UNIT-4

Prepared By:
Deepak Kumar Sharma
Asst. Professor
SoCS UPES Dehradun

UNIT-IV

Static Nested Types, Inner Classes, Local Inner Classes, Anonymous Inner Classes
Inheriting Nested Types, Nesting in Interfaces, Implementation of Nested Types
Creating Threads, Using Runnable, Synchronization
Wait, notify, notifyall, Waiting and Notification
Thread Scheduling, Deadlocks, Ending Thread Execution, volatile
Thread Management, Security, and ThreadGroup, Threads and Exceptions, debugging threads

Java Nested and Inner Class

- Nested Class: Defining a class within another class.

```
class OuterClass {
    // ...
    class NestedClass {
        // ...
    }
}
```

Two types of nested classes can be created in Java:

- Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class
- Static nested class

Non-Static Nested Class (Inner Class)

- Non-static nested classes are known as inner classes.
- It has access to members of the enclosing class (outer class).
- *Must instantiate the outer class first, in order to instantiate the inner class.*

Need of inner class:

- Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

Advantage of Java inner classes

- Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
- Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
- **Code Optimization**: It requires less code to write.

Java Member Inner class

- A non-static class that is created inside a class but outside a method is called **member inner class**.
- It is also known as a **regular inner class**.
- It can be declared with access modifiers like public, default, private, and protected.

Syntax:

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
//to create an object of the member inner class
//OuterClassReference.new MemberInnerClassConstructor();
TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Output: data is 30

Java Anonymous inner class

- Java anonymous inner class is an inner class without a name and for which only a single object is created.
- An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, *without having to actually subclass a class*.
- Java Anonymous inner class can be created in two ways:
  - Class (may be abstract or concrete).
  - Interface

Example- Java Anonymous inner class

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

Output: nice fruits

Internal Working:

- A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
- An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Output: nice fruits

Java Local inner class

- A class i.e., created inside a method, is called local inner class in java.
- Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body.
- Sometimes this block can be a for loop, or an if clause.
- Local Inner classes are not a member of any enclosing classes.
- They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them.
- However, they can be marked as final or abstract.
- These classes have access to the fields of the class enclosing it.
- If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Java local inner class example

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

output:
30

Example- Inner Class

```
class CPU {
    double price;
    // nested class
    class Processor{
        // members of nested class
        double cores;
        String manufacturer;
        double getCache(){
            return 4.3;
        }
    }
    // nested protected class
    protected class RAM{
        // members of protected nested class
        double memory;
        String manufacturer;
        double getClockSpeed(){
            return 5.5;
        }
    }
}
```

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

        // create object of Outer class CPU
        CPU cpu = new CPU();

       // create an object of inner class Processor using outer class
        CPU.Processor processor = cpu.new Processor();

        // create an object of inner class RAM using outer class CPU
        CPU.RAM ram = cpu.new RAM();
        System.out.println("Processor Cache = " + processor.getCache());
        System.out.println("Ram Clock speed = " + ram.getClockSpeed());
    }
}
```

Output:
Processor Cache = 4.3
Ram Clock speed = 5.5

Accessing Members of Outer Class within Inner Class

```
class Car {
    String carName;
    String carType;
    // assign values using constructor
    public Car(String name, String type) {
        this.carName = name;
        this.carType = type;
    }
    // private method
    private String getCarName() {
        return this.carName;
    }
// inner class
    class Engine {
        String engineType;
        void setEngine() {
          // Accessing the carType property of Car
           if(Car.this.carType.equals("4WD")){
              // Invoking method getCarName() of Car
              if(Car.this.getCarName().equals("Crysler")) {
                 this.engineType = "Smaller";
              } else {
                 this.engineType = "Bigger";
              }
           }else{
              this.engineType = "Bigger";
           }
        }
        String getEngineType(){
           return this.engineType;
        }
    }
}
```

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

// create an object of the outer class Car
        Car car1 = new Car("Mazda", "8WD");
        // create an object of inner class using the outer class
        Car.Engine engine = car1.new Engine();
        engine.setEngine();
        System.out.println("Engine Type for 8WD= " + engine.getEngineType());

        Car car2 = new Car("Crysler", "4WD");
        Car.Engine c2engine = car2.new Engine();
        c2engine.setEngine();
        System.out.println("Engine Type for 4WD = " + c2engine.getEngineType());
    }
}
```

- can access the members of the outer class by using "**this**" keyword.

Static Nested Class

- A static class inside another class.
- Static nested classes are not called static inner classes.
- Unlike inner class, a static nested class cannot access the member variables of the outer class. It is because the **static nested class** doesn't require you to create an instance of the outer class.
- It cannot access non-static data members and methods.
- It can access static data members of the outer class, including private.
- Static nested classes can include both static and non-static fields and methods.
- To access the static nested class, we don't need objects of the outer class.

Note: In Java, only nested classes are allowed to be static.

Static Nested class

```
class TestOuter1{
  static int data=30;

   static class Inner{
   void msg(){System.out.println("data is "+data);}
  }

public static void main(String args[]){

TestOuter1.Inner obj=new TestOuter1.Inner();

  obj.msg();
  }
}
```

Output:

data is 30

Static Nested Class

```
class Animal {

// inner class
   class Reptile {
      public void displayInfo() {
        System.out.println("I am a reptile.");
      }
   }

// static class
   static class Mammal {
      public void displayInfo() {
        System.out.println("I am a mammal.");
      }
   }
}
```

```
class Main {
   public static void main(String[] args) {
      // object creation of the outer class
      Animal animal = new Animal();

      // object creation of the non-static class
      Animal.Reptile reptile = animal.new Reptile();
      reptile.displayInfo();

      // object creation of the static nested class
      Animal.Mammal mammal = new Animal.Mammal();
      mammal.displayInfo();

   }
}
```

Output

I am a reptile.
I am a mammal.

Java static nested class example with a **static method**

```
public class TestOuter2{
  static int data=30;
  static class Inner{
   static void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter2.Inner.msg();//no need to create the instance of static nested class

  }
}
```

Accessing Non-static members

```
class Animal {
  static class Mammal {
    public void displayInfo() {
      System.out.println("I am a mammal.");
    }
  }

  class Reptile {
    public void displayInfo() {
      System.out.println("I am a reptile.");
    }
  }

  public void eat() {
    System.out.println("I eat food.");
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Animal animal = new Animal();
    Animal.Reptile reptile = animal.new Reptile();
    reptile.displayInfo();

    Animal.Mammal mammal = new Animal.Mammal();
    mammal.displayInfo();
    mammal.eat();
  }
}
```

OUTPUT:

```
Main.java:28: error: cannot find symbol
    mammal.eat();
           ^
  symbol:   method eat()
  location: variable mammal of type Mammal
1 error
compiler exit status 1
```

Note: static nested classes can only access the class members (static fields and methods) of the outer class.

CheckPoint?

```
static class Animal {
 public static void displayInfo() {
   System.out.println("I am an animal");
 }
}

class Main {
 public static void main(String[] args) {
   Animal.displayInfo();
 }
}
```

Output

```
Main.java:1: error: modifier static not allowed here
static class Animal {
       ^
1 error
compiler exit status 1
```

Java Nested Interface

- An interface, i.e., declared within another interface or class, is known as a nested interface.
- The nested interfaces are used to group related interfaces so that they can be easy to maintain.
- The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

Syntax of nested interface

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

← within the interface

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

← within the class

Example of nested interface which is declared within the interface

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
```

Internally →

```
public static interface Showable$Message
{
  public abstract void msg();
}
```

```
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

Output:

hello nested interface

Example of nested interface which is declared within the class

```
class A{
  interface Message{
   void msg();
  }
}

class TestNestedInterface2 implements A.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  A.Message message=new TestNestedInterface2();//upcasting here
  message.msg();
 }
}
```

Output:

hello nested interface

Can we define a class inside the interface?

```
interface M{
  class A{}
}
```

- Yes, if we define a class inside the interface, the Java compiler creates a **static nested class.**

Inheriting Nested Types

- One inner class can extend another inner class of the same class.

```
class OuterClass
{
   class InnerClassOne
   {
      int x = 10;
      void methodOfInnerClassOne()
      {
         System.out.println("From InnerClassOne");
      }
   }

   class InnerClassTwo extends InnerClassOne
   {
      //One Inner Class can extend another inner class
   }
}
```

```
public class InnerClasses
{
   public static void main(String args[])
   {
      OuterClass outer = new
OuterClass();   //Instantiating OuterClass

      OuterClass.InnerClassTwo innerTwo =
outer.new InnerClassTwo();  //Instantiating
InnerClassTwo

      System.out.println(innerTwo.x);    //Accessing
inherited field x from InnerClassOne
      innerTwo.methodOfInnerClassOne();  //calling
inherited method from InnerClassOne
   }
}
```

Inheriting Nested Types

An inner class can be extended by another class outside of it's outer class.

- If you are extending static inner class (Static nested class), then it is a straight forward implementation.

- If you are extending non-static inner class, then sub class constructor must explicitly call super class constructor using an instance of outer class. Because, you can't access non-static inner class without the instance of outer class.

Example

```
class OuterClass
{
   static class InnerClassOne
   {
      //Class as a static member
   }
   class InnerClassTwo
   {
      //Class as a non-static member
   }
}

//Extending Static inner class or static nested class
class AnotherClassOne extends OuterClass.InnerClassOne
{
   //static nested class can be referred by outer class name,
}

//Extending non-static inner class or member inner class
class AnotherClassTwo extends OuterClass.InnerClassTwo
{
   public AnotherClassTwo()
   {
      new OuterClass().super();  //accessing super class constructor through OuterClass instance
   }
}
```

Example

```
class OuterClass
{
    int x;
    void methodOfOuterClass() {
        System.out.println("From OuterClass"); }
    //Class as a member
    class InnerClass {
        int y; }
}
class AnotherClass extends OuterClass {
     //Only fields and methods are inherited. To use inner class properties,
    //it's inner class must extend inner class of it's super class
    class AnotherInnerClass extends InnerClass {
        //Inner Class of AnotherClass extends Inner Class of OuterClass  }
}
public class InnerClasses
{
    public static void main(String args[])
    {
        AnotherClass anotherClass = new AnotherClass();  //creating AnotherClass Object
        System.out.println(anotherClass.x);    //accessing inherited field x from OuterClass
        anotherClass.methodOfOuterClass();    //calling inherited method from OuterClass

        //Using the properties of InnerClass
        AnotherClass.AnotherInnerClass anotherInnerClass = anotherClass.new AnotherInnerClass();
        //creating object to AnotherInnerClass
        System.out.println(anotherInnerClass.y);  //accessing inherited field y from InnerClass
    }
}
```

Multithreading in Java

Deepak Sharma, Asst. Professor UPES Dehradun

Multithreading

- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread.
- Threads are light-weight processes (the smallest unit of processing) within a process.
- To achieve multitasking: We use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Multithreading is mostly used in games, animation, etc.
- Threads can be created by using two mechanisms :
  - Extending the Thread class
  - Implementing the Runnable Interface

Advantages of Java Multithreading

- It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- You **can perform many operations together, so it saves time**.
- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously.

1) **Process-based Multitasking (Multiprocessing)**

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

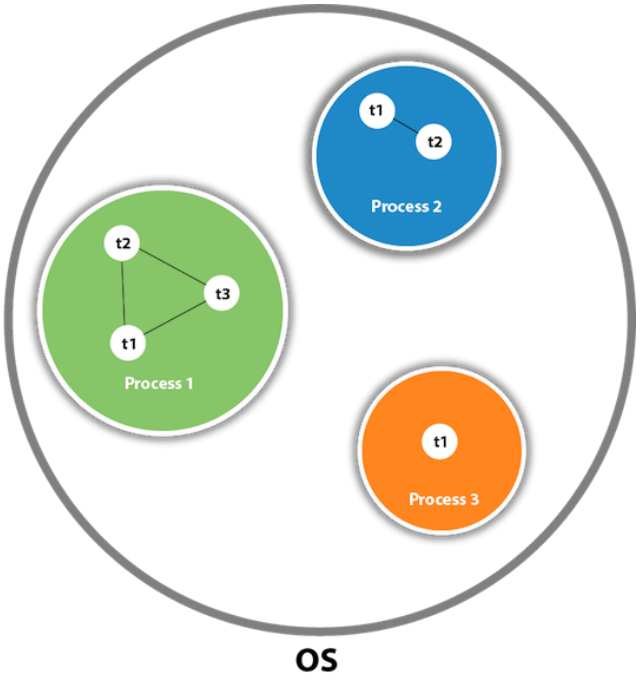
2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Thread in java

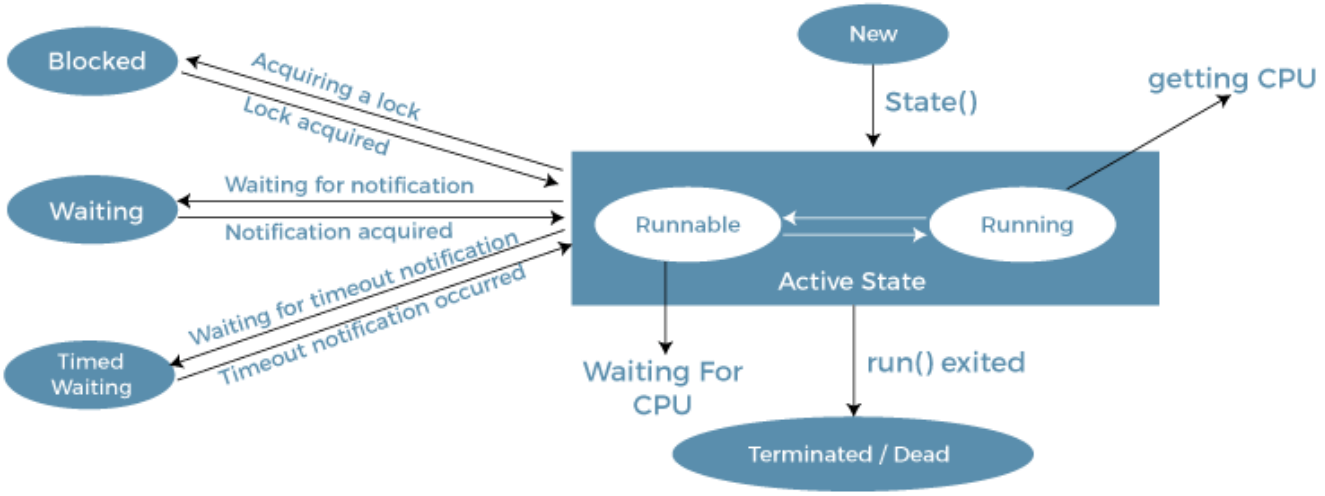
- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

- A thread is executed inside the process.
- There is context-switching between the threads.
- There can be multiple processes inside the OS, and one process can have multiple threads.



Note: At a time one thread is executed only.

Life cycle of a Thread



Life Cycle of a Thread

1. **New:** A new thread is created. The code has not been run yet and thus has not begun its execution.
2. **Active:** When a thread invokes the start() method
    - 2.1 **Runnable:** Ready to run.Waiting for thread scheduler to provide the thread time to run
    - 2.2 **Running:** When the thread gets the CPU
3. **Blocked / Waiting:** a thread is inactive for a span of time (not permanently) (Waiting for resource such as printer)
4. **Timed Waiting:** To avoid starvation. Thread lies in the waiting state for a specific span of time, and not forever. (e.g. sleep)
5. **Terminated:** thread has finished its job or due to unhandled exception.

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)

Creating Threads

Threads can be created by using two mechanisms :

- Extending the Thread class
- Implementing the Runnable Interface

Commonly used methods of Thread class:

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.
- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.

Commonly used methods of Thread class:

- **public boolean isAlive():** tests if the thread is alive.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).
- **public boolean isDaemon():** tests if the thread is a daemon thread.
- **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
- **public void interrupt():** interrupts the thread.
- **public boolean isInterrupted():** tests if the thread has been interrupted.
- **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().
- **public void run():** is used to perform action for a thread.

Starting a thread:

- The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:
  - A new thread starts(with new callstack).
  - The thread moves from New state to the Runnable state.
  - When the thread gets a chance to execute, its target run() method will run.

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...

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);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

Output:

thread is running...

- If you are not extending the Thread class, class object would not be treated as a thread object.
- So you need to explicitly create the Thread class object.
- We are passing the object of class that implements Runnable so that class run() method may execute.

Using the Thread Class: Thread(String Name)

```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

Output:

My first thread

Using the Thread Class: Thread(Runnable r, String name)

```
public class MyThread2 implements Runnable
{
public void run()
{
System.out.println("Now the thread is running ...");
}

public static void main(String argvs[])
{
// creating an object of the class MyThread2
Runnable r1 = new MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");

// the start() method moves the thread to the active state
th1.start();

// getting the thread name by invoking the getName() method
String str = th1.getName();
System.out.println(str);
}
}
```

Output:

My new thread
Now the thread is running ...

Example: by extending the Thread class

```
class MultithreadingDemo extends Thread {
  public void run()
  {
    try {
      // Displaying the thread that is running
      System.out.println("Thread " + Thread.currentThread().getId() + " is
running");
    }
    catch (Exception e) {
      // Throwing an exception
      System.out.println("Exception is caught");
    }
  }
}
// Main Class
public class Multithread {
  public static void main(String[] args)
  {
    int n = 8; // Number of threads
    for (int i = 0; i < n; i++) {
      MultithreadingDemo object= new MultithreadingDemo();
      object.start();
    }
  }
}
```

Output

```
Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running
```

Thread creation by implementing the Runnable Interface

```
class MultithreadingDemo implements Runnable {
  public void run()
  {
    try {
      // Displaying the thread that is running
      System.out.println( "Thread " + Thread.currentThread().getId() + " is
running");
    }
    catch (Exception e) {
      // Throwing an exception
      System.out.println("Exception is caught");
    }
  }
}
// Main Class
class Multithread {
  public static void main(String[] args)
  {
    int n = 8; // Number of threads
    for (int i = 0; i < n; i++) {
      Thread object = new Thread(new MultithreadingDemo());
      object.start();
    }
  }
}
```

Output

```
Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running
```

Example- Thread creation by implementing the Runnable Interface

```
class RunnableDemo implements Runnable {
   public void run() {
      System.out.println("Running " +  Thread.currentThread().getId());
        for(int i = 4; i > 0; i--) {
          System.out.println("Thread " + Thread.currentThread().getId() + ": " + i);
            }
      System.out.println("Thread " +  Thread.currentThread().getId() + " exiting.");
   }
}

public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1,"Thread 1");
      t1.start();
      Thread t2=new Thread(R1,"Thread 2");
      t2.start();

   }
}
```

Check this 

Output 1:

```
Running 21
Running 22
Thread 21: 4
Thread 22: 4
Thread 21: 3
Thread 21: 2
Thread 21: 1
Thread 22: 3
Thread 21 exiting.
Thread 22: 2
Thread 22: 1
Thread 22 exiting.
```

Output 2:

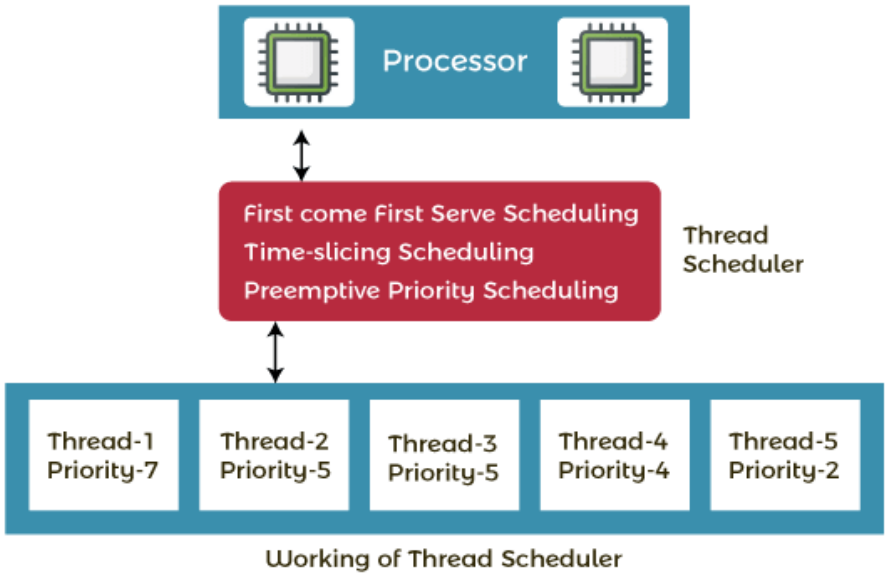
```
Running 21
Running 22
Thread 21: 4
Thread 21: 3
Thread 21: 2
Thread 21: 1Thread 21 exiting
Thread 22: 4Thread 22: 3
Thread 22: 2
Thread 22: 1
Thread 22 exiting.
```

Thread Scheduler in Java

- A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**.
- In case of multiple threads in runnable state, there are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.
- **Priority:** Priority of each thread lies between 1 (minimum) to 10 (maximum). If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.
- **Time of Arrival:** Suppose two threads of the same priority enter the runnable state, then **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread Scheduler

- In case of multiple threads in runnable state, thread scheduler decides which thread will get the CPU first.
- Selects the thread that has the highest priority.
- If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.
- When two threads with same priority than the thread that arrives first gets the opportunity to execute first.



Working of Thread Scheduler

Thread Scheduler Algorithms

First Come First Serve Scheduling (Non Preemptive)

- In this scheduling algorithm, the scheduler picks the threads that arrive first in the runnable queue.

Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3

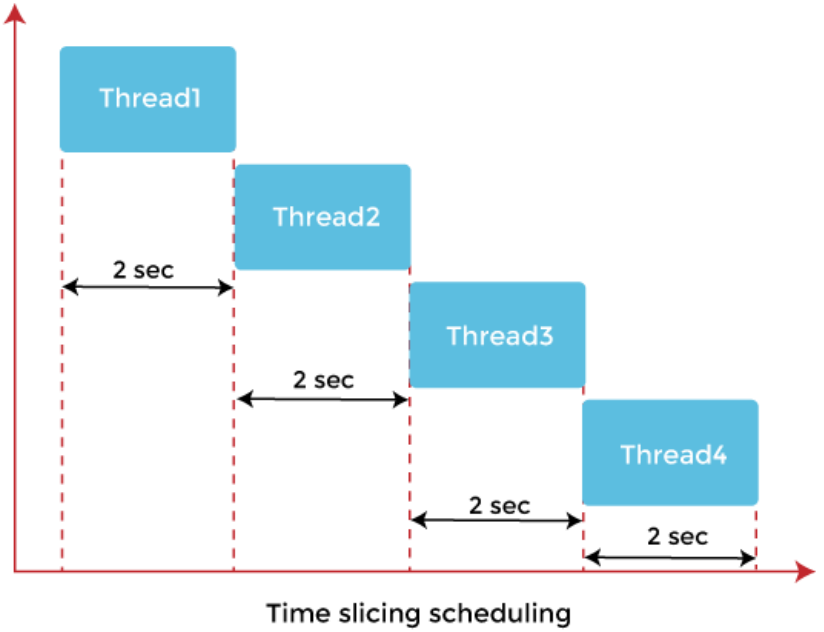
First Come First Serve Scheduling

Thread Scheduler Algorithms

Time-slicing scheduling

- FCFS algorithm is non-preemptive=> may cause starvation
- time-slices are provided to the threads so that after some time, the running thread has to give up the CPU.
- Thus, the other waiting threads also get time to run their job.

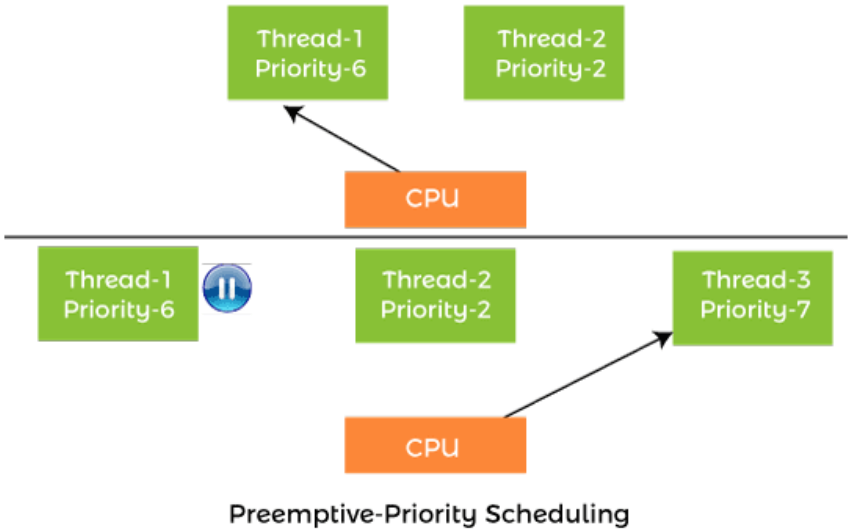
Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3



Thread Scheduler Algorithms

Preemptive-Priority Scheduling:

- The thread scheduler picks that thread that has the highest priority.
- Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation.



Preemptive-Priority Scheduling

Can we start a thread twice?

```
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

Output:

```
running
Exception in thread "main"
java.lang.IllegalThreadStateException
```

- No. After starting a thread, it can never be started again.
- If you does so, an *IllegalThreadStateException* is thrown.
- In such case, thread will run once but for second time, it will throw exception.

Thread.sleep() in Java

- The method sleep() is being used to halt the working of a thread for a given amount of time.
- The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread

The sleep() Method Syntax:

```
public static void sleep(long mls) throws InterruptedException
public static void sleep(long mls, int n) throws InterruptedException
```

Example- Sleep()

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getId());
        try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread " + Thread.currentThread().getId() + ": " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      }catch (InterruptedException e) {
         System.out.println("Thread " +  Thread.currentThread().getId() + " interrupted.");
      }
      System.out.println("Thread " +  Thread.currentThread().getId() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1,"Thread 1");
      t1.start();
      Thread t2=new Thread(R1,"Thread 2");
      t2.start();
   }
}
```

```
Running 22
Running 21
Thread 22: 4
Thread 21: 4
Thread 22: 3
Thread 21: 3
Thread 22: 2
Thread 21: 2
Thread 22: 1
Thread 21: 1
Thread 22 exiting.
Thread 21 exiting.
```

What if we call Java run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

Calling start()

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getId());
        for(int i = 4; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getId() + ": " + i);
        }
      System.out.println("Thread " +  Thread.currentThread().getId() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1,"Thread 1");
      Thread t2=new Thread(R1,"Thread 2");
      Thread t3=new Thread(R1,"Thread 3");
      t1.start();
      t2.start();
      t3.start();
   }
}
```

Note:

- Separate call stack for each thread
- Threads may execute in any order.
- Different outputs are possible

```
Running 22
Running 23
Running 21
Thread 23: 4
Thread 22: 4
Thread 23: 3
Thread 21: 4
Thread 23: 2Thread 22: 3
Thread 22: 2
Thread 23: 1
Thread 21: 3
Thread 23 exiting.
Thread 22: 1
Thread 21: 2
Thread 22 exiting.
Thread 21: 1
Thread 21 exiting.
```

Calling run()

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getId());
        for(int i = 4; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getId() + ": " + i);
        }
      System.out.println("Thread " +  Thread.currentThread().getId() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1,"Thread 1");
      Thread t2=new Thread(R1,"Thread 2");
      Thread t3=new Thread(R1,"Thread 3");
      t1.run();
      t2.run();
      t3.run();
   }
}
```

Note:

- All threads will lie in single call stack.
- no context-switching because here t1, t2 and t3 will be treated as normal object not thread object.
- Therefore same Output Everytime.

```
Running 1
Thread 1: 4
Thread 1: 3
Thread 1: 2
Thread 1: 1
Thread 1 exiting.
Running 1
Thread 1: 4
Thread 1: 3
Thread 1: 2
Thread 1: 1
Thread 1 exiting.
Running 1
Thread 1: 4
Thread 1: 3
Thread 1: 2
Thread 1: 1
Thread 1 exiting.
```

Java join() method

- The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution.

join():

- When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state.
- The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state.
- If interruption of the thread occurs, then it throws the InterruptedException.

Syntax:

public final void join() **throws** InterruptedException

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getName());
        for(int i = 4; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getName() + ": " + i);
        }
     System.out.println("Thread " +  Thread.currentThread().getName() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1,"T1");
      Thread t2=new Thread(R1,"T2");
      Thread t3=new Thread(R1,"T3");
      t1.start();
      try{
                t1.join();
      }catch(Exception e){System.out.println(e);}
      t2.start();
      t3.start();
}
}
```

Note: T1 completes its task than only T2 & T3 will start

```
Running T1
Thread T1: 4
Thread T1: 3
Thread T1: 2
Thread T1: 1
Thread T1 exiting.
Running T2Running T3Thread T2: 4
Thread T2: 3
Thread T2: 2
Thread T2: 1
Thread T3: 4
Thread T2 exiting.
Thread T3: 3
Thread T3: 2
Thread T3: 1
Thread T3 exiting.
```

Syntax:

public final void join() **throws** InterruptedException

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getName());
        for(int i = 4; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getName() + ": " + i);
        }
     System.out.println("Thread " +  Thread.currentThread().getName() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1,"T1");
      Thread t2=new Thread(R1,"T2");
      Thread t3=new Thread(R1,"T3");
      t1.start();
      t2.start();
      try{
              t1.join();
      }catch(Exception e){System.out.println(e);}
     t3.start();
}
}
```

Note: T3 will start only when T1 & T2 both completes their task

```
Running T2
Running T1
Thread T1: 4
Thread T1: 3
Thread T1: 2
Thread T1: 1
Thread T2: 4
Thread T1 exiting.
Thread T2: 3
Thread T2: 2
Thread T2: 1 Thread T2 exiting.
Running T3
Thread T3: 4
Thread T3: 3
Thread T3: 2
Thread T3: 1
Thread T3 exiting.
```

Syntax:

public final synchronized void join(**long** mls) **throws** InterruptedException

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getId());
        for(int i = 4; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getId() + ": " + i);
        }
      System.out.println("Thread " +  Thread.currentThread().getId() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1,"Thread 1");
      Thread t2=new Thread(R1,"Thread 2");
      Thread t3=new Thread(R1,"Thread 3");
      t1.start();
      try{
              t1.join(20);  //wait till t1 is dead or till 20ms
      }catch(Exception e){System.out.println(e);}
      t2.start();
      t3.start();
   }
}
```

```
Running 21
Thread 21: 4
Thread 21: 3
Running 22Thread 22: 4
Thread 22: 3Thread 22: 2
Thread 21: 2
Thread 21: 1
Thread 21 exiting.
Thread 22: 1
Thread 22 exiting.
Running 23
Thread 23: 4Thread 23: 3Thread 23: 2
Thread 23: 1
Thread 23 exiting.
```

Naming Thread and Current Thread

- By default, each thread has a name, i.e. thread-0, thread-1 and so on.
- But we can change the name of the thread by using the setName() method.

public String getName(): is used to **return** the name of a thread.

public void setName(String name): is used to change the name of a thread.

Example- getName()

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getName());
        for(int i = 2; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getName() + ": " + i);
        }
     System.out.println("Thread " +  Thread.currentThread().getName() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
     RunnableDemo R1 = new RunnableDemo( );
     // Default Name assigned
     Thread t1=new Thread(R1); //Thread-0
     Thread t2=new Thread(R1); //Thread-1
     Thread t3=new Thread(R1);//Thread-2
     t1.start();
     t2.start();
     t3.start();
     System.out.println("First thread:"+t1.getName());
   }
}
```

```
First thread:Thread-0
Running Thread-1
Running Thread-2
Running Thread-0
Thread Thread-2: 2
Thread Thread-1: 2
Thread Thread-2: 1
Thread Thread-2 exiting.
Thread Thread-0: 2
Thread Thread-1: 1
Thread Thread-0: 1
Thread Thread-1 exiting.
Thread Thread-0 exiting.
```

Example- getName()

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getName());
        for(int i = 2; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getName() + ": " + i);
        }
      System.out.println("Thread " +  Thread.currentThread().getName() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1,"A");
      Thread t2=new Thread(R1,"B");
      Thread t3=new Thread(R1,"C");
      t1.start();
      t2.start();
      t3.start();
   }
}
```

```
Running B
Running A
Running C
Thread A: 2
Thread B: 2
Thread A: 1
Thread C: 2
Thread A exiting.
Thread B: 1
Thread B exiting.
Thread C: 1
Thread C exiting.
```

Example- setName()

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getName());
        for(int i = 1; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getName() + ": " + i);
        }
     System.out.println("Thread " +  Thread.currentThread().getName() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
     RunnableDemo R1 = new RunnableDemo( );
     Thread t1=new Thread(R1);
     Thread t2=new Thread(R1);
     Thread t3=new Thread(R1);
     t1.setName("First T"); //to set the name of thread
      t2.setName("Second T");
     t1.start();
     t2.start();
     t3.start();
     System.out.println("First thread:"+t1.getName());
     System.out.println("Second thread:"+t2.getName());
     System.out.println("Third thread:"+t3.getName());//default name

   }
}
```

```
First thread:First T
Running Second T
Running Thread-2
Running First T
Thread Thread-2: 1
Thread Second T: 1
Second thread:Second T
Thread Second T exiting.
Thread Thread-2 exiting.
Thread First T: 1
Third thread:Thread-2
Thread First T exiting.
```

Priority of a Thread

- Each thread has a priority.
- Priorities are represented by a number between 1 (Min) and 10 (Max).
- Java programmer can assign the priorities of a thread explicitly.

3 constants defined 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.

Setter & Getter Method of Thread Priority

public final int getPriority()

- The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority)

- The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority.
- The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

Example

```
class RunnableDemo implements Runnable {
    public void run() {
    System.out.println("Running " +  Thread.currentThread().getName());
        for(int i = 5; i > 0; i--) {
           System.out.println("Thread " + Thread.currentThread().getName() + ": " + i);
        }
      System.out.println("Thread " +  Thread.currentThread().getName() + " exiting.");
   }
}
public class TestThread {
   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( );
      Thread t1=new Thread(R1);
      Thread t2=new Thread(R1);
      Thread t3=new Thread(R1);
System.out.println("Priority of the thread t1 is : " + t1.getPriority());
System.out.println("Priority of the thread t2 is : " + t2.getPriority());
System.out.println("Priority of the thread t3 is : " + t3.getPriority());
        t1.start();
      t2.start();
      t3.start();
t1.setPriority(6);
t2.setPriority(3);
t3.setPriority(9);
   }
}
```

```
Priority of the thread t1 is : 5
Priority of the thread t2 is : 5
Priority of the thread t3 is : 5
Running Thread-0
Running Thread-2
Thread Thread-2: 5Thread Thread-2: 4
Running Thread-1
Thread Thread-2: 3
Thread Thread-0: 5Thread Thread-0: 4
Thread Thread-0: 3
Thread Thread-2: 2Thread Thread-1: 5
Thread Thread-2: 1
Thread Thread-0: 2
Thread Thread-2 exiting.Thread Thread-1: 4
Thread Thread-0: 1
Thread Thread-1: 3
Thread Thread-0 exiting.Thread Thread-1: 2
Thread Thread-1: 1
Thread Thread-1 exiting.
```

Example- Setting Priority

```
public class ThreadDemo extends Thread {
   public void run() {
      System.out.println("Running...");
   }
   public static void main(String[] args) {
      ThreadDemo thread1 = new ThreadDemo();
      ThreadDemo thread2 = new ThreadDemo();
      System.out.println("Default thread priority of Thread 1: " + thread1.getPriority());
      System.out.println("Default thread priority of Thread 2: " + thread2.getPriority());
      thread1.setPriority(MAX_PRIORITY);
      thread2.setPriority(MIN_PRIORITY);
      System.out.println("The maximum thread priority of Thread 1 is: " + thread1.getPriority());
      System.out.println("The minimum thread priority of Thread 2 is: " + thread2.getPriority());
      System.out.println("" + Thread.currentThread().getName());
      System.out.println("Default thread priority of Main Thread: " +
Thread.currentThread().getPriority());
      Thread.currentThread().setPriority(MAX_PRIORITY); //main thread
      System.out.println("The maximum thread priority of Main Thread is: " +
Thread.currentThread().getPriority());
   }
}
```

```
Default thread priority of Thread 1: 5
Default thread priority of Thread 2: 5
The maximum thread priority of Thread 1 is: 10
The minimum thread priority of Thread 2 is: 1
main
Default thread priority of Main Thread: 5
The maximum thread priority of Main Thread is: 10
```

Daemon Thread in Java

- **Daemon thread in Java** is a service provider thread that provides services to the user thread.
- Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- Daemon threads running automatically e.g. gc, finalizer etc.
- It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
public boolean isDaemon()	is used to check that current is daemon.

Example: Daemon Thread

```
public class TestDaemonThread1 extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
  System.out.println("user thread work");
 }
 }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
  TestDaemonThread1 t3=new TestDaemonThread1();

  t1.setDaemon(true);//now t1 is daemon thread

  t1.start();//starting threads
  t2.start();
  t3.start();
 }
}
```

Output:

```
daemon thread work
user thread work
user thread work
```

Example: Daemon Thread

```
class TestDaemonThread2 extends Thread{
 public void run(){
  System.out.println("Name: "+Thread.currentThread().getName());
  System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }

 public static void main(String[] args){
  TestDaemonThread2 t1=new TestDaemonThread2();
  TestDaemonThread2 t2=new TestDaemonThread2();
  t1.start();
  t1.setDaemon(true);//will throw exception here
  t2.start();
 }
}
```

Output:

```
exception in thread main:
java.lang.IllegalThreadStateException
```

Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

Synchronization in Java

- Synchronization in Java is the capability to control the access of multiple threads to any shared resource.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.
- The synchronization is mainly used to
  - To prevent thread interference.
  - To prevent consistency problem.
- There are two types of synchronization
  - Process Synchronization
  - Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

- Mutual Exclusive: keep threads from interfering with one another while sharing data. Three ways:
  - Synchronized method
  - Synchronized block
  - Static synchronization
- Cooperation (Inter-thread communication in java)

Concept of Lock in Java

- Synchronization is built around an internal entity known as the lock or monitor.
- Every object has a lock associated with it.
- By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
- From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

```
class Table{
void printTable(int n){//method not synchronized
  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);
}
}
class TestSynchronization1{
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
100
10
200
15
300
20
400
25
500
```

Java Synchronized Method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Java Synchronized Method

```
//example of java synchronized method
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
```

Example of synchronized method by using anonymous class

```
//Program of synchronized method by using annonymous class
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);}
   }

 }
}
        public class TestSynchronization3{
        public static void main(String args[]){
        final Table obj = new Table();//only one object
        Thread t1=new Thread(){
        public void run(){
        obj.printTable(5);
        }
        };
        Thread t2=new Thread(){
        public void run(){
        obj.printTable(100);
        }
        };
        t1.start();
        t2.start();
        }
        }
```

Output:

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

Synchronized Block in Java

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- Java synchronized block is more efficient than Java synchronized method.

Syntax:

```
synchronized (object reference expression) {
  //code block
}
```

Example of Synchronized Block

```
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:

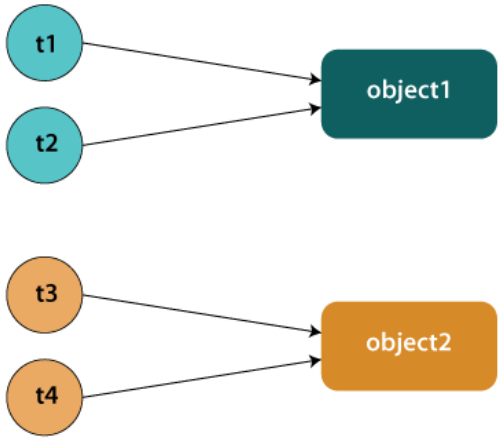
```
5
10
15
20
25
100
200
300
400
500
```

Static Synchronization

- If you make any static method as synchronized, the lock will be on the class not on object.

Problem without static synchronization

- Suppose there are two objects of a shared class (e.g. Table) named object1 and object2.
- In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.
- But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.
- We don't want interference between t1 and t3 or t2 and t4.
- Static synchronization solves this problem.



Example of Static Synchronization

```
class Table
{
 synchronized static void printTable(int n){
   for(int i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}
class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}
class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}
class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}
class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}
public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

Output:

```
1
2
..
9
10
10
20
..
90
100
100
200
..
1000
1000
..
10000
```

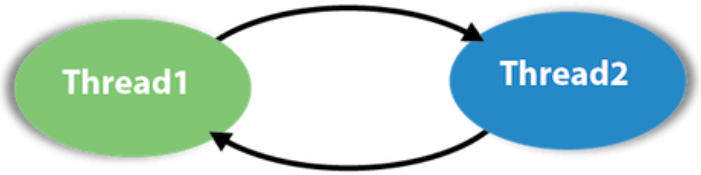
Example of static synchronization by Using the anonymous class

```
class Table{
 synchronized static  void printTable(int n){
   for(int i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}

public class TestSynchronization5 {
public static void main(String[] args) {

    Thread t1=new Thread(){
        public void run(){
            Table.printTable(1);
        }
    };

    Thread t2=new Thread(){
        public void run(){
            Table.printTable(10);
        }
    };

    Thread t3=new Thread(){
        public void run(){
            Table.printTable(100);
        }
    };

    Thread t4=new Thread(){
        public void run(){
            Table.printTable(1000);
        }
    };
    t1.start();
    t2.start();
    t3.start();
    t4.start();

}
}
```

Output:

```
1
2
..
9
10
10
20
 ..
90
100
100
200
..
1000
1000
..
10000
```

Deadlock in Java



- 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 of Deadlock in Java

```
public class TestDeadlockExample1 {
  public static void main(String[] args) {
    final String resource1 = “Ahan Chhetri";
    final String resource2 = “Rohan Chhetri";
    // t1 tries to lock resource1 then resource2
    Thread t1 = new Thread() {
      public void run() {
          synchronized (resource1) {
           System.out.println("Thread 1: locked resource 1");

           try { Thread.sleep(100);} catch (Exception e) {}

           synchronized (resource2) {
            System.out.println("Thread 1: locked resource 2");
           }
          }
      }
    };
```

```
    // t2 tries to lock resource2 then resource1
    Thread t2 = new Thread() {
      public void run() {
        synchronized (resource2) {
          System.out.println("Thread 2: locked resource 2");

          try { Thread.sleep(100);} catch (Exception e) {}

          synchronized (resource1) {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };
    t1.start();
    t2.start();
  }
}
```

Output:

```
Thread 1: locked resource 1
Thread 2: locked resource 2
```

More Complicated Deadlocks

Thread 1 locks A, waits for B
Thread 2 locks B, waits for C
Thread 3 locks C, waits for D
Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

Inter-thread Communication in Java

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of **Object class**:
  - wait()
  - notify()
  - notifyAll()

wait() method

- The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
- The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

notify() method

- The notify() method wakes up a single thread that is waiting on this object's monitor.
- If many threads are waiting on this object, one of them is chosen to be awakened.
- The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

notifyAll() method

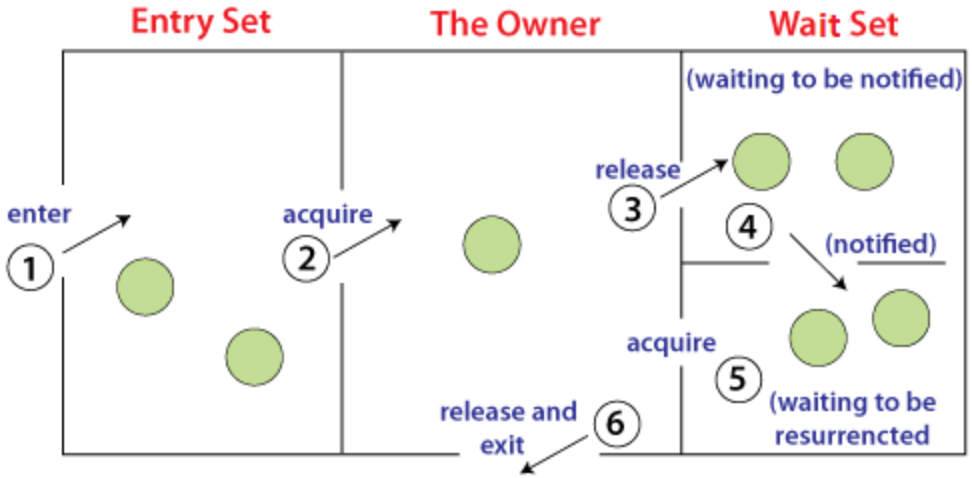
- Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
public final void notifyAll()
```

Understanding the process of inter-thread communication

1. Threads enter to acquire lock.
2. Lock is acquired by a thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.



Example of Inter Thread Communication in Java

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}
```

```
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}}
```

Output:
going to withdraw...
Less balance; waiting for deposit...
going to deposit...deposit completed...
withdraw completed...

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

- It is because they are related to lock and object has a lock.

Difference between wait and sleep?

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Interrupting a Thread

- If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.
- If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

The 3 methods provided by the Thread class for interrupting a thread

- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

Example of interrupting a thread that stops working

```
class TestInterruptingThread1 extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
throw new RuntimeException("Thread interrupted..."+e);
}
}

public static void main(String args[]){
TestInterruptingThread1 t1=new TestInterruptingThread1();
t1.start();
try{
t1.interrupt();
}catch(Exception e){System.out.println("Exception handled "+e);}
}
}
```

Note:
In this example, after interrupting the thread, we are propagating it, so it will stop working

Output:

```
Exception in thread-0
      java.lang.RuntimeException: Thread interrupted...
      java.lang.InterruptedException: sleep interrupted
      at A.run(A.java:7)
```

Example of interrupting a thread that doesn't stop working

```
class TestInterruptingThread2 extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
System.out.println("Exception handled "+e);
}
System.out.println("thread is running...");
}

public static void main(String args[]){
TestInterruptingThread2 t1=new TestInterruptingThread2();
t1.start();

t1.interrupt();

}
}
```

Note:
In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

Output:

```
Exception handled
    java.lang.InterruptedException: sleep interrupted
    thread is running...
```

Example of interrupting thread that behaves normally

```
class TestInterruptingThread3 extends Thread{

public void run(){
for(int i=1;i<=5;i++)
System.out.println(i);
}

public static void main(String args[]){
TestInterruptingThread3 t1=new TestInterruptingThread3();
t1.start();

t1.interrupt();

}
}
```

Note: If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

Output:

1
2
3
4
5

isInterrupted and interrupted method

- The isInterrupted() method returns the interrupted flag either true or false.
- The static interrupted() method returns the interrupted flag after that it sets the flag to false if it is true.

```
public class TestInterruptingThread4 extends Thread{

public void run(){
for(int i=1;i<=2;i++){
if(Thread.interrupted()){
System.out.println("code for interrupted thread");
}
else{
System.out.println("code for normal thread");
}

}//end of for loop
}
```

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

TestInterruptingThread4 t1=new TestInterruptingThread4();
TestInterruptingThread4 t2=new TestInterruptingThread4();

t1.start();
t1.interrupt();

t2.start();

}
}
```

Output:

```
Code for interrupted thread
code for normal thread
code for normal thread
code for normal thread
```

Volatile Keyword in Java

- Not substitute but an alternative way of achieving synchronization in Java.
- Volatile keyword is used to modify the value of a variable by different threads.
- The volatile keyword does not cache the value of the variable and always read the variable from the main memory.
- The volatile keyword cannot be used with classes or methods. However, it is used with variables.

Volatile keyword

```
class Test
{
static int var=5;
}
```

- assume that two threads are working on the same class.
- Both threads run on different processors where each thread has its local copy of var.
- If any thread modifies its value, the change will not reflect in the original one in the main memory.
- It leads to data inconsistency because the other thread is not aware of the modified value.

```
class Test
{
static volatile int var =5;
}
```

- static variables are class members that are shared among all objects.
- There is only one copy in the main memory.
- The value of a volatile variable will never be stored in the cache.
- All read and write will be done from and to the main memory.

Volatile- Important Points

- You can use the volatile keyword with variables. Using volatile keyword with classes and methods is illegal.
- It guarantees that value of the volatile variable will always be read from the main memory, not from the local thread cache.
- If you declared variable as volatile, Read and Writes are atomic
- It reduces the risk of memory consistency error.
- The volatile variables are always visible to other threads.
- When a variable is not shared between multiple threads, you do not need to use the volatile keyword with that variable.

Difference between synchronization and volatile keyword

Volatile Keyword	Synchronization Keyword
Volatile keyword is a field modifier.	Synchronized keyword modifies code blocks and methods.
The thread cannot be blocked for waiting in case of volatile.	Threads can be blocked for waiting in case of synchronized.
It improves thread performance.	Synchronized methods degrade the thread performance.
It synchronizes the value of one variable at a time between thread memory and main memory.	It synchronizes the value of all variables between thread memory and main memory.
Volatile fields are not subject to compiler optimization.	Synchronize is subject to compiler optimization.

Example of Volatile Keyword

- We have defined a class which increases the counter value.
- The run () method in the VolatileThread.java gets the updated value and old value when the thread begins execution.

VolatileData.java

```
public class VolatileData
{
private volatile int counter = 0;
public int getCounter()
{
return counter;
}
public void increaseCounter()
{
++counter;      //increases the value of counter by 1
}
}
```

Example- Volatile (p1)
See next slides for p2 & p3

Example- Volatile (p2)

VolatileThread.java

```
public class VolatileThread extends Thread
{
private final VolatileData data;
public VolatileThread(VolatileData data)
{
this.data = data;
}
@Override
public void run()
{
int oldValue = data.getCounter();
System.out.println("[Thread " + Thread.currentThread().getId() + "]: Old value = " + oldValue);
data.increaseCounter();
int newValue = data.getCounter();
System.out.println("[Thread " + Thread.currentThread().getId() + "]: New value = " + newValue);

}
}
```

Example- Volatile (p3)

VolatileMain.java

```
public class VolatileMain
{

private final static int noOfThreads = 2;
public static void main(String[] args) throws InterruptedException
{
VolatileData volatileData = new VolatileData();     //object of VolatileData class
Thread[] threads = new Thread[noOfThreads];     //creating Thread array
for(int i = 0; i < noOfThreads; ++i)
threads[i] = new VolatileThread(volatileData);
for(int i = 0; i < noOfThreads; ++i)
threads[i].start();                 //starts all reader threads
for(int i = 0; i < noOfThreads; ++i)
threads[i].join();                  //wait for all threads
}
}
```

Output:

```
[Thread 9]: Old value = 0
[Thread 9]: New value = 1
[Thread 10]: Old value = 1
[Thread 10]: New value = 2
```

Java Thread Pool

- **Java Thread pool** represents a group of worker threads that are waiting for the job and reused many times.
- In the case of a thread pool, a group of fixed-size threads is created.
- A thread from the thread pool is pulled out and assigned a job by the service provider.
- After completion of the job, the thread is contained in the thread pool again.

Advantages of Java Thread Pool

- **Better performance:** It saves time because there is no need to create a new thread.

Real time usage

- It is used in Servlet and JSP where the container creates a thread pool to process the request.

Thread Pool Methods

- **newFixedThreadPool(int s):** The method creates a thread pool of the fixed size s.
- **newCachedThreadPool():** The method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.
- **newSingleThreadExecutor():** The method creates a new thread.

Example of Java Thread Pool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class WorkerThread implements Runnable {
    private String message;
    public WorkerThread(String s){
        this.message=s;
    }

     public void run() {
        System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);

        processmessage();//call processmessage method that sleeps the thread for 2 seconds
        System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
    }

    private void processmessage() {
        try {  Thread.sleep(2000);  } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

Cont..

Example of Java Thread Pool Cont..

```
public class TestThreadPool {
    public static void main(String[] args) {
      ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads

      for (int i = 0; i < 10; i++) {
          Runnable worker = new WorkerThread("" + i);
          executor.execute(worker);//calling execute method of ExecutorService
       }
      executor.shutdown();

     while (!executor.isTerminated()) {   }

      System.out.println("Finished all threads");
   }
}
```

```
pool-1-thread-1 (Start) message = 0
pool-1-thread-2 (Start) message = 1
pool-1-thread-3 (Start) message = 2
pool-1-thread-5 (Start) message = 4
pool-1-thread-4 (Start) message = 3
pool-1-thread-2 (End)
pool-1-thread-2 (Start) message = 5
pool-1-thread-1 (End)
pool-1-thread-1 (Start) message = 6
pool-1-thread-3 (End)
pool-1-thread-3 (Start) message = 7
pool-1-thread-4 (End)
pool-1-thread-4 (Start) message = 8
pool-1-thread-5 (End)
pool-1-thread-5 (Start) message = 9
pool-1-thread-2 (End)
pool-1-thread-1 (End)
pool-1-thread-4 (End)
pool-1-thread-3 (End)
pool-1-thread-5 (End)
Finished all threads
```

Note: The shutdown() method of ThreadPoolExecutor class initiates an orderly shutdown in which already submitted task is accepted, but no new task is accepted.

ThreadPool Example

```
// important import statements
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.text.SimpleDateFormat;

class Tasks implements Runnable
{
private String taskName;

// constructor of the class Tasks
public Tasks(String str)
{
// initializing the field taskName
taskName = str;
}

// Printing the task name and then sleeps for 1 sec
// The complete process is getting repeated five times
public void run()
{
try  {
for (int j = 0; j <= 5; j++)  {
if (j == 0)  {
Date dt = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("hh : mm : ss");
//prints the initialization time for every task
System.out.println("Initialization time for the task name: "+ taskName + " = " + sdf.format(dt));
}
else  {
Date dt = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("hh : mm : ss");
// prints the execution time for every task
System.out.println("Time of execution for the task name: " + taskName + " = " +sdf.format(dt));

}
// 1000ms = 1 sec
Thread.sleep(1000);
}
System.out.println(taskName + " is complete.");
}
catch(InterruptedException ie)  {
ie.printStackTrace();
}
}
}
```

Example Cont..

```
public class ThreadPoolExample
{
// Maximum number of threads in the thread pool
static final int MAX_TH = 3;

// main method
public static void main(String argvs[])
{
// Creating five new tasks
Runnable rb1 = new Tasks("task 1");
Runnable rb2 = new Tasks("task 2");
Runnable rb3 = new Tasks("task 3");
Runnable rb4 = new Tasks("task 4");
Runnable rb5 = new Tasks("task 5");

// creating a thread pool with MAX_TH number of
// threads size the pool size is fixed
ExecutorService pl = Executors.newFixedThreadPool(MAX_TH);

// passes the Task objects to the pool to execute (Step 3)
pl.execute(rb1);
pl.execute(rb2);
pl.execute(rb3);
pl.execute(rb4);
pl.execute(rb5);

// pool is shutdown
pl.shutdown();
}
}
```

Output->>>

Example Cont.. Output->

```
Initialization time for the task name: task 1 = 06 : 13 : 02
Initialization time for the task name: task 2 = 06 : 13 : 02
Initialization time for the task name: task 3 = 06 : 13 : 02
Time of execution for the task name: task 1 = 06 : 13 : 04
Time of execution for the task name: task 2 = 06 : 13 : 04
Time of execution for the task name: task 3 = 06 : 13 : 04
Time of execution for the task name: task 1 = 06 : 13 : 05
Time of execution for the task name: task 2 = 06 : 13 : 05
Time of execution for the task name: task 3 = 06 : 13 : 05
Time of execution for the task name: task 1 = 06 : 13 : 06
Time of execution for the task name: task 2 = 06 : 13 : 06
Time of execution for the task name: task 3 = 06 : 13 : 06
Time of execution for the task name: task 1 = 06 : 13 : 07
Time of execution for the task name: task 2 = 06 : 13 : 07
Time of execution for the task name: task 3 = 06 : 13 : 07
Time of execution for the task name: task 1 = 06 : 13 : 08
Time of execution for the task name: task 2 = 06 : 13 : 08
Time of execution for the task name: task 3 = 06 : 13 : 08
task 2 is complete.
Initialization time for the task name: task 4 = 06 : 13 : 09
task 1 is complete.
Initialization time for the task name: task 5 = 06 : 13 : 09
task 3 is complete.
Time of execution for the task name: task 4 = 06 : 13 : 10
Time of execution for the task name: task 5 = 06 : 13 : 10
Time of execution for the task name: task 4 = 06 : 13 : 11
Time of execution for the task name: task 5 = 06 : 13 : 11
Time of execution for the task name: task 4 = 06 : 13 : 12
Time of execution for the task name: task 5 = 06 : 13 : 12
Time of execution for the task name: task 4 = 06 : 13 : 13
Time of execution for the task name: task 5 = 06 : 13 : 13
Time of execution for the task name: task 4 = 06 : 13 : 14
Time of execution for the task name: task 5 = 06 : 13 : 14
task 4 is complete.
task 5 is complete.
```

Risks involved in Thread Pools

- **Deadlock:** all the threads that are executing are waiting for the results from the threads that are blocked and waiting in the queue because of the non-availability of threads for the execution may lead to deadlock.
- **Thread Leakage:** Leakage of threads occurs when a thread is being removed from the pool to execute a task but is not returning to it after the completion of the task.
  - For example, when a thread throws the exception and the pool class is not able to catch this exception, then the thread exits and reduces the thread pool size by 1.
  - If the same thing repeats a number of times, then there are fair chances that the pool will become empty, and hence, there are no threads available in the pool for executing other requests.
- **Resource Thrashing:** A lot of time is wasted in context switching among threads when the size of the thread pool is very large. Whenever there are more threads than the optimal number may cause the starvation problem, and it leads to resource thrashing.

Points to Remember

- Do not queue the tasks that are concurrently waiting for the results obtained from the other tasks. It may lead to a deadlock situation.
- Care must be taken whenever threads are used for the operation that is long-lived. It may result in the waiting of thread forever and will finally lead to the leakage of the resource.
- The thread pool has to be ended explicitly. If it does not happen, then the program continues to execute, and it never ends. Invoke the shutdown() method on the thread pool to terminate the executor. Note that if someone tries to send another task to the executor after shutdown, it will throw a RejectedExecutionException.

Tuning the Thread Pool

- The accurate size of a thread pool is decided by the number of available processors and the type of tasks the threads have to execute.
- The tasks may have to wait for I/O, and in such a scenario, one has to take into consideration the ratio of the waiting time (W) and the service time (S) for the request
- maximum size of the pool **P * (1 + W / S)** for the maximum efficiency.