Day 1: Multithreading Introduction,Thread class,Runnable Interface,Process based and ThreadBased multitasking

Multithreading In Java:

Multithreading is a programming concept in which the application can create a small unit of tasks to execute in parallel. If you are working on a computer, it runs multiple applications and allocates processing power to them. A simple program runs in sequence and the code statements execute one by one. This is a single-threaded application. But, if the programming language supports creating multiple threads and passes them to the operating system to run in parallel, it's called multithreading.

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

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

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

Advantages of Java Multithreading:

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at the 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 an exception occurs in a single thread.

Some of application areas to apply multithreading:

- To develop multimedia graphics
- To develop animations
- To develop video games
- To develop Web Servers

Multitasking:

Multitasking is the process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

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.
- The 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.

Thread-based Multitasking (Multithreading):

- Threads share the same address space.(a thread is a part of a process)
- A thread is lightweight.
- The cost of communication between the thread is low.

Note: At least one process is required for each thread.

2 Ways to create threads

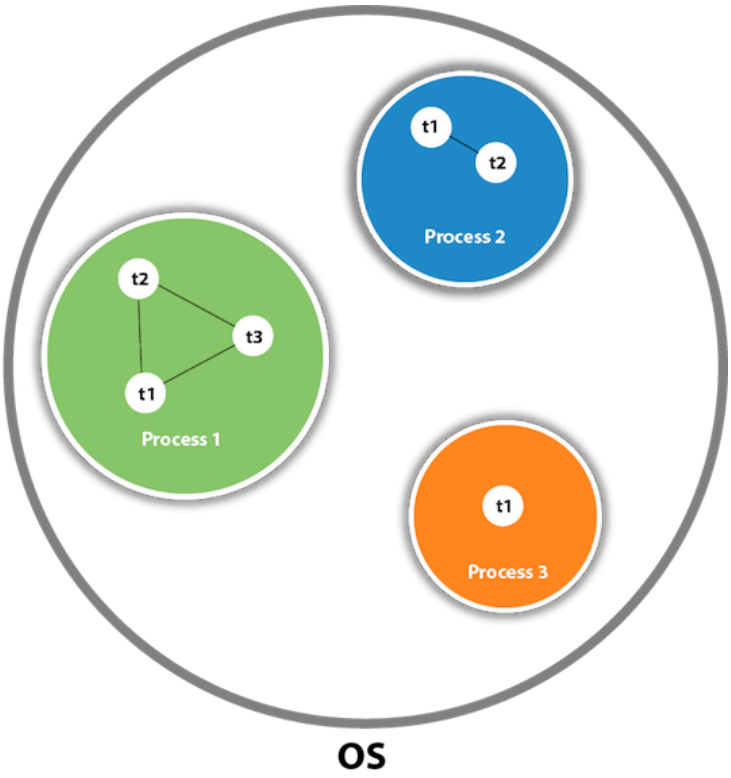
Thread in java:

An application when it is under execution is called a process.

A thread is a part or sub-process of an application.

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

A thread is a separate flow of execution that execute some functionality of a program with the other part of the program simultaneously.



Note: In Java, every program/application has a default flow of execution, a default thread, it is called the main thread. if we can start another flow of execution(another thread) along with the main thread simultaneously then it is called a multithreaded application or program.

Implementing thread in java:

Implementing thread in java is a two-step process:

1. first of all, we have to define a functionality that can be executed as a thread along with the main thread.
2. Start that functionality as a thread.

Thread class and Runnable interface are the two structures using which we implement Thread based multitasking in java.

The signature of a function using which we implement a thread is defined in an interface by the name **Runnable.**

This **Runnable** interface belongs to **java.lang** package. it has only one abstract method:

```
public void run();
```

Inside this run() method we need to define the functionality, that we want to execute as a thread along with the main thread. after providing the body we need to execute this functionality as a thread (i.e. simultaneously with the other part of the program)

There is a class by the name **Thread** present in **java.lang** package, which has a method called **start()**, this start() method is used to execute a given functionality defined inside the run() method of **Runnable** interface as a separate thread.

This start() method of the Thread class recognize the run() method of the Runnable interface and then the run() method is executed as a separate individual thread.

We implement threads either of the following two ways:

1. By implementing Runnable Interface
2. By extending Thread class

Example:

```
class A implements Runnable{

  @Override
  public void run(){
    //define the taks which we want to execute as a thread
  }
}

//or

class A extends Thread
{
  @Override
  public void run(){
    //define the taks which we want to execute as a thread
  }

}
```

Note: Internally the Thread class implements the Runnable interface and override run() method with empty implementation.

Example:

```
public class Thread implements Runnable{

  public void run(){
    //it is empty body overriden from Runnable interface
  }

  public void start(){//this is thread class own method....
  }

  //other methods of the Thread class

}
```

Note:-weather we extends Thread class or implement Runnable interface directly, we have to use run() method of the Runnable interface.

Example : Creating a new thread by extending the Thread class:

```
class X extends Thread{

  @Override
  public void run(){
    for(int i=0;i<30;i++){
      System.out.println("inside run mehod "+i);
    }
    System.out.println("end of run() method");
  }

  public static void main(String[] args){

    //------here one thread (main)

    X x1=new X();

    //x1.run();//it will be called as a normal method.
    x1.start();//here second thread will start

    for(int i=25;i<60;i++){
        System.out.println("inside main mehod"+i);
    }

    System.out.println("end of main()..");
  }
}
```

Here functionality of the start() method is to pick the run() method present in the object on which the start() method is called and to handover this run() method to the thread-scheduler for the Scheduling.
Control will be in the main() method and other statements of main() will be executed simultaneously along with run() method.

Since both the threads are getting executed simultaneously, the start/end execution of a thread completely depends on the time slice allocated for each method thread

Because of scheduling we can't guess the output of the above application.

Responsibility of start() method of Thread class:

The start() method present in the Thread class is responsible to perform all the mandatory activity which are required for our thread. (like, registering our thread with the Thread-scheduler, performing all the low level task to start a separate flow of execution. and then calling the run() method).

Hence without executing thread class start() method there is no chance of starting new thread in Java.

After starting a thread we are not allowed to restart the same thread once again otherwise we will get a runtime exception called **IllegalThreadStateException**.

Defining multiple Thread simultaneously:

```
class ThreadA extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("inside run mehod of ThreadA" + i);
        }
        System.out.println("end of ThreadA");
    }
}

    class ThreadB extends Thread {
    @Override
    public void run() {
        for (int i = 50; i < 60; i++) {
            System.out.println("inside run mehod of ThreadB" + i);
        }
        System.out.println("end of ThreadB");
    }
    }

    class ThreadC extends Thread {
    @Override
    public void run() {
            for (int i = 20; i < 30; i++) {
                System.out.println("inside run mehod of ThreadC" + i);
            }
            System.out.println("end of ThreadC");
        }
    }

  class Main {
```

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

            ThreadA t1 = new ThreadA();
            ThreadB t2 = new ThreadB();
            ThreadC t3 = new ThreadC();



            t1.start();
            t2.start();
            t3.start(); //4

            for (int i = 70; i < 80; i++) {
                System.out.println("inside main of Test:" + i);
            }
            System.out.println("end of main");
        }
    }
```

If we extends Thread class to create a new thread then we loose the chance of Object Orientation biggest advantage of inheritance i.e. we cannot extends another class simultaneously.

To solve the above problem, we use the Runnable interface, so we take a separate class by implementing Runnable interface and overrides the run() method and inside the run() method define the functionality which we want to execute as a separate thread,

then supply the object of that class to the Thread class constructor and create Thread class object and start that thread.

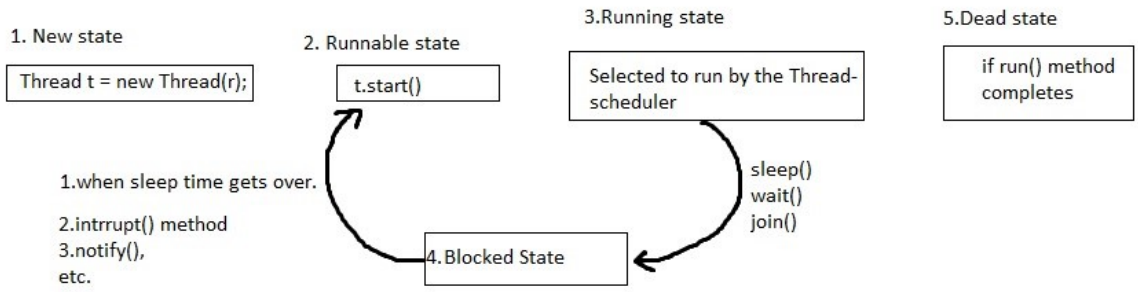
Example: Starting a thread using **Runnable** interface

```
class ThreadA implements Runnable{
  @Override
  public void run(){
    for(int i=20;i<40;i++){
      System.out.println("inside run() of ThreadA"+i);
    }
  }
  public static void main(String[] args){

    ThreadA t1=new ThreadA();
    Thread t=new Thread(t1); //passing Runnable object to the constructor of Thread class
    t.start();

    for(int i=20;i<40;i++){
      System.out.println("inside main of ThreadA:"+i);
    }
  }
}
```

Life-cycle of a Thread (State of a thread):

In Java, a thread always exists in any one of the following states. These states are:

1. New state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state



Getting and setting name of a thread:

Every thread in java has some name, it may be default name generated by the JVM or explicitly provided by the programmer.

Thread class has provides following getter and setter methods to get and set name of a thread.

public final String getName();

public final void setName(String name);

Example:

```
class ThreadA implements Runnable{

  @Override
```

```
  public void run() {

    for(int i=0;i<20;i++){
      String tname=Thread.currentThread().getName();
      System.out.println(tname +" is running ");
    }
  }
}

class ThreadB implements Runnable{

  @Override
  public void run() {

    for(int i=0;i<20;i++){
      String tname=Thread.currentThread().getName();
      System.out.println(tname +" is running ");
    }
  }
}


public class Main {

  public static void main(String[] args) {

    ThreadA ta=new ThreadA();
    ThreadB tb=new ThreadB();

    Thread t1=new Thread(ta);
    Thread t2=new Thread(tb);

    t1.setName("Raj");
    t2.setName("simran");

    t1.start();
    t2.start();

  }
}
```

Note:- in the above example, two separate thread executes on the two different objects simultaneously.

Running two or more threads on the same/single object:

We can run multiple threads on a single object also, which means one single functionality (run() method) of one object will be executed by two separate threads separately and individually in their own call stack.

.

```
public class RunThread implements Runnable{

  public void run(){

    for(int i=0;i<25;i++){
      String tname=Thread.currentThread().getName();
      System.out.println(tname +": is running");


    }
  }

  public static void main(String[] args){

    RunThread job=new RunThread();

    Thread one=new Thread(job);
    Thread two=new Thread(job);

    one.setName("Dhoni thread..");
    two.setName("Kohli thread..");

    one.start();
    two.start();
  }
}
```

Here also, if we run this program multiple time, we can’t predict the output.

Thread Priorities:

Every thread in java has some priority. it may be default priority generated by the JVM or explicitly provided by the programmer.

The valid range of thread priority is 1 to 10. but not 0 to 10.

1 is the least and 10 is the highest.

Note: priority should be assigned to the corresponding thread before calling the start() method.

Thread class defines the following constant to represent standard priorities:-

1. Thread.MAX_PRIORITY---->10
2. Thread.MIN_PRIORITY---->1
3. Thread.NORM_PRIORITY--->5

The thread scheduler will use these priorities while allocating the processer to our thread.

The thread which is having highest priority will get a chance first.

If two threads have the same priority then we can't expect the exact execution order. it depends on the thread scheduler.

Thread class defines the following setter and getter methods to get and set the priority of a thread.

1. **public final int getPriority();**
2. **public final void setPriority(int priority);** it allows values from 1 to 10 otherwise we will get a runtime exception.

Note:-the default priority for the main thread is 5 and for all remaining threads, it will be inherited from parent to child.

A high-priority thread does not run faster than a lower-priority thread.

Thread priority is not a rule for the thread scheduler, it is just a hint. so no guarantee for the execution (it will work only if the thread is in the waiting state or if there is a limited CPU time).

• If we are using thread priority for thread scheduling then we should always keep in mind that the underlying platform should provide support for scheduling based on thread priority.

Example:

```
class Main extends Thread {

    public void run(){
        System.out.println("Inside run method");
    }

    public static void main(String[] args){

         Main t1 = new Main();
         Main t2 = new Main();
         Main t3 = new Main();

        System.out.println("t1 thread priority : " + t1.getPriority());

        System.out.println("t2 thread priority : " + t2.getPriority());

        System.out.println("t3 thread priority : " + t3.getPriority());

        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

        // t3.setPriority(21); will throw IllegalArgumentException

        System.out.println("t1 thread priority : " + t1.getPriority());

        System.out.println("t2 thread priority : " + t2.getPriority());
```

```
        System.out.println("t3 thread priority : " + t3.getPriority());

       System.out.println("Currently Executing Thread : " + Thread.currentThread().getName());

       System.out.println("Main thread priority : "+ Thread.currentThread().getPriority());

       Thread.currentThread().setPriority(10);

        System.out.println("Main thread priority : "+Thread.currentThread().getPriority());
    }
}
```