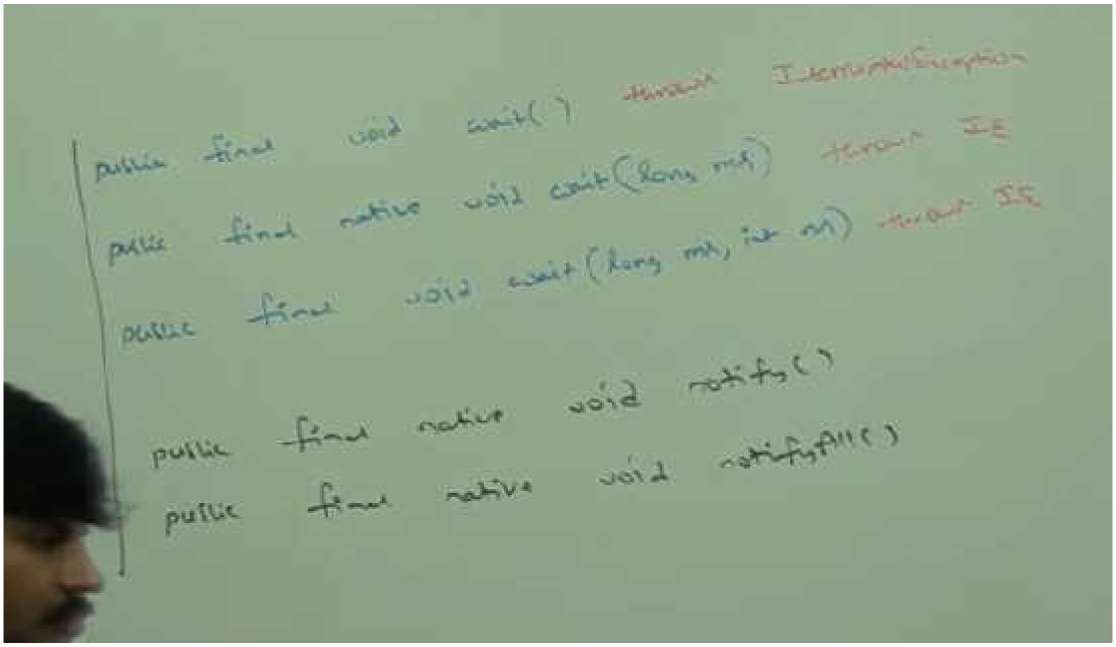
If a thread calls wait(), method on any object, it immediately releses lock of that particular object and entered into waiting state.

If a thread calls notify method on any object it releses the lock of that object but may not immediately.

Except wait(), notify() and notifyAll() there is no other all where thread releses the lock.



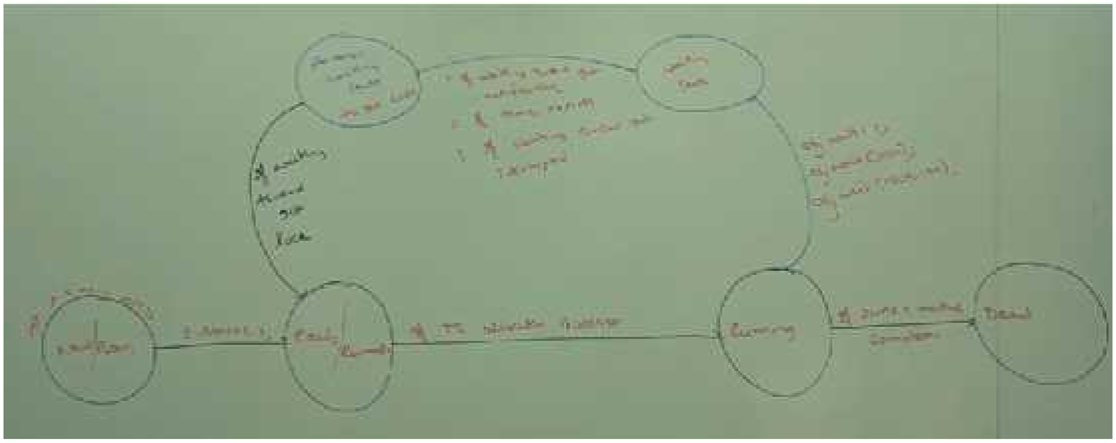
method	Is Thread releases Lock ?
yield() →	NO
join() →	NO
sleep() →	NO
wait() →	yes
notify() →	yes
notifyAll() →	yes

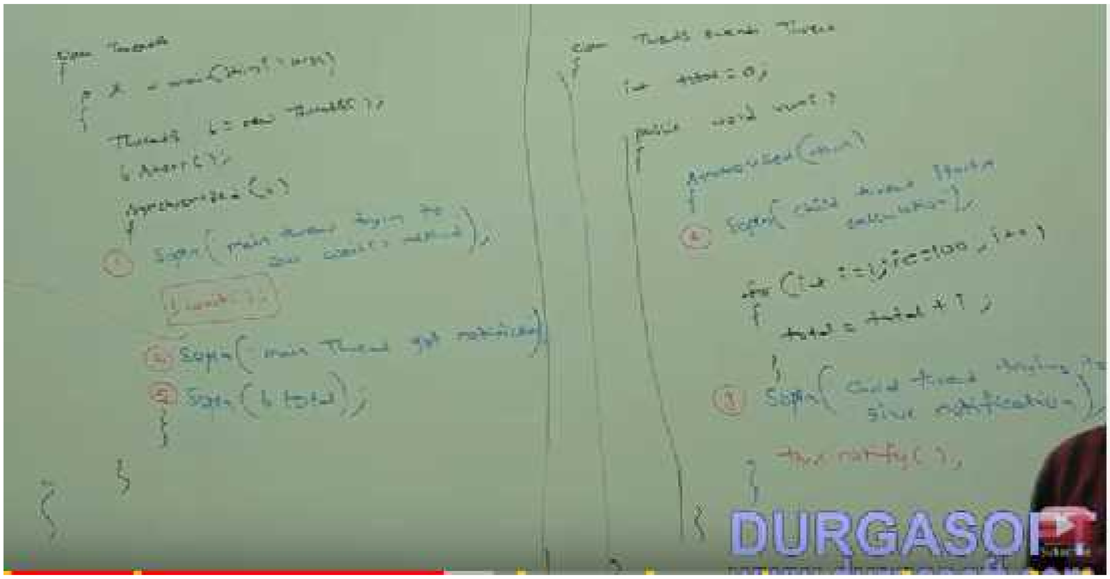


```
public final void wait() throws InterruptedException
public final native void wait(long ms) throws IE
public final void wait(long ms, int ns) throws IE

public final native void notify()
public final native void notifyAll()
```

*Note:

Every wait(), method throws interrupted exception which is checked exception, hence whenever we are using wait(), method compulsory we should handle this interrupted exception either by try catch or by throws, otherwise we ll get compile time error.





```
class ThreadA
{
 p s v main(String[] args)
 {
  ThreadB b = new ThreadB();
  b.start();
  synchronized(b)
  {
 ① Sopln("main thread trying to
        call wait() method");
    b.wait();
  ④ Sopln("main Thread got notification");
  ⑤ Sopln(b.total);
  }
 }
}
```

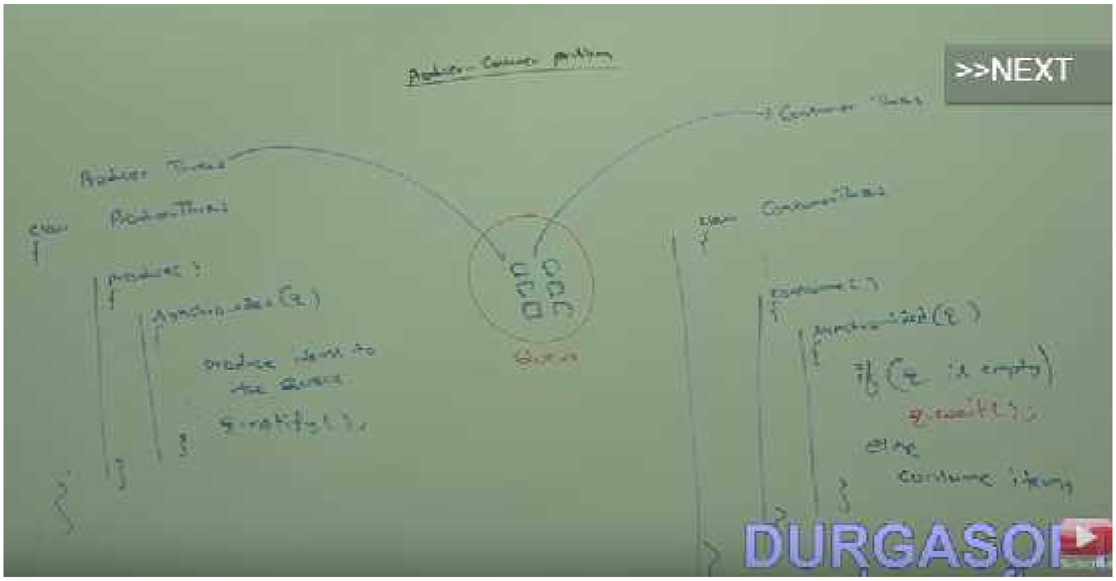
```
class ThreadB extends Thread
{
 int total=0;
 public void run()
 {
  synchronized(this)
  {
  ② Sopln("child thread starts
         calculation");
    for(int i=1;i<=100;i++)
    {
     total=total+i;
    }
  ③ Sopln("child thread trying to
         give notification");
    this.notify();
  }
 }
}
```

Producer-Consumer Problem:

producer thread is responsible to produce items to the queue and consumer thread is responsible to consume items from the queue.

If queue is empty then consumer thread will call wait(), method and entered into waithing state.

After producing items to the queue producer thread is responsible to call notify(), method then waiting consumer will get that notification and continue it's execution with updated items.



Q. Difference between notify() and notifyAll() ?

We can use notify method to give the notification for only one waiting thread. If multiple threads are waiting then only one thread will be notified and the remaining threads have to wait for the further notifications.

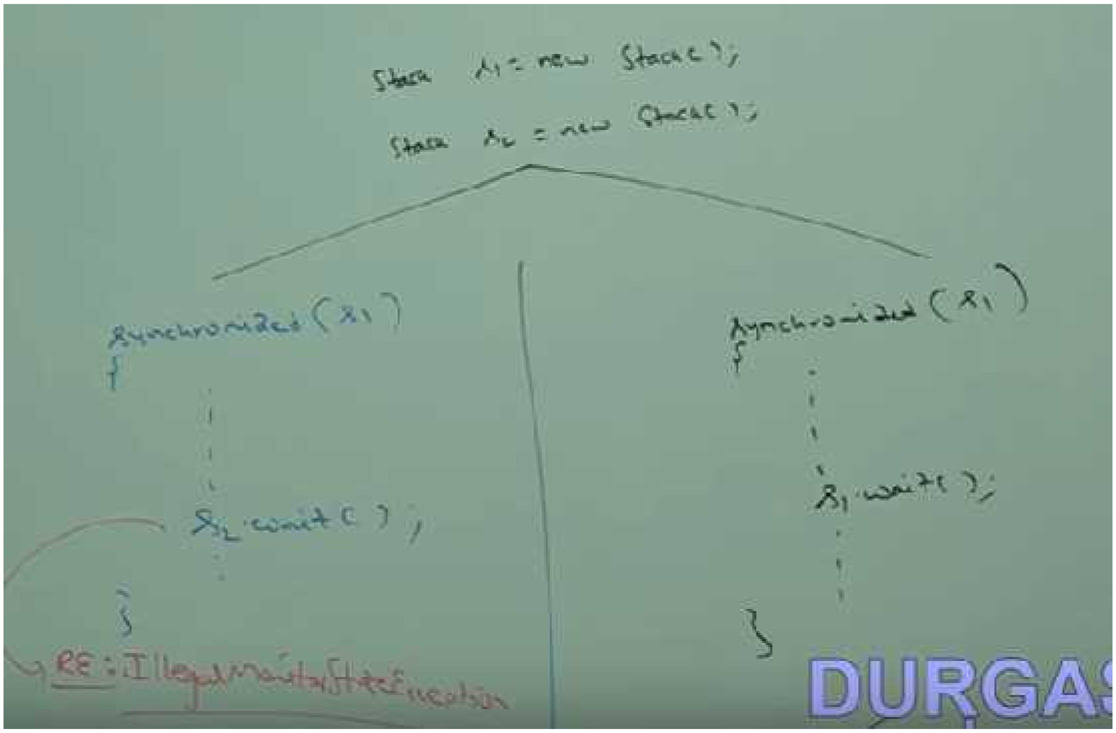
Which thread will be notified we can't except. It depends on JVM.

We can use notifyAll(), to give the notification for all waiting threads of a particular object.

Even though multiple threds notified but execution will be performed one by one because threads required lock and only one lock is available.

On which object we are calling wait(), thread require the lock of that particular object. e.g.

if we are calling wait(), in s1, then we have to get lock of s1 object but not s2 object.



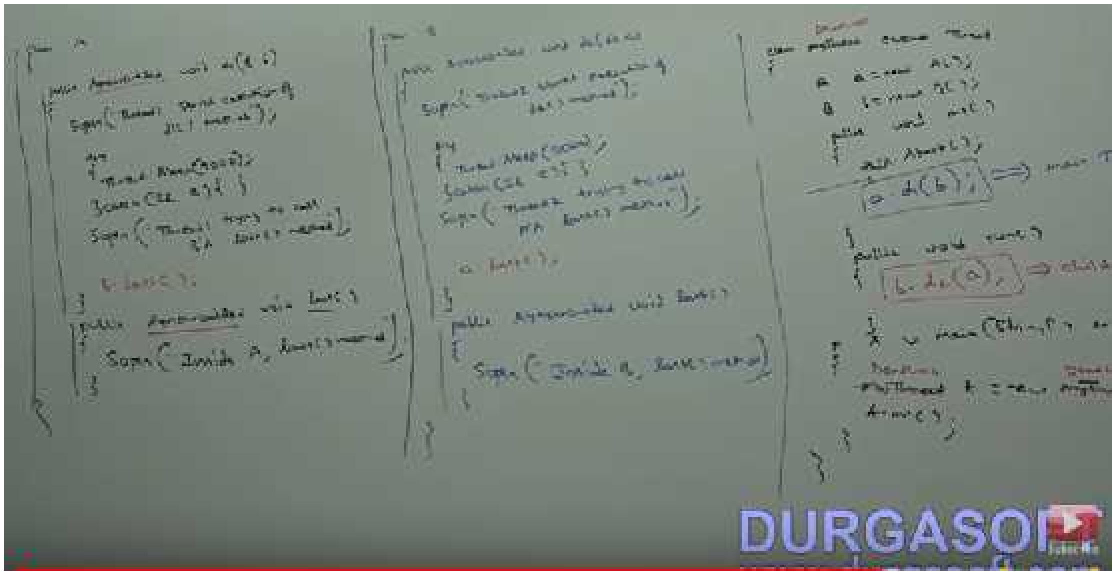
```
Stack s1 = new Stack();
Stack s2 = new Stack();

synchronized(s1)              synchronized(s1)
{                             {
    s2.wait();                    s1.wait();
}                             }
RE: IllegalMonitorStateException
```

Deadlock:

If two threads are waiting for each other forever such type of infinite waiting is called deadlock.

Syncronized keyword is the only reason for deadlock situation. Hence while using syncronized keyword we have to take special care.

There are no resolution techniques for deadlock but several prevention techniques are available.



Deadlock vs Starvation:

Long waiting of a thread where waiting never ends is called deadlock.

where as long waiting of a thread where waiting ends at certain point is callled Starvation.

e.g. Low priority thread has to wait until completing all high prority threads, it may be long waiting but ends at certain point,which is nothing but starvation.

Daemon Threads:

The threads which are executing in the background are called daemon threads.

e.g.

1. Garbage Collector

2. Signal Dispatcher

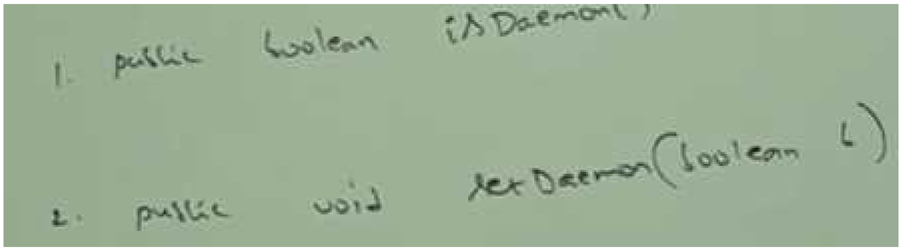
3. Attach listner etc

the main objective of demon threads is to provide support for non daemon threads(main thread)

e.g. if main thread runs with low memory then JVM runs GC to destroy useless objects so that number of bytes of free memory will be improved. With this freee memory main thread can continue it's execution.

Usually Demon threads having low prority but based on our requirement Demon threads can run with high prority also.

We can check demon nature of a thread by using isDemon(), method of Thread class.



1. public boolean isDaemon()

2. public void setDaemon(boolean b)

We can change demon nature of a thread by using setDemon(boolean b), method.

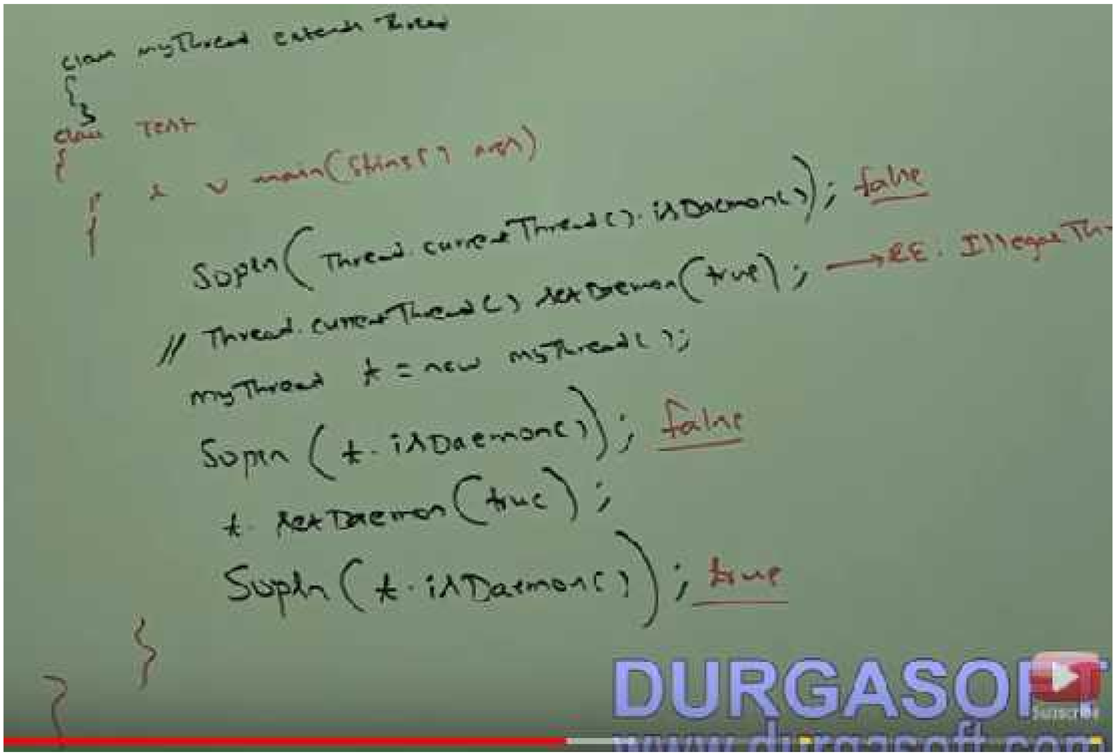
But changing Demon nature is possible before starting of a thread only. After starting a thread if we are trying to change demon nature then we will get runtime exception saying IlligalThreadState Exception.

Default Nature of a Thread:

By default main, thread is always non-demon and for all remaiining threads demon nature will be inherited from parent to child i.e. if the parent thread is demon then automatically child thread is also demon and if the parent thread is non demon then automatically child thread is also non daemon.

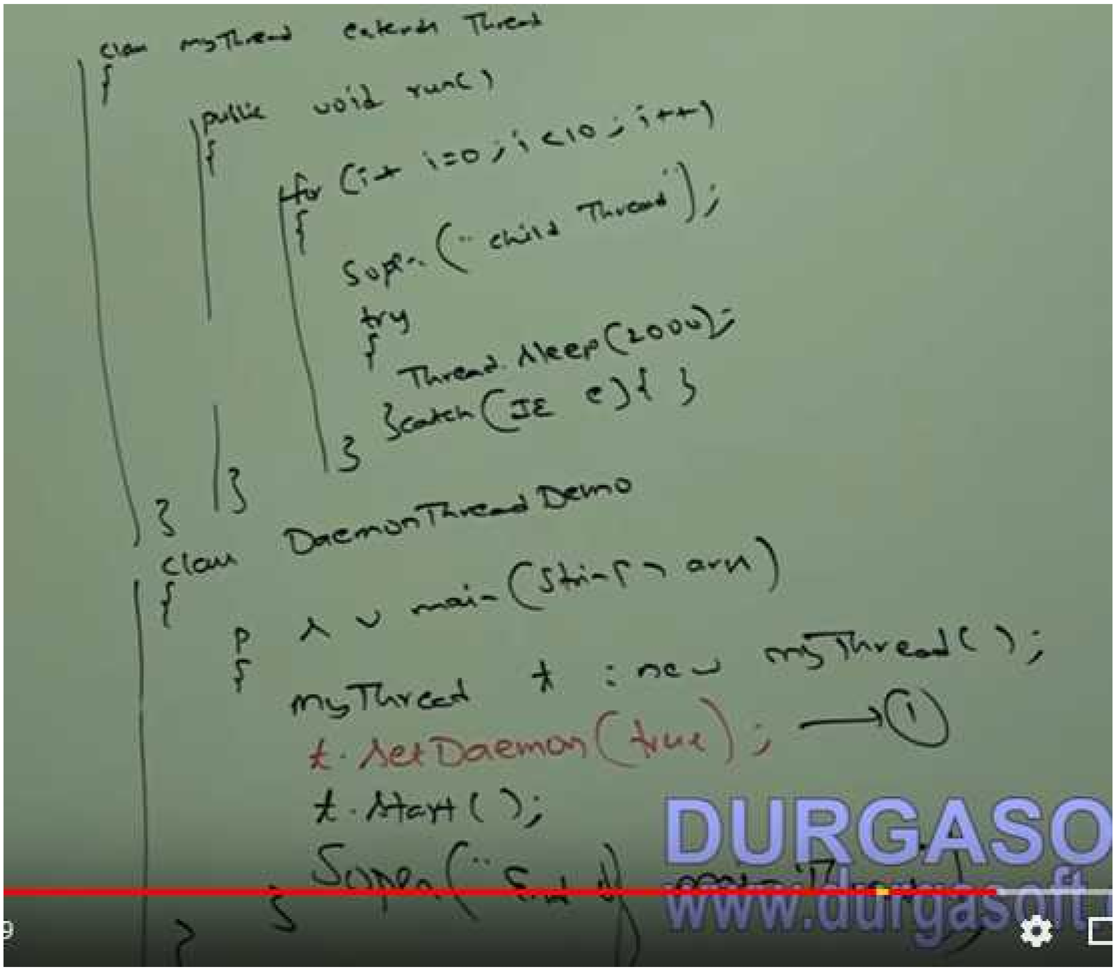
Note:

It is impossible to change daemon nature of main thread because it is already started by JVM at beginning.



```
class myThread extends Thread
{
}
class Test
{
    p s v main(String[] args)
    {
        Sopln(Thread.currentThread().isDaemon()); false
        // Thread.currentThread().setDaemon(true); → RE: IllegalTh
        myThread t = new myThread();
        Sopln(t.isDaemon()); false
        t.setDaemon(true);
        Sopln(t.isDaemon()); true
    }
}
```

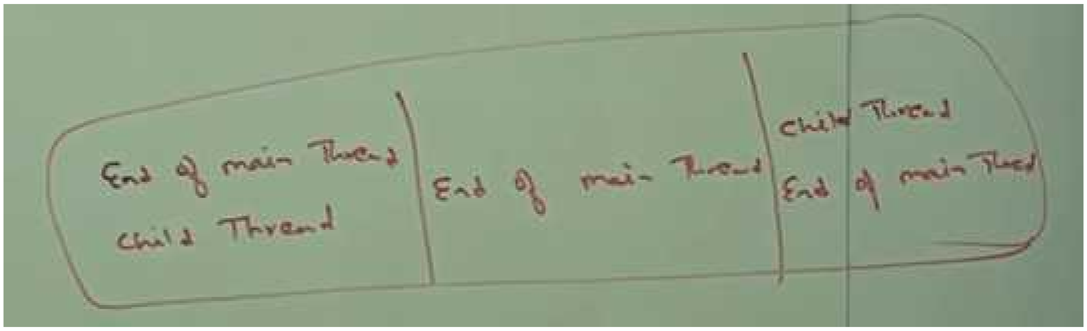
Whenever last non-daemon thread terminates automatically all daemon threads will be terminated irrerespective of their position.



```
class myThread extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            Sopln("child Thread");
            try
            {
                Thread.sleep(2000);
            }catch(IE e){ }
        }
    }
}
class DaemonThreadDemo
{
    p s v main(String[] args)
    {
        myThread t = new myThread();
        t.setDaemon(true); ⟶ ①
        t.start();
        Sopln("End of main Thread");
    }
}
```

if we are commenting line 1 both main and child threads are non-demon and hence both threads will be executed until their completion.

if we are not commenting line 1 then main thread is non demon and child thread is demon hence whenever main thread terminates automatically child thread will be terminated. In this case output is



```
End of main Thread  | End of main Thread | Child Thread
Child Thread        |                    | End of main Thread
```

=====

Green Thread Model:

The thread which is managed by JVM without taking underline OS support is called Green thread,

very few OS like Sun Solaris provide support for green thread model.

Anyway Green thread model is depricated and not recommandate to use.

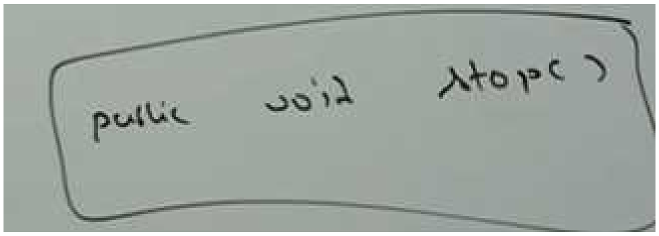
Native OS Model:

The thread which is managed by the JVM with the help of underlying OS, is called Native OS Model.

All Windows based OS provide support for native OS Model.

Q. How to stop a thread ?

We can stop a thread execution by using stop(), method of thread class.



```
public void stop()
```

If we call stop(), method then immediately the thread will entered into dead state, anyway stop(), method is depricated and not recommandate to use.

Q. How to suspend and resume of a thead ?

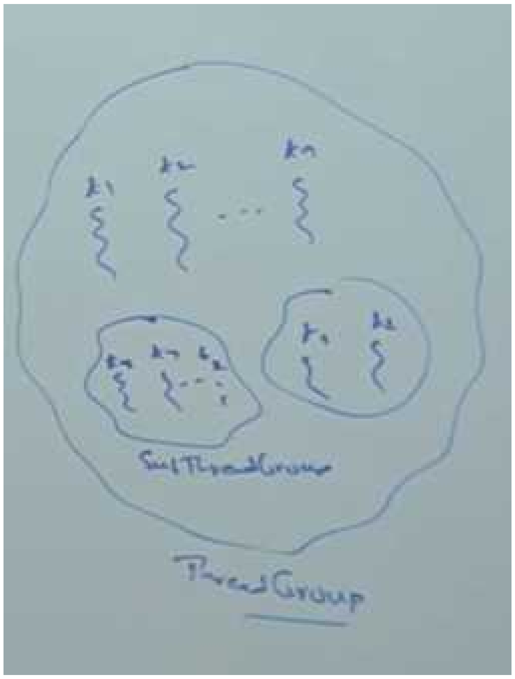
We can suspand a thread by using suspend method of thread class then immediately the thread will be entered into suspanded state. we can resume a suspended thered by using resume(), method of thread class.

Thread Group:

Based on functionality we can group threads into a single unit which is nothing but thread group.

i.e. Thread grop containns a group of threads.

In adition to threads thread group can also contain sub thread groups.



The main advantadge of maintaing threads in the form of thread group is we can perform common operations very easily.

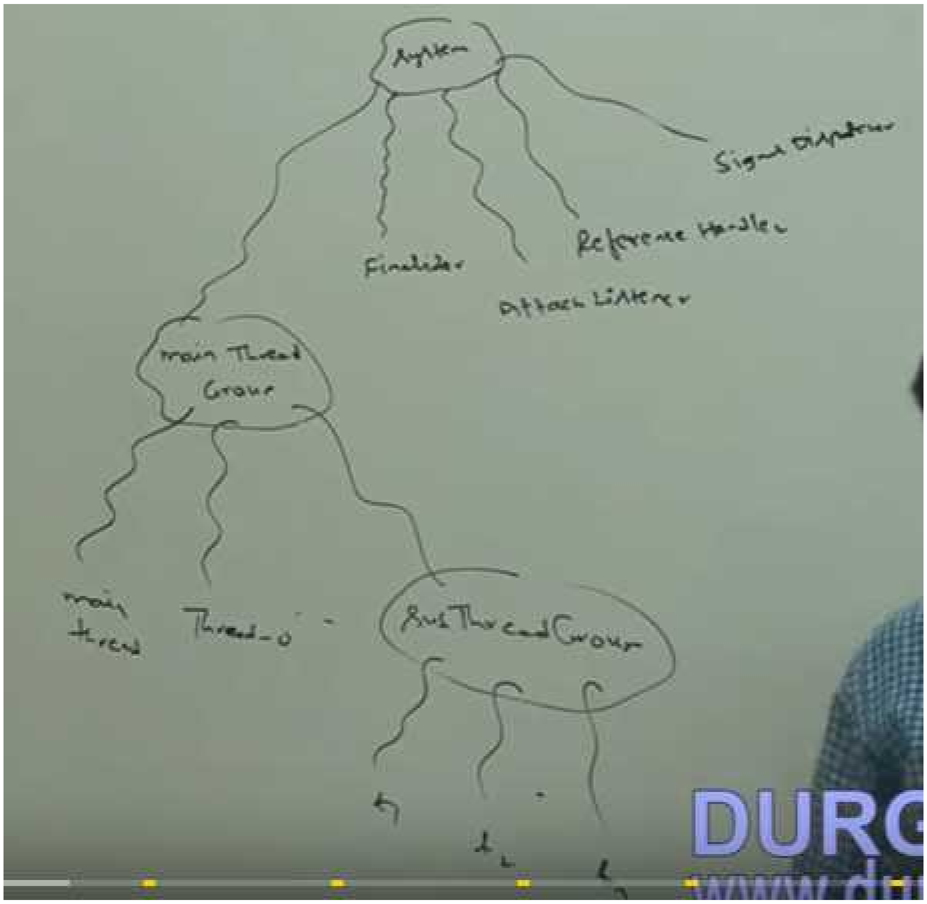
Every thread in java belongs to some group.

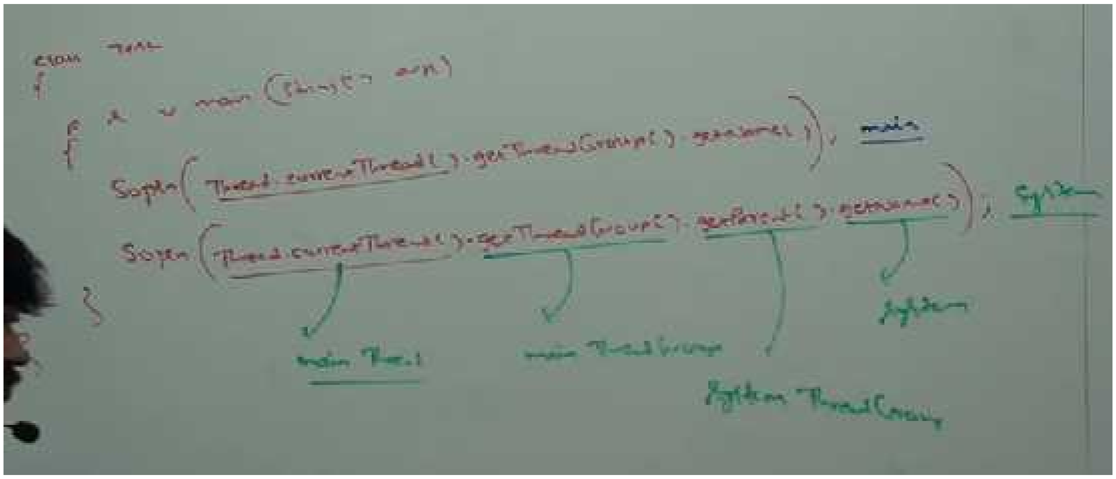
Main thread belongs to main group.

Every thread group in java is the child group of system group either directly or indirectly, hence system grroup acts as root for all thread groups in Java.

System group contains several system level threads like

1. Finalizer
2. Reference handler
3. signal dispatcher
4. Attach listner

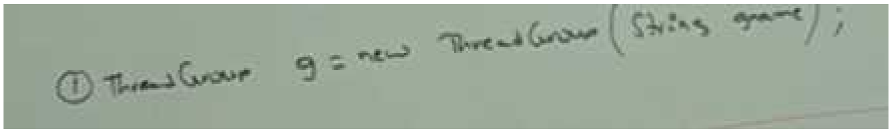




```
class Test
{
 p s v main(String[] args)
 {
  Sopln(Thread.currentThread().getThreadGroup().getName()); main
  Sopln(Thread.currentThread().getThreadGroup().getParent().getName()); System
 }
```

Thread group is a java class present in java.lang pkg and it is the direct child class of object.

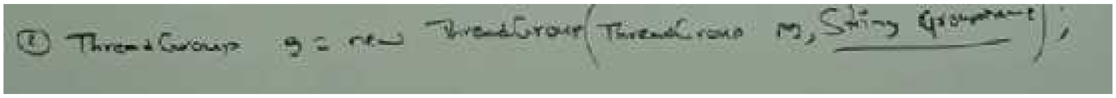
Constructors:



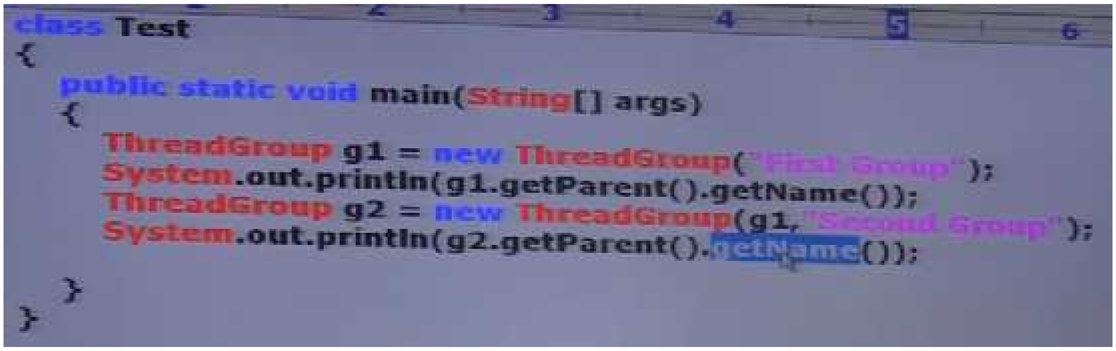
① ThreadGroup g = new ThreadGroup(String gname);

Creats a new thread group with the specified group name.

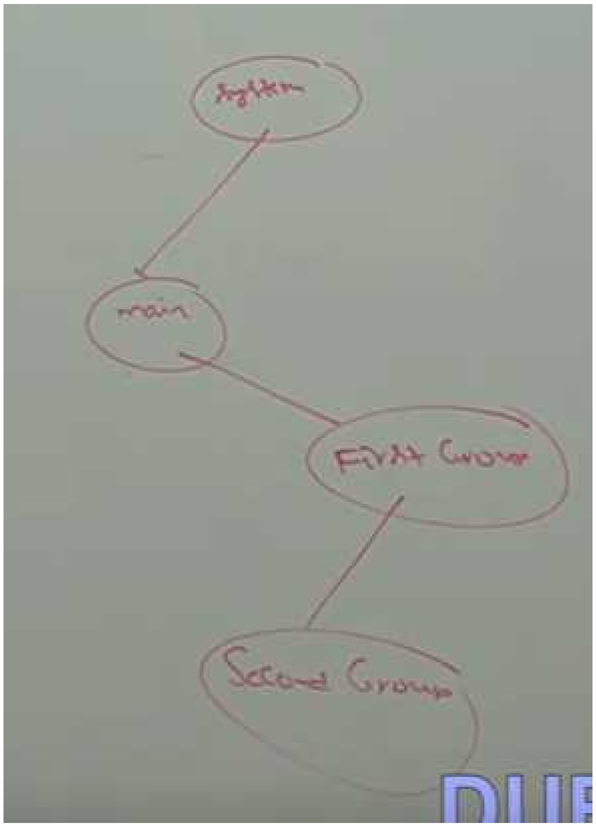
The parent of this new group is the thread group of currently executing thread.



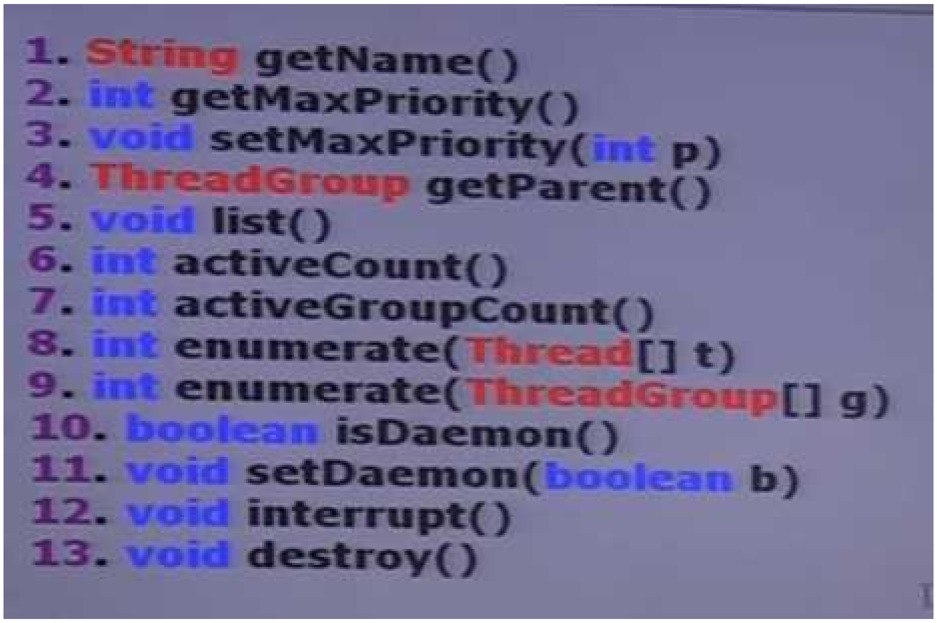
② ThreadGroup g = new ThreadGroup(ThreadGroup pg, String Groupname);



```
class Test
{
   public static void main(String[] args)
   {
      ThreadGroup g1 = new ThreadGroup("First Group");
      System.out.println(g1.getParent().getName());
      ThreadGroup g2 = new ThreadGroup(g1,"Second Group");
      System.out.println(g2.getParent().getName());

   }
}
```

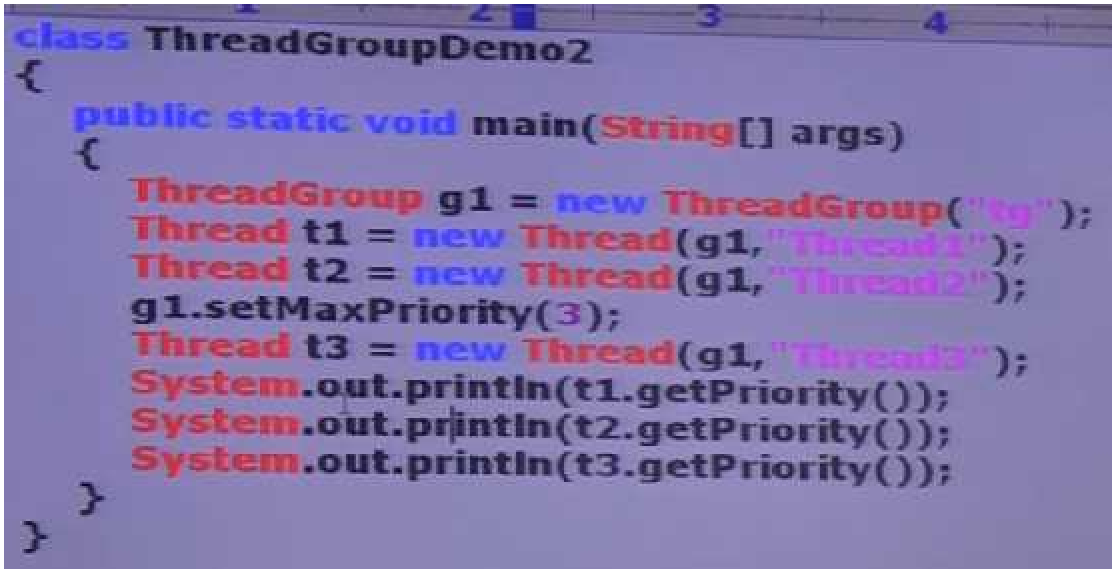


Important Methods of Thread group class:



1. String getName()
2. int getMaxPriority()
3. void setMaxPriority(int p)
4. ThreadGroup getParent()
5. void list()
6. int activeCount()
7. int activeGroupCount()
8. int enumerate(Thread[] t)
9. int enumerate(ThreadGroup[] g)
10. boolean isDaemon()
11. void setDaemon(boolean b)
12. void interrupt()
13. void destroy()

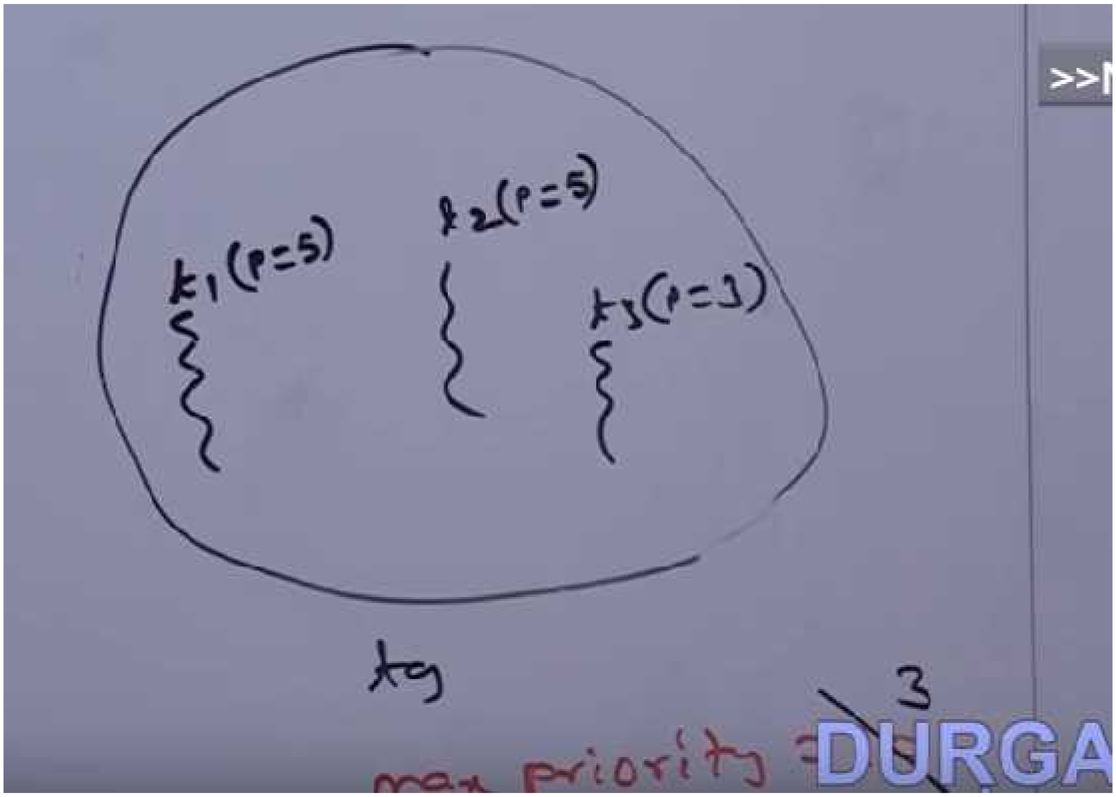
Threads in the thread group that have already have higher priority won't be affected, but for newly added threads this max priority is applicable.



```
class ThreadGroupDemo2
{
   public static void main(String[] args)
   {
      ThreadGroup g1 = new ThreadGroup("tg");
      Thread t1 = new Thread(g1,"Thread1");
      Thread t2 = new Thread(g1,"Thread2");
      g1.setMaxPriority(3);
      Thread t3 = new Thread(g1,"Thread3");
      System.out.println(t1.getPriority());
      System.out.println(t2.getPriority());
      System.out.println(t3.getPriority());
   }
}
```

o/p:

```
5

5

3
```



void list():

It prints information about thread group to the console.

int activeCount():

Returns number of active threads present in the thread group.

int activeGroupCount()

It teturns number of active groups present in the current thread group.

int enumerate(Thread[] t):

To copy all active threads of this thread group into provided thread array. In this case subthread group threads also will be considered.

int enumerate(ThreadGroup[] g)

To copy all active sub thread groups into thread group array.

boolean isDemon():

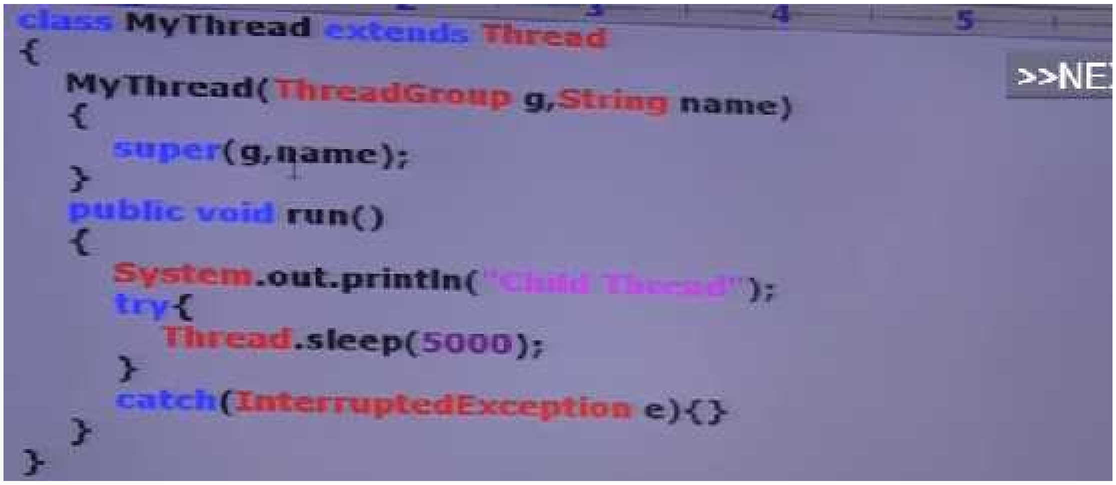
To check whether the thread group is demon or not.

void interrupt():

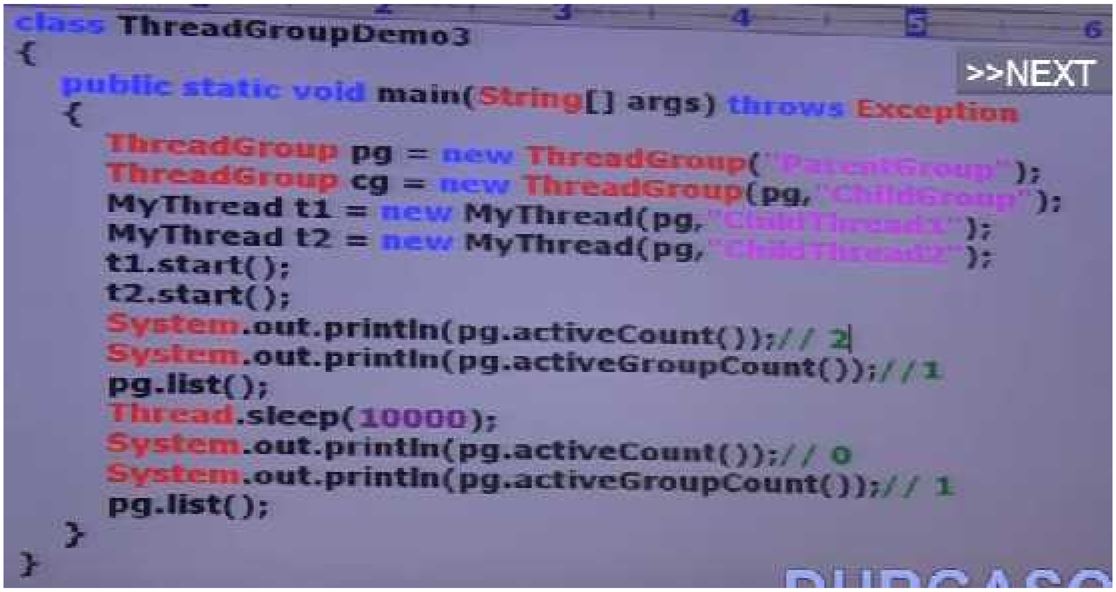
To interrupt all waiting or sleeping threads present in the thread group.

void destroy():

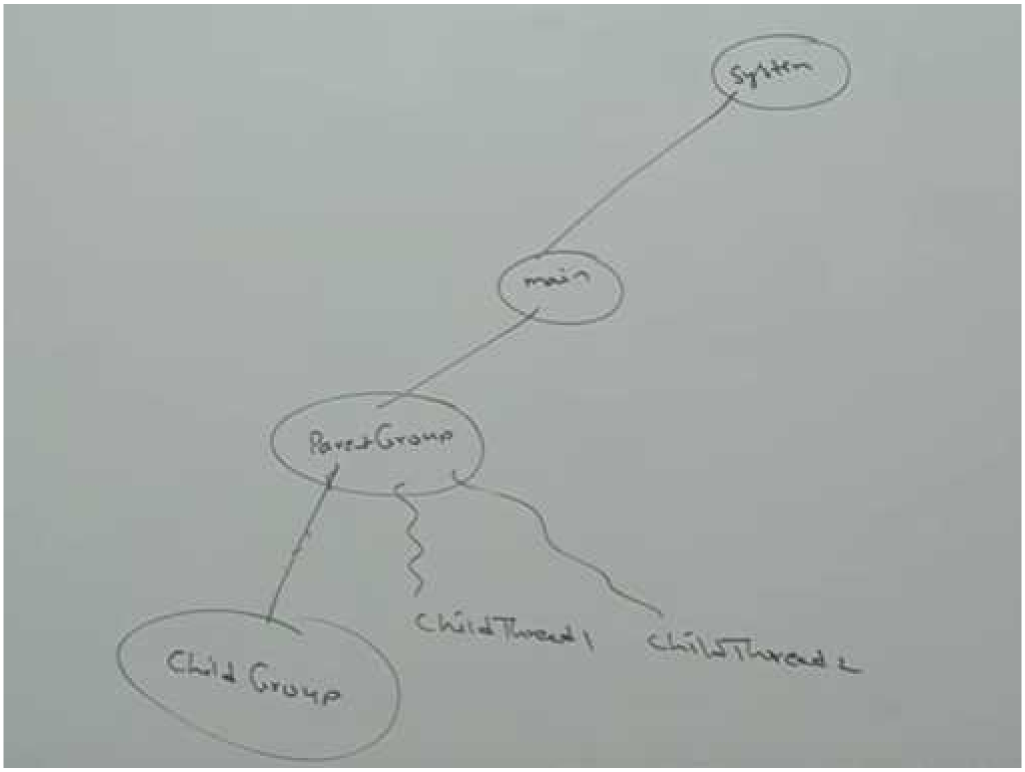
To destroy thread group and it's sub thread groups.



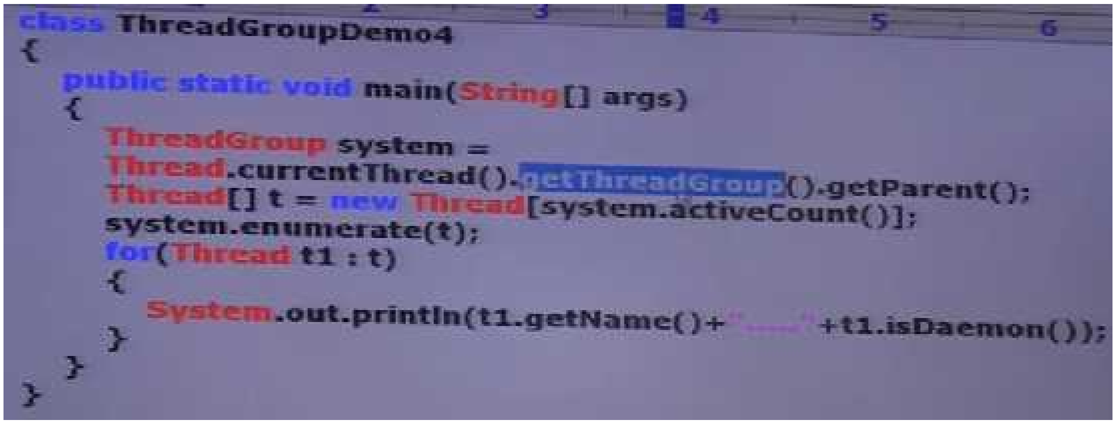
```
class MyThread extends Thread
{
   MyThread(ThreadGroup g,String name)
   {
     super(g,name);
   }
   public void run()
   {
     System.out.println("Child Thread");
     try{
       Thread.sleep(5000);
     }
     catch(InterruptedException e){}
   }
}
```



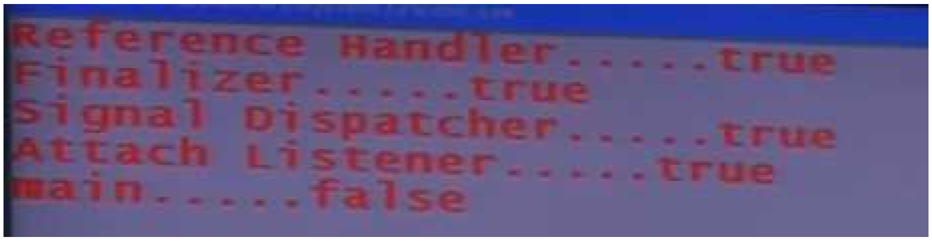
```
class ThreadGroupDemo3
{
  public static void main(String[] args) throws Exception
  {
    ThreadGroup pg = new ThreadGroup("ParentGroup");
    ThreadGroup cg = new ThreadGroup(pg,"ChildGroup");
    MyThread t1 = new MyThread(pg,"ChildThread1");
    MyThread t2 = new MyThread(pg,"ChildThread2");
    t1.start();
    t2.start();
    System.out.println(pg.activeCount());// 2
    System.out.println(pg.activeGroupCount());//1
    pg.list();
    Thread.sleep(10000);
    System.out.println(pg.activeCount());// 0
    System.out.println(pg.activeGroupCount());// 1
    pg.list();
  }
}
```



Q. Write a program to display all active thread names belongs to system group and it's child groups ?



```
class ThreadGroupDemo4
{
   public static void main(String[] args)
   {
      ThreadGroup system =
      Thread.currentThread().getThreadGroup().getParent();
      Thread[] t = new Thread[system.activeCount()];
      system.enumerate(t);
      for(Thread t1 : t)
      {
        System.out.println(t1.getName()+"....."+t1.isDaemon());
      }
   }
}
```



```
Reference Handler.....true
Finalizer.....true
Signal Dispatcher.....true
Attach Listener.....true
main.....false
```

java.util.concurrent pkg:

The problems with traditional syncronized keyword:

1. We are not having any flexibility to try for a lock without waiting.

2. There is no way to specify maximum waiting time for a thread to get lock so that thread will wait until getting the lock which may creats performance problems and which may cause deadlock.

3. If a thread releses lock then which waiting thrad will get that lock we are not having any control on this.

4. There is no API to list out all waiting threads for a lock.

5. The syncronized keyword we have to use either at method level or within the method and it is not possible to use accross multiple methods.

To overcome these problems Sun people introduced java.util.concurrent.locks pkg in 1.5 version.

It also provides several enhancements to the programmer to provide more control on concurrency.

Lock Interface:

Lock object is similar to implicit lock acquired by a thread to execute syncronized method or syncronized block.

Lock implementations provide more extensive operations than traditional implecit locks.

Important Methods of Lock interface:

1. void lock()

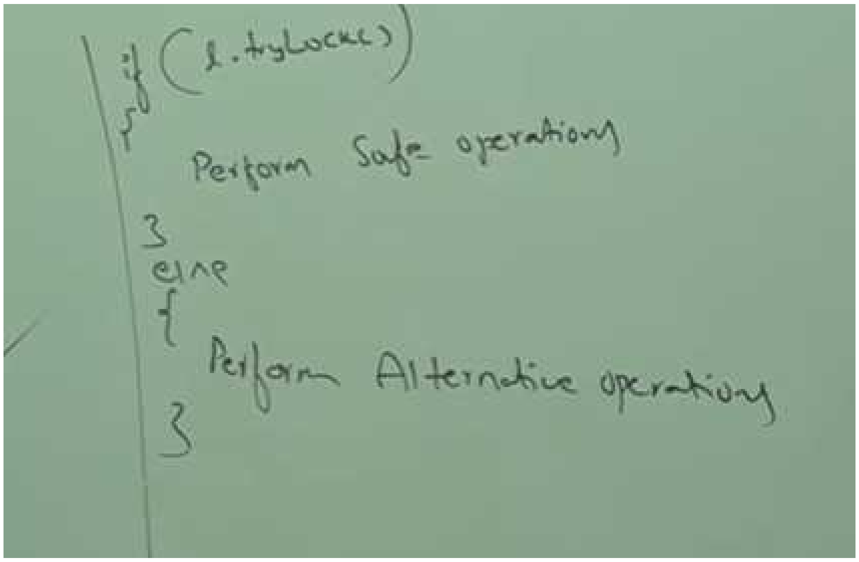
We can use this method to acquired a lock. if lock is already available then immediately current thread will get that lock.

If the lock is not already available then it will wait until getting the lock.

It is exactly same behaviour of traditional Syncronized keyword.

2. boolean tryLock()

To acquire the lock without waiting. If the lock is available then the thread acquires the lock and returns true, is the lock is not available this method returns false and can continue it's execution without waiting. In this case thread never be entered into waiting state.



```
if (l.tryLock())
{
    Perform Safe operations
}
else
{
    Perform Alternative operations
}
```

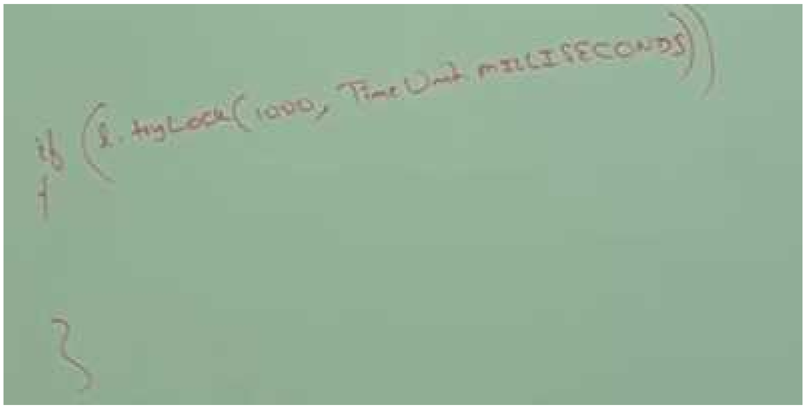
3. boolean tryLock(long time, TimeUnit unit)

If lock is available then the thread will get the lock and continue it's execution.

If the lock is not available then the thread will wait until specified amount of time still if the lock is not available then thread can continue it's execution.

TimeUnit:

Time unit is an enum present in java.util.concurrent pkg.



```
if (l.tryLock(1000, TimeUnit.MILLISECONDS))
{

}
```

4. void lockInterruptibly()

Acquires the lock if it is available and returns immediately.

If the lock is not available then it will wait. while waiting if the thread is interruped then thread won't get the lock.

5. void unlock()

To releses the lock.

To call this method compulsory current thread should be owner of the lock otherwise we ll get runtime exception saying IlligalMoniterStateException.

ReentrantLock(C):

It is the implementation class of lock interface and it is the direct child class of object.

Reentrant means a thread can acquire same lock multiple times without any issue.

Internally Reentrant lock increments threads personal count whenever we call lock method and decrements count value whenever thread calls unlock method and lock will be relesed whenever count reaches zero.

Constructors:

ReentrantLock l = new ReentrantLock()

Creats an instance of ReentrantLock.

ReentrantLock l = new ReentrantLock(boolean fairness)

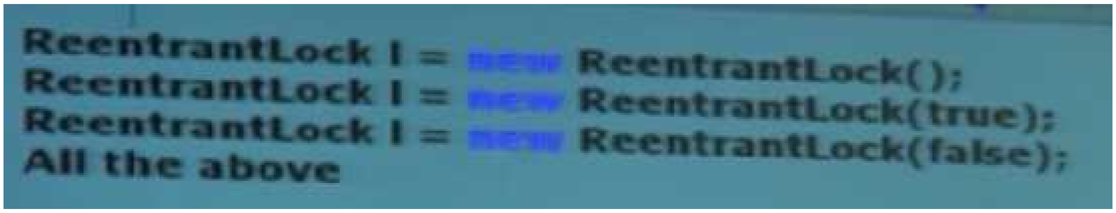
creats reentrantlock with the given fairness policy.

If the fairness is true then longest waiting thread can acquire the lock if it is available i.e. it follows first come first serve policy.

If fairness is false then which waiting thread will get the chance we can't expect.

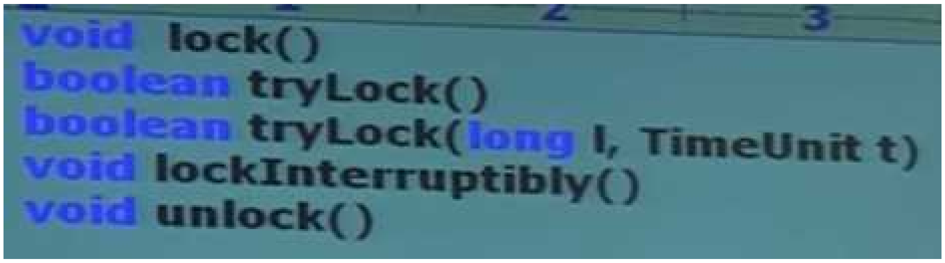
The default value for fairness is 'false'.

Q. Which of the following declarations are equal ?

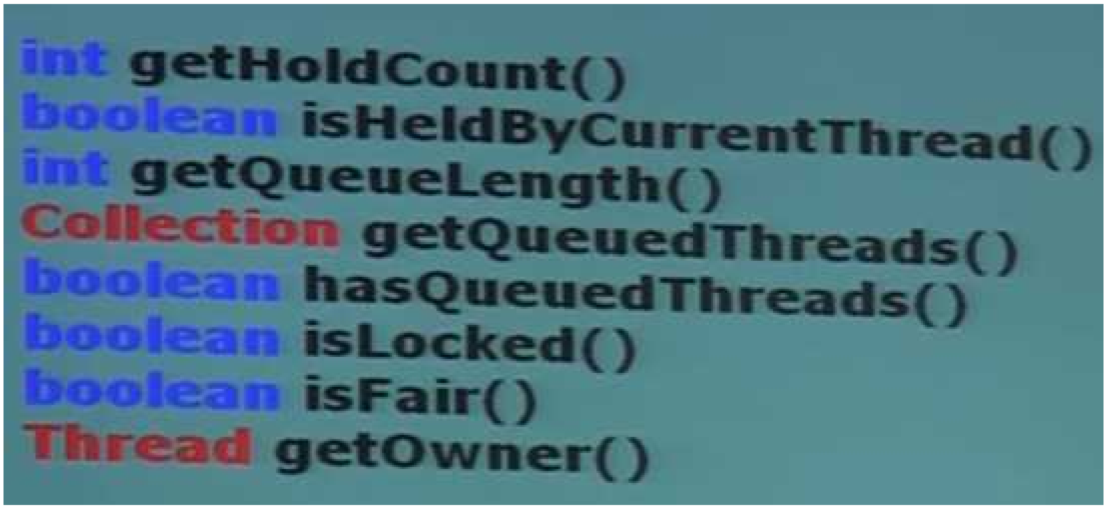


```
ReentrantLock l = new ReentrantLock();
ReentrantLock l = new ReentrantLock(true);
ReentrantLock l = new ReentrantLock(false);
All the above
```

Important methods of Reentrant Lock:



```
void  lock()
boolean tryLock()
boolean tryLock(long l, TimeUnit t)
void lockInterruptibly()
void unlock()
```



```
int getHoldCount()
boolean isHeldByCurrentThread()
int getQueueLength()
Collection getQueuedThreads()
boolean hasQueuedThreads()
boolean isLocked()
boolean isFair()
Thread getOwner()
```

int getHoldCount(), returns number of holdes on this lock by current thread.

boolean isHeldByCurrentThread(), returns true if and only if lock is hold by current thread.

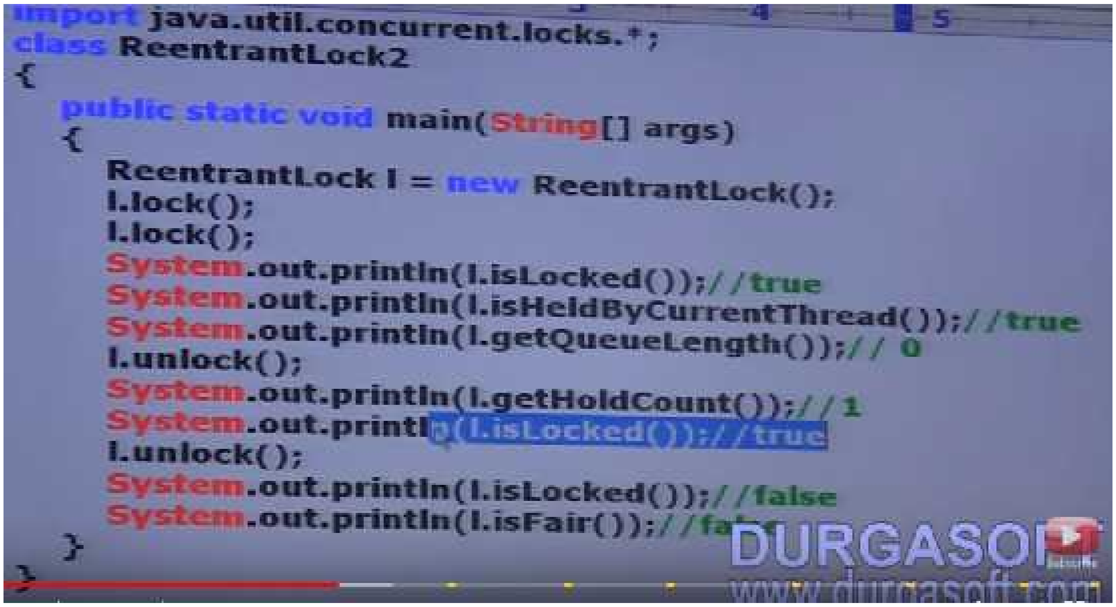
int getQueueLength(), returns number of threads witing for the lock.

Collection getQueuedThreads(), it returns a collection of threads which are waiting to get the lock.

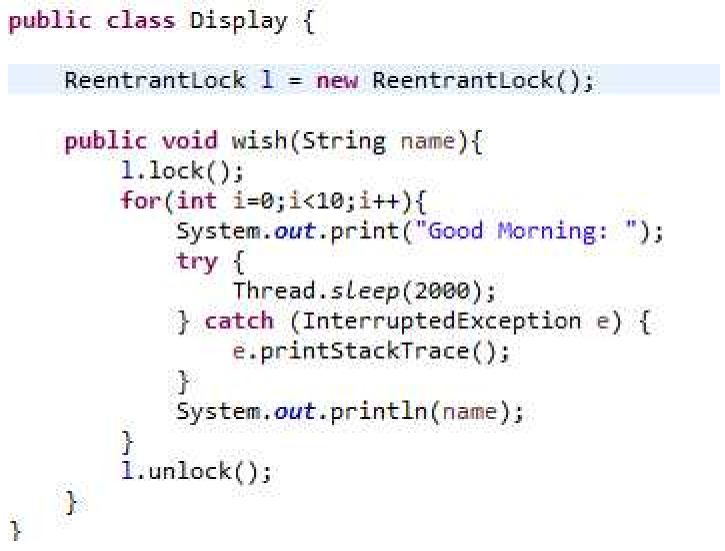
boolean hasQueuedThreads(), returns true if any thread waiting to get the lock.

boolean isLocked(), returns true if the lock is acquired by some thread.

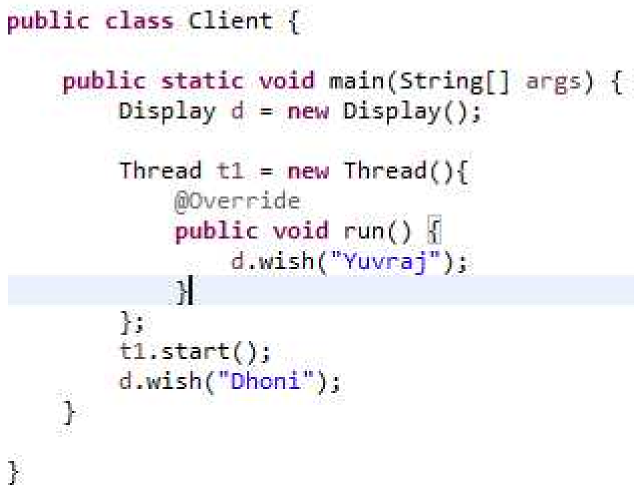
Thread Owner(), returns the thread which acquires the lock.



```
import java.util.concurrent.locks.*;
class ReentrantLock2
{
    public static void main(String[] args)
    {
        ReentrantLock l = new ReentrantLock();
        l.lock();
        l.lock();
        System.out.println(l.isLocked());//true
        System.out.println(l.isHeldByCurrentThread());//true
        System.out.println(l.getQueueLength());// 0
        l.unlock();
        System.out.println(l.getHoldCount());//1
        System.out.println(l.isLocked());//true
        l.unlock();
        System.out.println(l.isLocked());//false
        System.out.println(l.isFair());//fa
    }
}
```

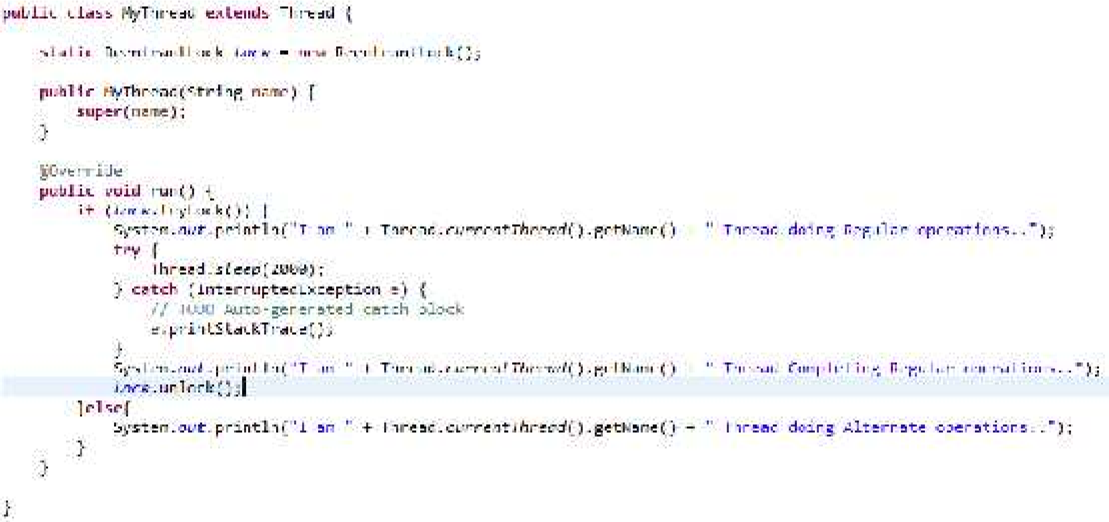


```java
public class Display {

    ReentrantLock l = new ReentrantLock();

    public void wish(String name){
        l.lock();
        for(int i=0;i<10;i++){
            System.out.print("Good Morning: ");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(name);
        }
        l.unlock();
    }
}
```

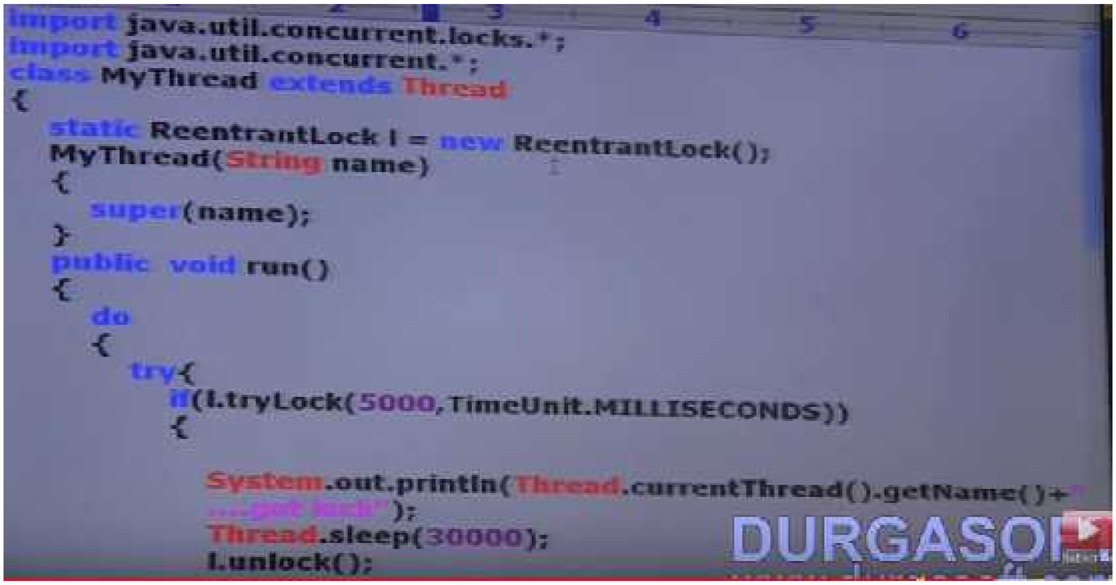


```java
public class Client {

    public static void main(String[] args) {
        Display d = new Display();

        Thread t1 = new Thread(){
            @Override
            public void run() {
                d.wish("Yuvraj");
            }
        };
        t1.start();
        d.wish("Dhoni");
    }

}
```

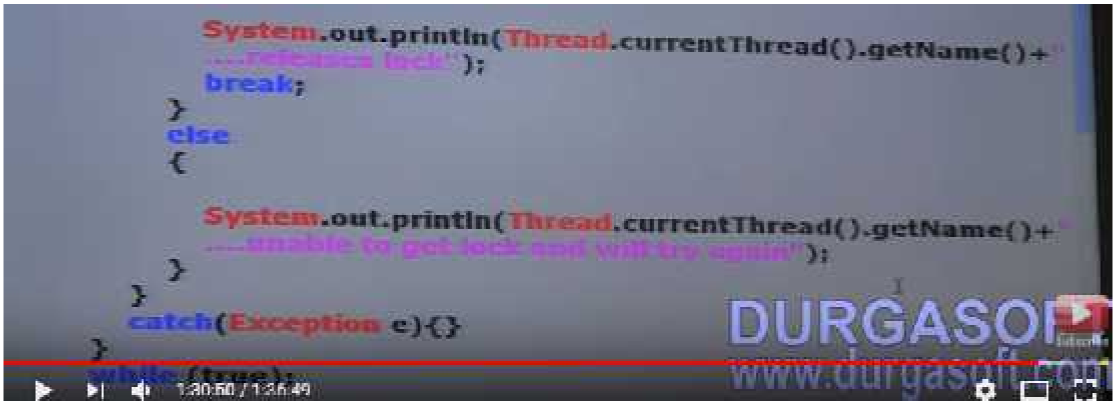
Demo Program for tryLock Method:



```
public class MyThread extends Thread {

    static ReentrantLock lock = new ReentrantLock();

    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        if (lock.tryLock()) {
            System.out.println("I am " + Thread.currentThread().getName() + " Thread doing Regular operations..");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println("I am " + Thread.currentThread().getName() + " Thread Completing Regular operations..");
            lock.unlock();
        }else{
            System.out.println("I am " + Thread.currentThread().getName() + " Thread doing Alternate operations..");
        }
    }

}
```



```
import java.util.concurrent.locks.*;
import java.util.concurrent.*;
class MyThread extends Thread
{
    static ReentrantLock l = new ReentrantLock();
    MyThread(String name)
    {
        super(name);
    }
    public void run()
    {
        do
        {
            try{
                if(l.tryLock(5000,TimeUnit.MILLISECONDS))
                {

                    System.out.println(Thread.currentThread().getName()+"
                    ....got lock");
                    Thread.sleep(30000);
                    l.unlock();
```



```
                    System.out.println(Thread.currentThread().getName()+"
                    ....releases lock");
                    break;
                }
                else
                {

                    System.out.println(Thread.currentThread().getName()+"
                    ....unable to get lock and will try again");
                }
            }
            catch(Exception e){}
        }
        while(true);
```

Thread Pools (Executer Framework):

Creating a new thread for every job may create performance and memory problems.

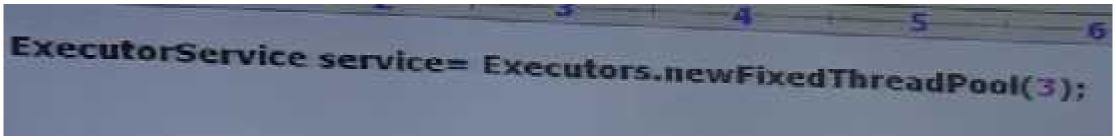
To overcome this we should go for thread pool.

Thread pool is pool of already created threads ready to do our Job.

Java 1.5 version introduces Thread pool framework to implement thread pools.

Thread pool framework also known as Executer framework.

We can create a thread pool as follows,



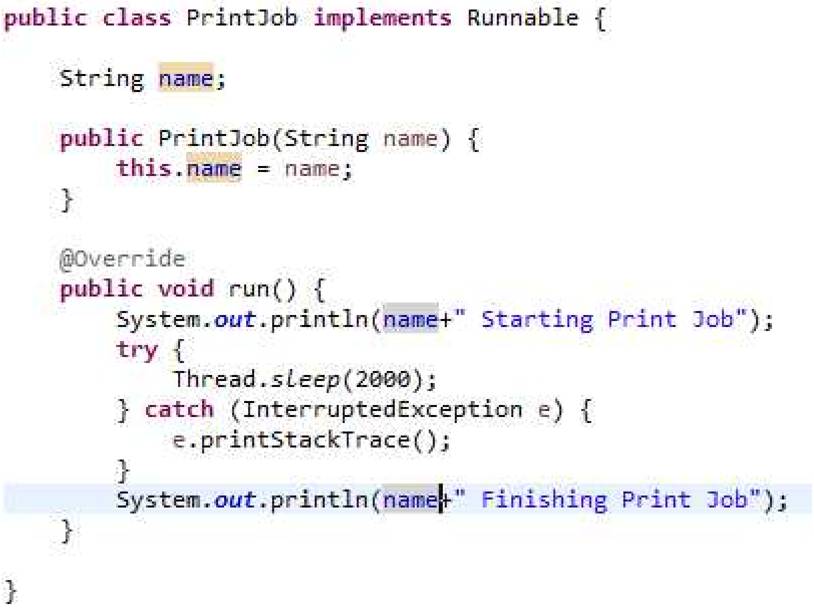
```
ExecutorService service= Executors.newFixedThreadPool(3);
```

We can submit a runnable job by using submit() method.

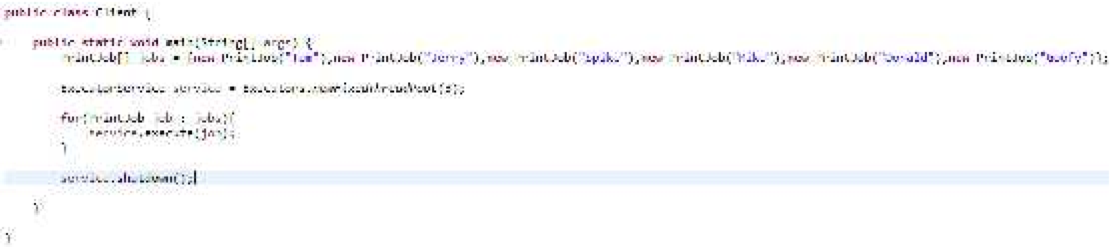
service.submit(job);

We can shutdown executer service by using shutdown method.

service.shutdown();



```
public class PrintJob implements Runnable {

    String name;

    public PrintJob(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println(name+" Starting Print Job");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(name+" Finishing Print Job");
    }

}
```



```
public class Client {

    public static void main(String[] args) {
        PrintJob[] jobs = {new PrintJob("Tom"),new PrintJob("Jerry"),new PrintJob("Spike"),new PrintJob("Mike"),new PrintJob("Donald"),new PrintJob("Goofy")};

        ExecutorService service = Executors.newFixedThreadPool(3);

        for(PrintJob job : jobs){
            service.execute(job);
        }

        service.shutdown();

    }
}
```

In the above example 3 threads are responsible to execute 6 jobs, so that a single thread can be reused for multiple jobs.

Note:

While developing web servers and application servers we can use thread pool concept.

=================

What is the difference between Executor.submit() and Executor.execute() method in Java? is one of the good multi-threading questions for experienced Java programmers, mostly asked in Investment Banks like Barclays, Deutsche Bank, or Citibank. A main difference between the submit() and execute() method is that ExecuterService.submit()can return result of computation because it has a return type of Future, but execute() method cannot return anything because it's return type is void.

The submit() can accept both Runnable and Callable task but execute() can only accept the Runnable task.

The return type of submit() method is a Future object but return type of execute() method is void.

Read more: http://javarevisited.blogspot.com/2016/04/difference-between-ExecutorServie-submit-vs-Executor-execute-method-in-Java.html#ixzz5ERWK8dwC

Read more: http://javarevisited.blogspot.com/2016/04/difference-between-ExecutorServie-submit-vs-Executor-execute-method-in-Java.html#ixzz5ERWCaryi

Read more: http://javarevisited.blogspot.com/2016/04/difference-between-ExecutorServie-submit-vs-Executor-execute-method-in-Java.html#ixzz5ERTjO2oP

Callable And Future:

In the case of Runable job, thread won't return anything after completing the job.

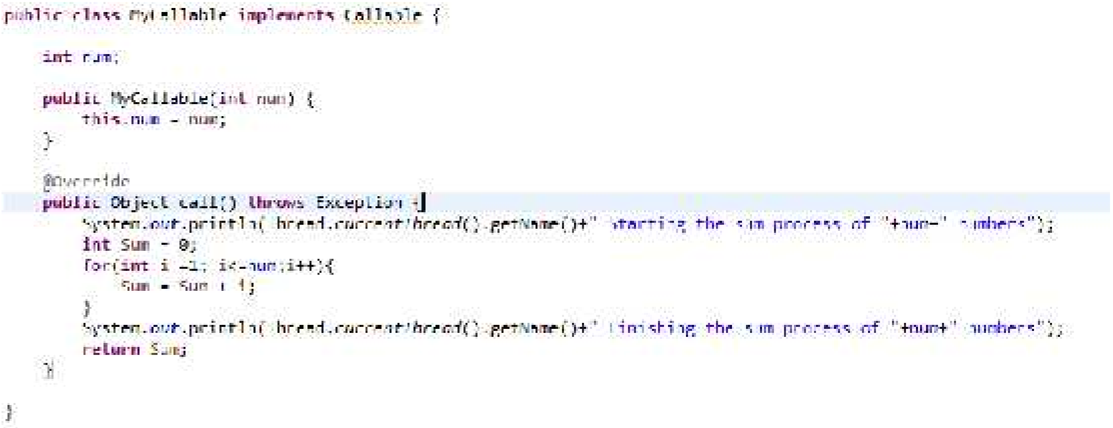
If a thread is required to return some result after execution then we should go for Callable.

Callable(I), contains call().

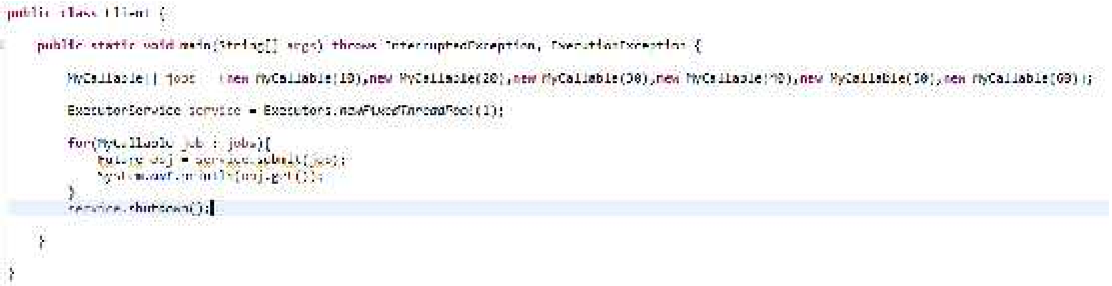
public Object call() throws Exception

If we submit callable object to executer then after completing the job thread returns an Object of the type Future.

i.e. Future object can be used to retrive the result from callable job.



```
public class MyCallable implements Callable {

    int num;

    public MyCallable(int num) {
        this.num = num;
    }

    @Override
    public Object call() throws Exception {
        System.out.println(Thread.currentThread().getName()+" starting the sum process of "+num+" numbers");
        int Sum = 0;
        for(int i =1; i<=num;i++){
            Sum = Sum + i;
        }
        System.out.println(Thread.currentThread().getName()+" finishing the sum process of "+num+" numbers");
        return Sum;
    }

}
```



```
public class Client {

    public static void main(String[] args) throws InterruptedException, ExecutionException {

        MyCallable[] jobs = {new MyCallable(10),new MyCallable(20),new MyCallable(30),new MyCallable(40),new MyCallable(50),new MyCallable(60)};

        ExecutorService service = Executors.newFixedThreadPool(3);

        for(MyCallable job : jobs){
            Future obj = service.submit(job);
            System.out.println(obj.get());
        }
        service.shutdown();

    }
}
```

Differences between Runnable and Callable:

If a thread is not required to return anything after completing the job then we should go for Runnable.

If a thread required to return something after completing the job then we should go for Callable.

Runnable interface contains only one method run().

Callable interface contains only one mthod call().

Runnable job not required to return anything and hence return type of run method is void.

Callable job is required to return something and return type of call method is object.

Withinn the run method if there is any chance of raising checked exception compusory we should handle by using try catch because we can't use throws keyword for run method.

Inside call method if there is any chance of raising checked exception we are not required to handle by using try catch because call method already throws exception.

Runnable ineterface present in java.lang pkg.

Callable Interface present in java.util.concurrent pkg.

Runnable Introduced in 1.0 version

Callable Introduced in 1.5 version.

ThreadLocal:

ThreadLocal class provide thread local variables.

It maintains values per thread basis.

Each thread local object maintains a separate value like userid, transaction id etc.. for each thread that accesses that object.

Thread can access it's local value, can manipulate it's value and even can remove it's value.

In every part of the code which is executed by the thread we can access it's local variable.

E.g. Consider a servlet which invokes some business methods.

We have a requirement to generate a unique transaction id for each and every request and we have to pass this transaction id to the business methods, for this requirement we can use thread local to maintain a separate transaction id for every request i.e. for every thread.

Note:

1. ThreadLocal class intoduced in 1.2 version and enhanced in 1.5 version

2. ThreadLocal can be associated with thread scope.

3. Total code which is executed by the thread has access to the corrosponding thread local variables.

4. A thread can access it's own local variables and can't access other threads local variables.

5. Once thread entered into dead state all it's local variables are by default eligible for GC.

Constructor:

ThreadLocal tl = new ThreadLocal()

Creates a thread local variable.

Methods:

1. Object get()

returns the value of threadLocal variable associated with current thread.

2. Object initialValue()

returns initial value of thread local variable associated with current thread.

The default implementation of this method returns null.

To customize our own initial value we have to override this method.

3. void set(Object newValue)

To set a new value

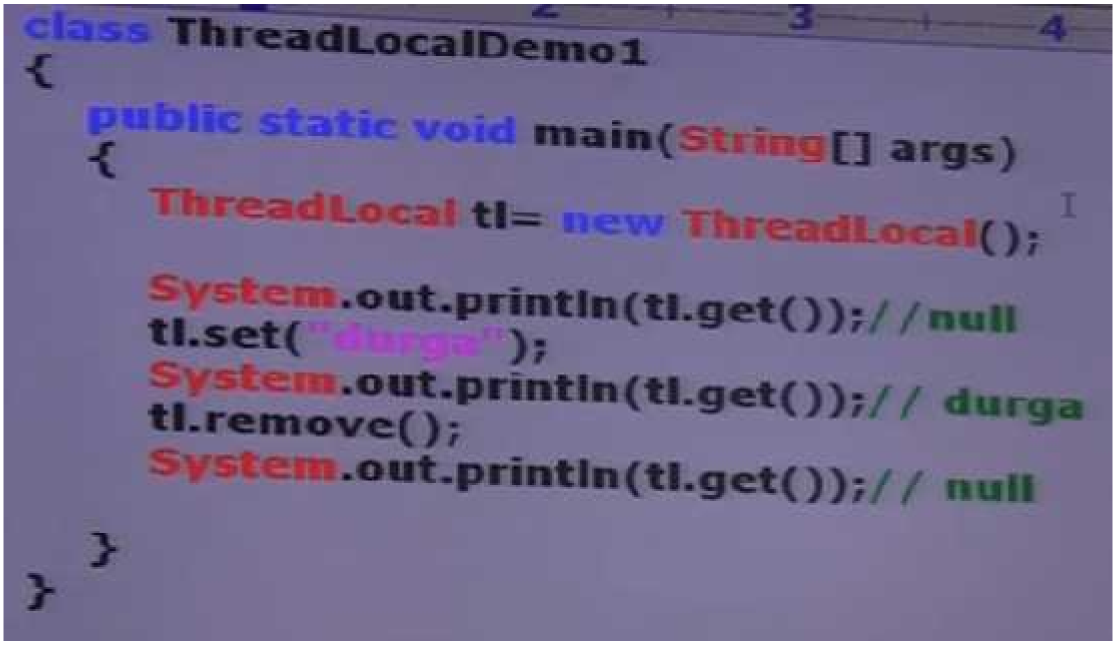
4. void remove()

To remove the value of thread local variable associated with current thread.

It is newly added method in 1.5 version

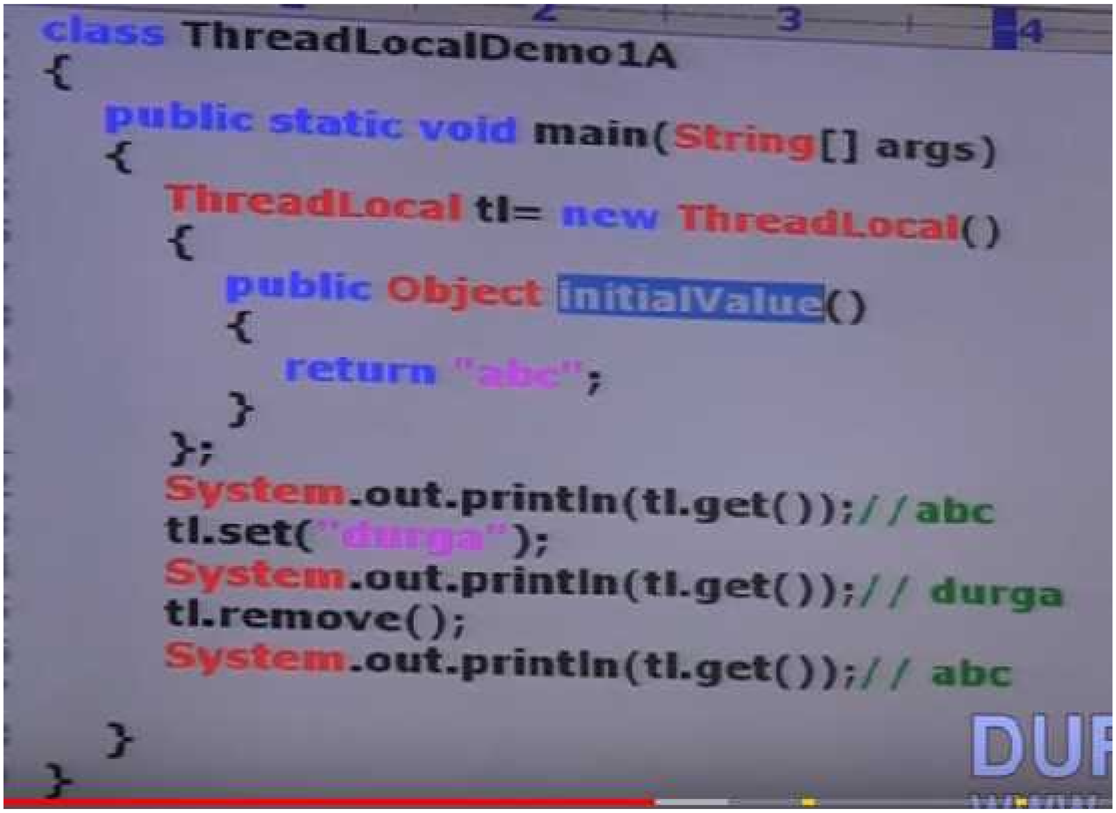
After removal if we are trying to access it will be reinitialized once again by invoking it's initial value method.

e.g. 1



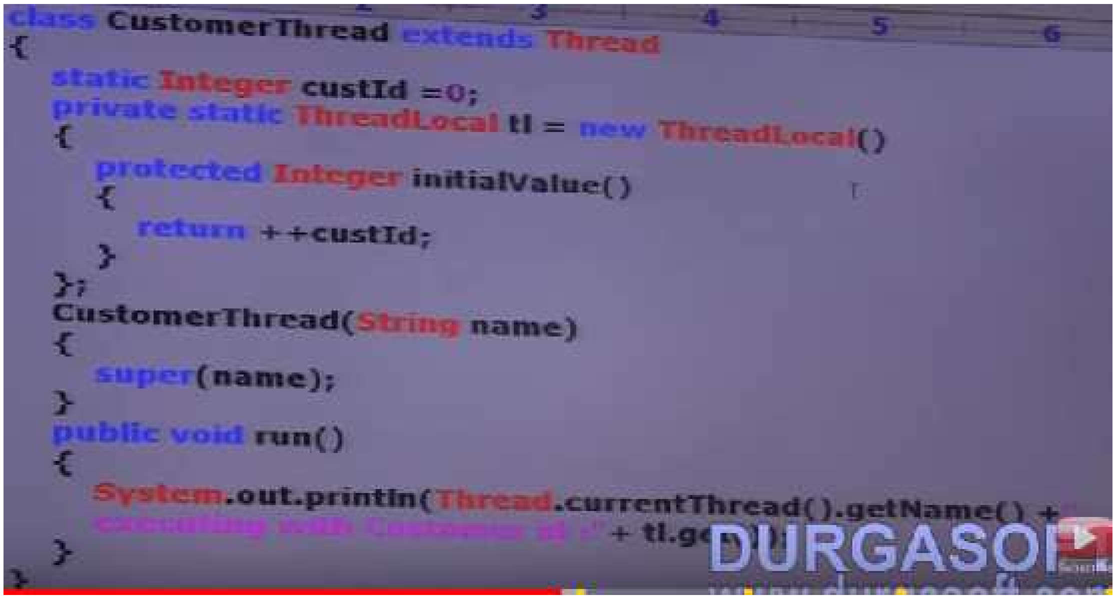
```
class ThreadLocalDemo1
{
  public static void main(String[] args)
  {
    ThreadLocal tl= new ThreadLocal();

    System.out.println(tl.get());//null
    tl.set("durga");
    System.out.println(tl.get());// durga
    tl.remove();
    System.out.println(tl.get());// null

  }
}
```

Overriding of initial value method:

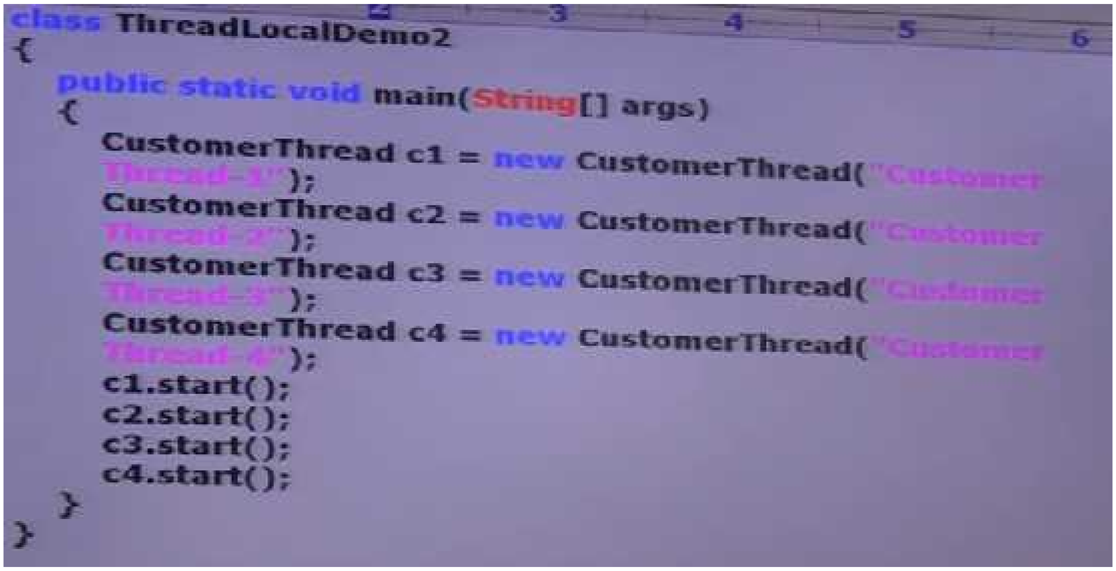


```
class ThreadLocalDemo1A
{
  public static void main(String[] args)
  {
    ThreadLocal tl= new ThreadLocal()
    {
      public Object initialValue()
      {
        return "abc";
      }
    };
    System.out.println(tl.get());//abc
    tl.set("durga");
    System.out.println(tl.get());// durga
    tl.remove();
    System.out.println(tl.get());// abc

  }
}
```

e.g. 2



```
class CustomerThread extends Thread
{
   static Integer custId =0;
   private static ThreadLocal tl = new ThreadLocal()
   {
      protected Integer initialValue()
      {
         return ++custId;
      }
   };
   CustomerThread(String name)
   {
      super(name);
   }
   public void run()
   {
      System.out.println(Thread.currentThread().getName() +"
      executing with Customer id :" + tl.ge
   }
}
```



```
class ThreadLocalDemo2
{
   public static void main(String[] args)
   {
      CustomerThread c1 = new CustomerThread("Customer
      Thread-1");
      CustomerThread c2 = new CustomerThread("Customer
      Thread-2");
      CustomerThread c3 = new CustomerThread("Customer
      Thread-3");
      CustomerThread c4 = new CustomerThread("Customer
      Thread-4");
      c1.start();
      c2.start();
      c3.start();
      c4.start();
   }
}
```

For every customer thread a separate customer id is maintained by thread local object.

ThreadLocal vs Inheritance:

Parent threads thread local value by default not availble to child thread. if we want to male parent's threads thread local variable value available to the child thread then we should go for InheritableThreadLocal class.

By default child thread value is exacyly same as parent thread's value but we can provide customized value for child thread by overriding child value method.

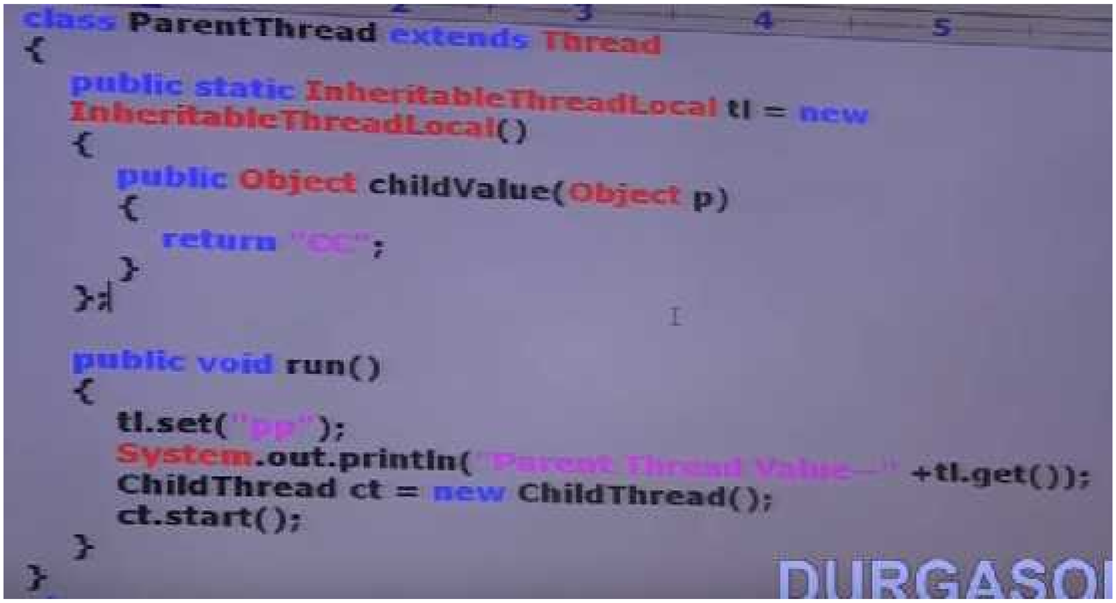
Constructor:

InheritableThreadLocal tl = new InheritableThreadLocal();

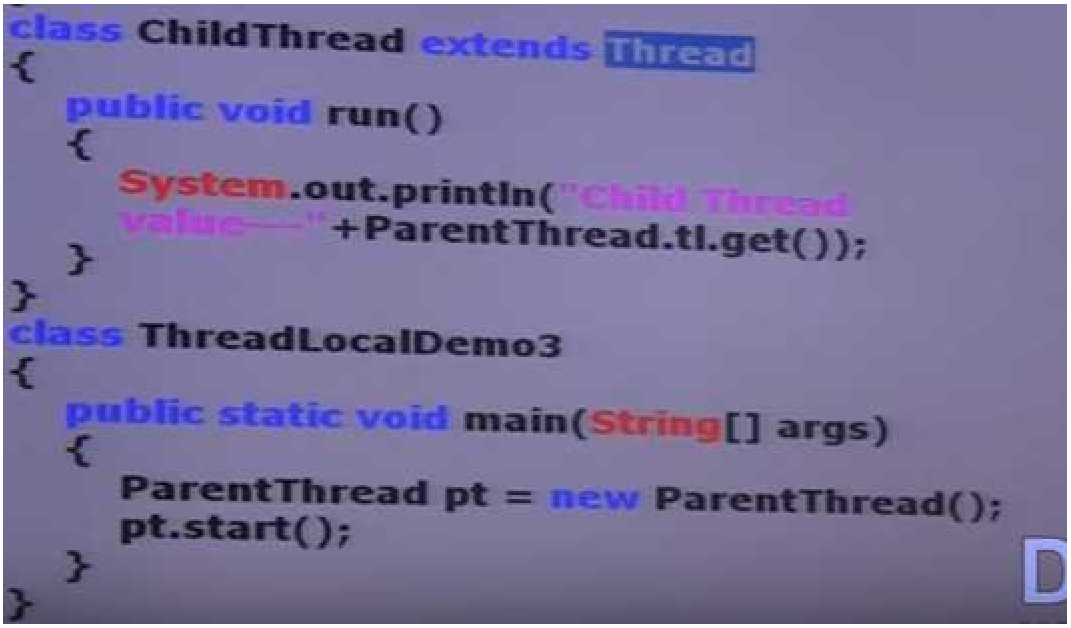
InheritablethreadLocal is the child class of thread local and hence all method's present in threadlocal by default available ti InheritableThreadLocal.

I addition these methods it contains only one method,

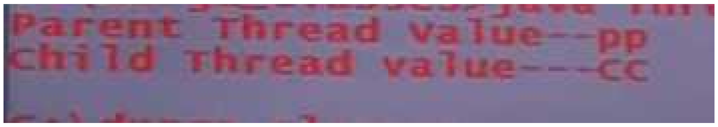
public object childValue(Object parentValue)



```
class ParentThread extends Thread
{
   public static InheritableThreadLocal tl = new
   InheritableThreadLocal()
   {
      public Object childValue(Object p)
      {
         return "cc";
      }
   };

   public void run()
   {
      tl.set("pp");
      System.out.println("Parent Thread Value---" +tl.get());
      ChildThread ct = new ChildThread();
      ct.start();
   }
}
```

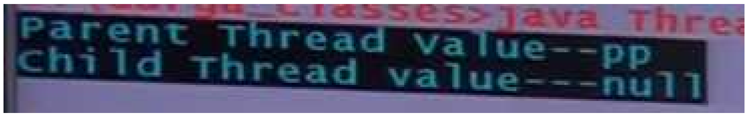


```
class ChildThread extends Thread
{
   public void run()
   {
      System.out.println("Child Thread
      value----"+ParentThread.tl.get());
   }
}
class ThreadLocalDemo3
{
   public static void main(String[] args)
   {
      ParentThread pt = new ParentThread();
      pt.start();
   }
}
```



```
Parent Thread value--pp
Child Thread value---CC
```

In the above program if we replace inheritableThreadLocal with ThreadLocal and if we are not overriding childValue method then the outout is



```
Parent Thread value--pp
Child Thread value---null
```

I