MODULE-3

PART-1

INTERFACES

Interface

Interface looks like class but it is not a class. Using interface we can fully abstract a class interface from its implementation. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default. Once it is defined a number of classes can implement an interface .Also, one class can implement any number of interfaces.

To implement a interface a class must create complete set of methods defined by the interface.

What is the use of interfaces?

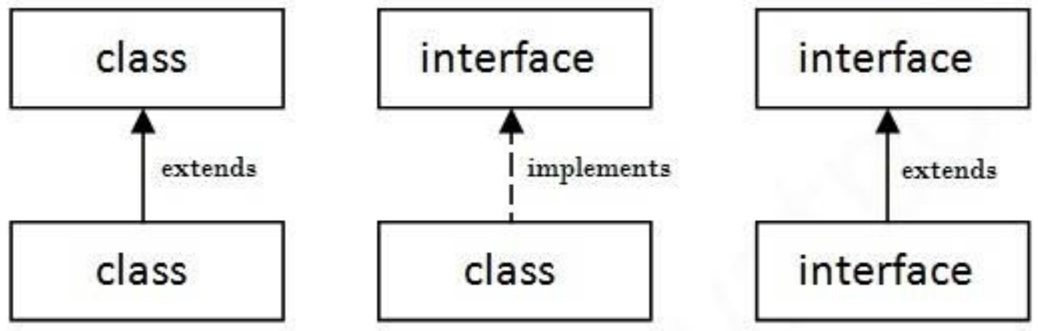
As mentioned above they are used for abstraction. Also, java programming language does not support multiple inheritances, using interfaces we can achieve this as a class can implement more than one interfaces, however it cannot extend more than one classes.

Declaration

Interfaces are declared by specifying a keyword "interface".

```
interface interfacename
{
 variable declaration;
 Methods declaration;
}
```

Relationship between classes and interfaces.



Example 1: Simple interface implementation

```
interface MyInterface
{
  public void method1();
 }
class interfacedemo implements MyInterface
{
 public void method1()
 {
     System.out.println("implementation of method1");
 }
  public static void main(String arg[])
 {
     MyInterface obj = new interfacedemo();
     obj. method1();
  }
}
```

Output:

```
implementation of method1
```

Example 2: An interface cannot implement another interface. It has to extend the other interface if required.

```
 interface Myinterface1
{
 public void method1();
 }
 interface Myinterface2 extends Myinterface1
{
   public void method2();

}
class interfacedemo2 implements Myinterface2
{
        public void method1()
        {
            System.out.println("implementation of method1");
        }
        public void method2()
        {
            System.out.println("implementation of method2");
        }
  public static void main(String arg[])
     {
      Interfacedemo2 obj = new interfacedemo2();
      obj. method1();
      obj. method1();

  }
}
```

Output:

```
implementation of method1
implementation of method2
```

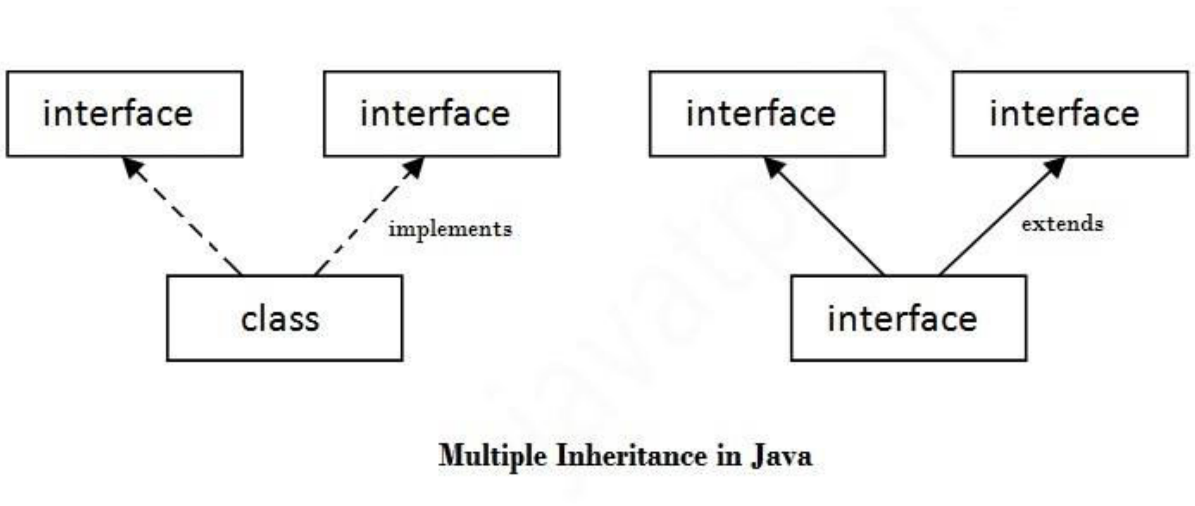
Example 3: An interface can be implemented by any number of classes.

```
interface area
 {
   final static float pi=3.14f;
   float compute(float x,float y);
 }
class rectangle implements area
{
  public float compute(float x,float y)
   {
     return(x*y);
    }
}
class circle implements area
{
  public float compute(float x,float y)
   {
     return(pi*x*x);
    }
}
class interfacetest
{
  public static void main(String args[])
   {
     rectangle rect=new rectangle();
     System.out.println("area of rect="+rect.compute(10,10));
     circle cir=new circle();
     System.out.println("area of circle="+cir.compute(10,0));

   }
}
```

Output:

```
area of rect=100.0
area of circle=314.0
```

Multiple inheritance in Java by interface:



Example 4: Interface can extend many interfaces separated by commas.

```
 interface Myinterface1
{
 public void method1();
}
 interface Myinterface2
{
   public void method2();
}
 interface Myinterface3 extends Myinterface1, Myinterface2
{
   public void method3();
}

class interfacedemo3 implements Myinterface3
{
  public void method1()
  {
      System.out.println("implementation of method1");
  }
  public void method2()
  {
      System.out.println("implementation of method2");
  }
public void method3()
  {
    System.out.println("implementation of method3");
 }
 public static void main(String arg[])
 {
    Interfacedemo3 obj = new interfacedemo3();
    obj. method1();
    obj. method2();
     obj. method3();
 }
}
```

Output:

```
implementation of method1
implementation of method2
implementation of method3
```

Example 5: A Class can implement two interfaces separated by commas.

```
interface Myinterface1
{
 public void method1();

}
 interface Myinterface2
{
  public void method2();
}

class interfacedemonotp implements Myinterface1,Myinterface2
{
 public void method1()
 {
    System.out.println("implementation of method1");
 }
 public void method2()
 {
    System.out.println("implementation of method2");
 }
 public static void main(String arg[])
 {
    interfacedemonotp obj = new interfacedemonotp();
    obj. method1();
    obj. method2();

 }
}
```

Output:

```
implementation of method1
implementation of method2
```

NOTE: If declared in public need to be saved in different files

```
public  interface Myinterface1   //save with the file name Myinterface1.java
{
 public void method1();
}
public interface Myinterface2 //save with the file name Myinterface2.java
{
   public void method2();
 }

//save with the file name Myinterface3.java
public interface Myinterface3 extends Myinterface1, Myinterface2
{
   public void method3();

}
//save with the file name interfacedemo3.java
public class interfacedemo3 implements Myinterface3
{
  public void method1()
  {
     System.out.println("implementation of method1");
  }
  public void method2()
  {
     System.out.println("implementation of method2");
  }
public void method3()
  {
     System.out.println("implementation of method3");
  }
  public static void main(String arg[])
  {
     Interfacedemo3 obj = new interfacedemo3();
     obj. method1();
     obj. method2();
      obj. method3();
  }}
```

PART-2

PACKAGES

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
Output: Hello
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java


package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;


class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
Output: Hello
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

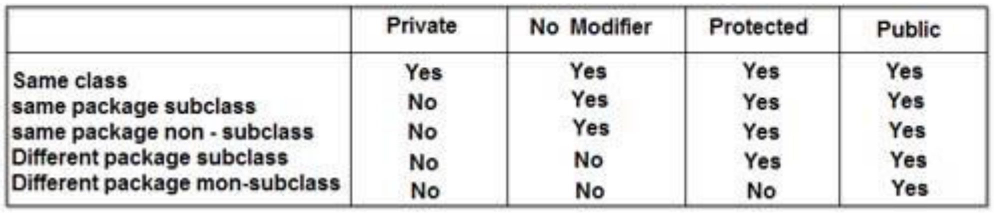
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output: Hello

Access modifiers:

Access modifiers define the scope of the class and its members (data and methods). For example, private members are accessible within the same class members (methods). Java provides many levels of security that provides the visibility of members (variables and methods) within the classes, subclasses, and packages. Packages are meant for encapsulating, it works as containers for classes and other subpackages. Class acts as containers for data and methods. There are four categories, provided by Java regarding the visibility of the class members between classes and packages:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
same package subclass	No	Yes	Yes	Yes
same package non - subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package mon-subclass	No	No	No	Yes

Example:

```
package pkg1;

public class Protection
{
        int n = 1;
        private int n_priv = 2;
        protected int n_prot = 3;
        public int n_publ = 4;

        public Protection()
        {
                System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_priv = " + n_priv);
                System.out.println("n_prot = " + n_prot);
                System.out.println("n_publ = " + n_publ);
        }
}
```

This is **Derived.java** file:

```
package pkg1;

class Derived extends Protection
{
        Derived()
        {
                System.out.println("derived constructor");
                System.out.println("n = " + n);

                System.out.println("n_prot = " + n_prot);
                System.out.println("n_publ = " +n_publ);
        }
}
```

This is **SamePackage.java** file:

```
package pkg1;

class SamePackage
{
        SamePackage()
```

```
	{
		Protection pro = new Protection();
		System.out.println("same package constructor");
		System.out.println("n = " + pro.n);

		/* class only
		*  System.out.println("n_priv = " + pro.n_priv); */

		System.out.println("n_prot = " + pro.n_prot);
		System.out.println("n_publ = " + pro.n_publ);
	}
}
package pkg2;

class Protection2 extends pkg1.Protection
{
	Protection2()
	{
		System.out.println("derived other package constructor");

		/* class or package only
		*  System.out.println("n = " + n); */

		/* class only
		*  System.out.println("n_priv = " + n_priv); */

		System.out.println("n_prot = " + n_prot);
		System.out.println("n_publ = " + n_publ);
	}
}
```

This is **OtherPackage.java** file:

```
package pkg2;

class OtherPackage
{
	OtherPackage()
	{
		pkg1.Protection pro = new pkg1.Protection();

		System.out.println("other package constructor");

		/* class or package only
		*  System.out.println("n = " + pro.n); */
```

```
                /* class only
                *  System.out.println("n_priv = " + pro.n_priv); */

                /* class, subclass or package only
                *  System.out.println("n_prot = " + pro.n_prot); */

                System.out.println("n_publ = " + pro.n_publ);
        }
}
```

The test file for pkg1

```
/* demo package pkg1 */

package pkg1;

/* instantiate the various classes in pkg1 */
public class Demo
{
        public static void main(String args[])
        {
                Protection obj1 = new Protection();
                Derived obj2 = new Derived();
                SamePackage obj3 = new SamePackage();
        }
}
```

The test file for **pkg2** is shown below :

```
/* demo package pkg2 */

package pkg2;

/* instantiate the various classes in pkg2 */
public class Demo
{
        public static void main(String args[])
        {
                Protection2 obj1 = new Protection2();
                OtherPackage obj2 = new OtherPackage();
        }
}
```

PART-3

MULTITHREADING

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantages of Multithreading:

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

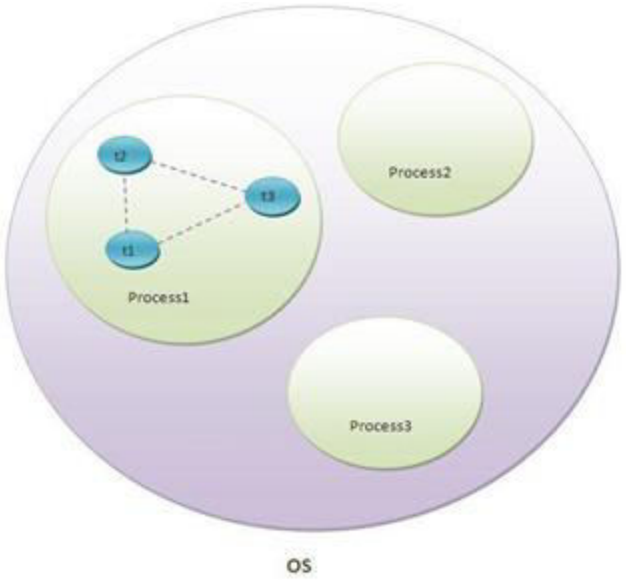
3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

Difference between Multithreading and Multitasking

Multithreading	Multitasking
It is a programming concept in which a program or process is divided into two or more sub programs.	It is an operating system concept in which multiple tasks are performed simultaneously.
It supports execution of multiple parts of a single program simultaneously.	It supports execution of multiple programs simultaneously.
The processor has to switch between different parts of thread or program	The processor has to switch between different programs
It is highly efficient	It is less efficient compared to multithreading
A thread is the smallest unit in multithreading	A program is smallest unit
It helps in developing efficient programs	It helps in developing efficient operating systems.



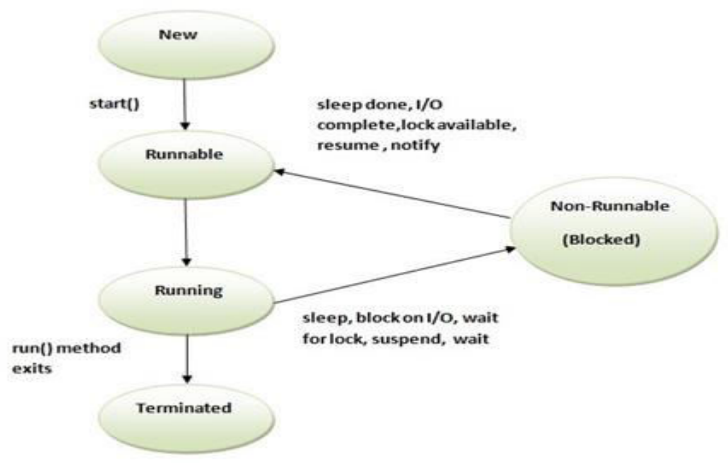
Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated


New
start()
Runnable
sleep done, I/O complete, lock available, resume , notify
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

1) New

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

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

1. By extending Thread class

2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Java Thread Example by extending Thread class

```
class Multi extends Thread{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

 }

}
```

OUTPUT: Thread is running…

By Implementing Runnable Interface

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

OUTPUT: Thread is running…

Example to create three threads

```
class Thread_A extends Thread{
public void run(){
for(int k=0;k<100;k++)
System.out.println("thread A is running...");
}
class Thread_B extends Thread{
public void run(){
for(int j=0;j<20;j++)
System.out.println("thread Bis running...");
}
class Thread_C extends Thread{
public void run(){
for(inti=0;i<10;i++)
System.out.println("thread Cis running...");
}
classthreaddemo
{
```

```
public static void main(String ae[])
{
Thread_A a=new Thread_A();
Thread_B b=new Thread_B();
Thread_C c=new Thread_C();
a.start();
b.start();
c.start();
}
}
```

1. **Sleep():** method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
   Example:
   ```
   import java.lang.*;
   public class ThreadDemo implements Runnable {
     public void run() {
       for (inti = 10; i< 13; i++) {
   System.out.println(Thread.currentThread().getName() + "  " + i);
       try {
       // thread to sleep for 1000 milliseconds
   Thread.sleep(1000);
       } catch (Exception e) {
   System.out.println(e);
       }
       }
       }

       public static void main(String[] args) throws Exception {
       Thread t = new Thread(new ThreadDemo());
       // this will call run() function
   t.start();
       Thread t2 = new Thread(new ThreadDemo());
   ```

```
   // this will call run() function
   t2.start();
   }
}
```

Expected Output:

```
Thread-0  10
Thread-1  10
Thread-0  11
Thread-1  11
Thread-0  12
Thread-1  12
```

Other Thread Methods:

public void suspend()

This method puts a thread in the suspended state and can be resumed using resume() method.

public void stop()

This method stops a thread completely.

public void resume()

This method resumes a thread, which was suspended using suspend() method.

public void wait()

Causes the current thread to wait until another thread invokes the notify().

public void notify()

Wakes up a single thread that is waiting on this object's monitor.

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defiend in Thread class:

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY

- public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY).

The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example:

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
    }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
    }
}
```

```
Output:running thread name is:Thread-0
       running thread priority is:10
       running thread name is:Thread-1
       running thread priority is:1
```

Synchronization

If multiple threads are simultaneously trying to access the same resource strange results may occur. To overcome them java synchronization is used. The operations performed on the resource must be synchronized.

Monitor is the key to synchronization. A *monitor* is an object that is used as a mutually exclusive lock.Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor (other threads are waiting at that time) .

Code can be synchronized in two ways:

1. Using synchronized Methods
2. Using synchronized Statement

1. Using synchronized Methods :

```
synchronized void update()

{

        -    - - -

}
```

When as method is declared as synchronized, java creates a monitor and hands it over to the thread that calls the method first time. As long as the thread holds the monitor no other thread can enter the synchronized section of code.

Example:

```
class Table
{
 synchronized void printTable(int n)
{//synchronized method
   for(int i=1;i<=5;i++)
{
     System.out.println(n*i);
     try
    {
      Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }
 }
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
       10
       15
       20
       25
       100
       200
       300
       400
       500
```

2. Using synchronized Statement:

   This is the general form of the **synchronized** statement:

   ```
   synchronized(objRef)
   {
   // statements to be synchronized
   }
   ```

   objRef is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a member of *objRef*'s class occurs only after the current thread has successfully entered *objRef*'s monitor.

```
class Table{
 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}
```

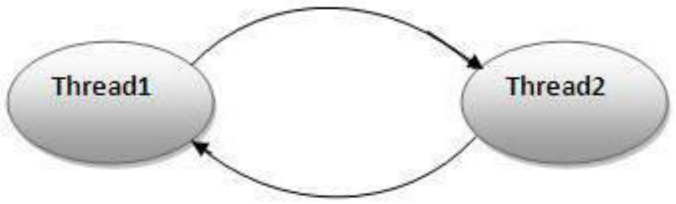
```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
     10
     15
     20
     25
     100
     200
```

```
300
400
500
```

Deadlocks

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example:

```
public class ThreadDeadlock {
   public static void main(String[] args) throws InterruptedException {
      Object obj1 = new Object();
      Object obj2 = new Object();
      Object obj3 = new Object();

      Thread t1 = new Thread(new SyncThread(obj1, obj2), "t1");
      Thread t2 = new Thread(new SyncThread(obj2, obj3), "t2");
      Thread t3 = new Thread(new SyncThread(obj3, obj1), "t3");

      t1.start();
      Thread.sleep(5000);
      t2.start();
      Thread.sleep(5000);
      t3.start();
   }
}
```

```
class SyncThread implements Runnable{
    private Object obj1;
    private Object obj2;
    public SyncThread(Object o1, Object o2){
        this.obj1=o1;
        this.obj2=o2;
    }
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name + " acquiring lock on "+obj1);
        synchronized (obj1) {
         System.out.println(name + " acquired lock on "+obj1);
         work();
         System.out.println(name + " acquiring lock on "+obj2);
         synchronized (obj2) {
            System.out.println(name + " acquired lock on "+obj2);
            work();
        }
         System.out.println(name + " released lock on "+obj2);
        }
        System.out.println(name + " released lock on "+obj1);
        System.out.println(name + " finished execution.");
    }
    private void work() {
        try {
            Thread.sleep(30000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```