Multithreading in Java

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 area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of 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.

Multitasking

Multitasking is a 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)

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)

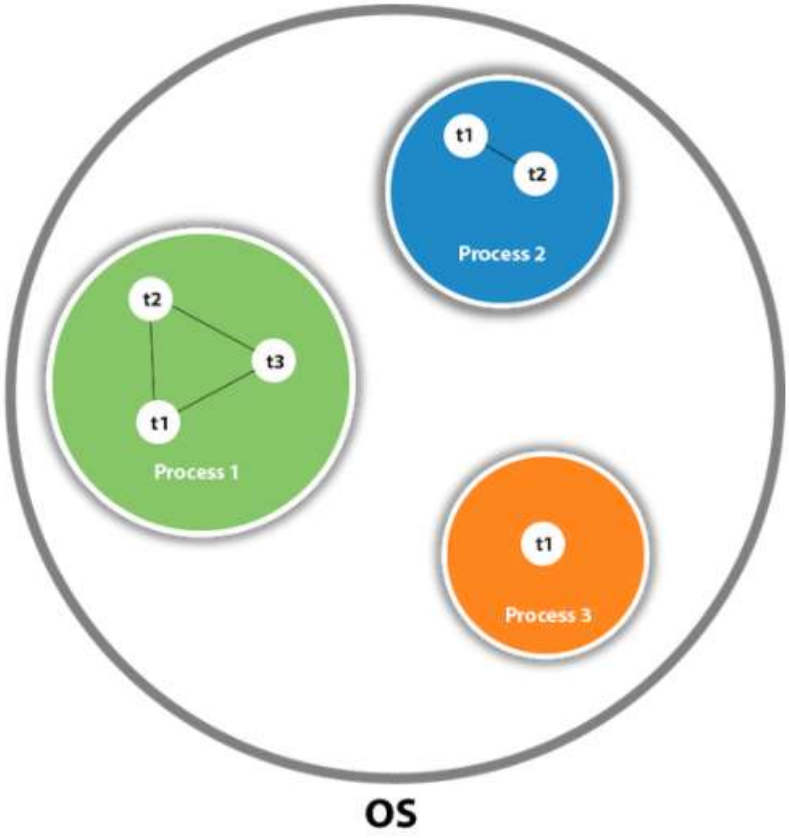
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

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

What is 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.



As shown in the above figure, 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.*

Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

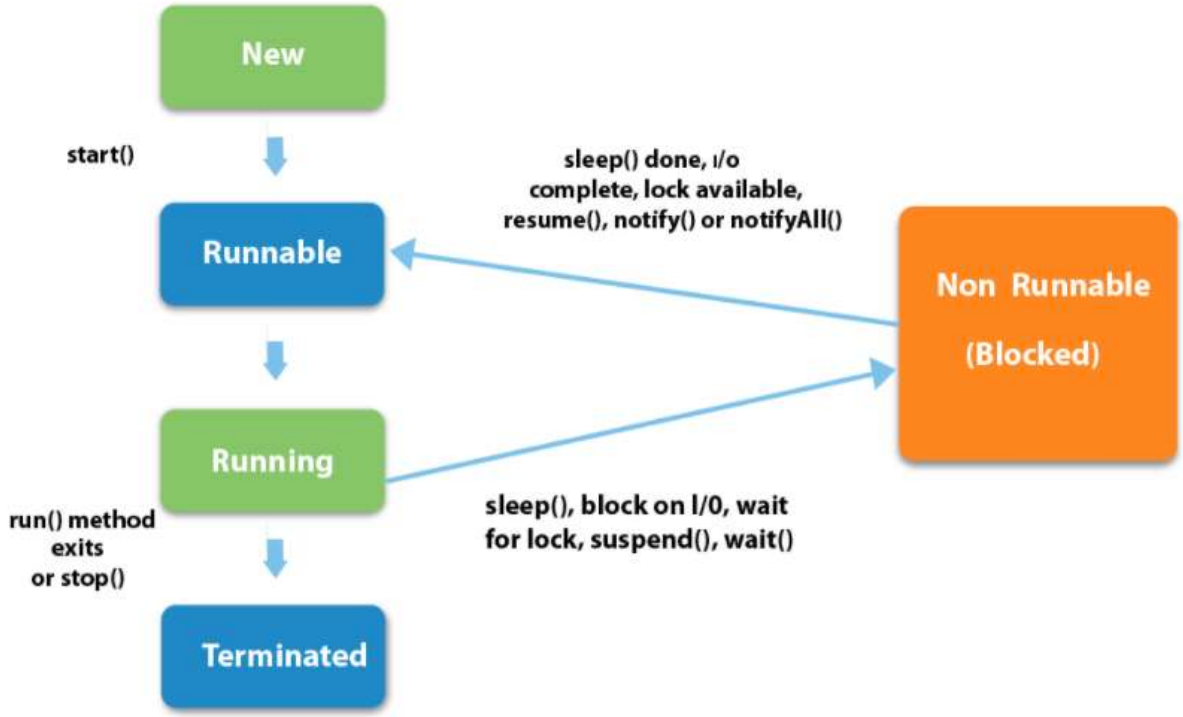
Life cycle of a Thread (Thread States)

1. Life cycle of a thread
    1. New
    2. Runnable
    3. Running
    4. Non-Runnable (Blocked)
    5. Terminated

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

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

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



1) New

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

2) Runnable

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

3) Running

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

4) Non-Runnable (Blocked)

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

5) Terminated

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

How to create thread

There are two ways to create a thread:

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

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

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

- Thread(Runnable r,String name)

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().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs 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.

---

1) Java Thread Example by extending Thread class

```
1. class Multi extends Thread{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5. public static void main(String args[]){
6. Multi t1=new Multi();
7. t1.start();
8.  }
9. }
Output:thread is running...
```

---

2) Java Thread Example by implementing Runnable interface

```
1. class Multi3 implements Runnable{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5.
6. public static void main(String args[]){
7. Multi3 m1=new Multi3();
```

8. Thread t1 =new Thread(m1); //necessary so that m1 gets treated as thread object and runs the Runnable interface run()
9. t1.start();
10. }
11. }

```
Output:thread is running...
```

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

Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly use preemptive or time slicing scheduling to schedule the threads.

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method in java

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();
```

```
  t1.start();
  t2.start();
 }
}
```

Output:

```
1
1
2
2
3
3
4
4
```

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Can we start a thread twice

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.

Let's understand it by the example given below:

```
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();
 }
}
```

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

The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads

to stop executing until the thread it joins with completes its task.

Syntax:

public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException

Example of join() method

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

```
Output:1
       2
       3
       4
       5
       1
       1
       2
       2
       3
       3
       4
       4
       5
       5
```

As you can see in the above example, when t1 completes its task then t2 and t3 starts executing.

Example of join(long miliseconds) method

```
class TestJoinMethod2 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod2 t1=new TestJoinMethod2();
 TestJoinMethod2 t2=new TestJoinMethod2();
 TestJoinMethod2 t3=new TestJoinMethod2();
 t1.start();
 try{
  t1.join(1500);
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

```
Output:1
       2
       3
       1
       4
       1
       2
       5
       2
       3
       3
       4
       4
       5
       5
```

In the above example, when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing.

getName(),setName(String) and getId() method:

public String getName()

public void setName(String name)

public long getId()

```
class TestJoinMethod3 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestJoinMethod3 t1=new TestJoinMethod3();
  TestJoinMethod3 t2=new TestJoinMethod3();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());
  System.out.println("id of t1:"+t1.getId());

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

  t1.setName("MyThread");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

```
Output:Name of t1:Thread-0
       Name of t2:Thread-1
       id of t1:8
       running...
       After changing name of t1:MyThread
       running...
```

The currentThread() method:

The currentThread() method returns a reference to the currently executing thread object.

Syntax:

public static Thread currentThread()

Example of currentThread() method

```
class TestJoinMethod4 extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 }
 public static void main(String args[]){
  TestJoinMethod4 t1=new TestJoinMethod4();
  TestJoinMethod4 t2=new TestJoinMethod4();

  t1.start();
  t2.start();
 }
}
```

```
Output:Thread-0
       Thread-1
```

Priority of a Thread (Thread Priority):

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

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
```

```
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

 }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

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

Java Thread stop() method

The **stop()** method of thread class terminates the thread execution. Once a thread is stopped, it cannot be restarted by start() method.

Syntax

1. **public final void** stop()
2. **public final void** stop(Throwable obj)

Parameter

obj : The Throwable object to be thrown.

Return

This method does not return any value.

Killing threads in Java

Last Updated: 24-12-2018

A thread is automatically destroyed when the run() method has completed. But it might be required to kill/stop a thread before it has completed its life cycle. Previously, methods **suspend()**, **resume()** and **stop()** were used to manage the execution of threads. But these methods were deprecated by Java 2 because they could result in system failures. Modern ways to suspend/stop a thread are by using a boolean flag and Thread.interrupt() method.

1. **Using a boolean flag:** We can define a boolean variable which is used for stopping/killing threads say 'exit'. Whenever we want to stop a thread, the 'exit' variable will be set to true.

```
// Java program to illustrate

// stopping a thread using boolean flag


class MyThread implements Runnable {


      // to stop the thread

      private boolean exit;


      private String name;

      Thread t;


      MyThread(String threadname)

      {

            name = threadname;

            t = new Thread(this, name);

            System.out.println("New thread: " + t);

            exit = false;

            t.start(); // Starting the thread

      }


      // execution of thread starts from run() method

      public void run()
```

```
	{
		int i = 0;
		while (!exit) {
			System.out.println(name + ": " + i);
			i++;
			try {
				Thread.sleep(100);
			}
			catch (InterruptedException e) {
				System.out.println("Caught:" + e);
			}
		}
		System.out.println(name + " Stopped.");
	}

	// for stopping the thread
	public void stop()
	{
		exit = true;
	}
}

// Main class
public class Main {
	public static void main(String args[])
	{
		// creating two objects t1 & t2 of MyThread
		MyThread t1 = new MyThread("First thread");
		MyThread t2 = new MyThread("Second thread");
		try {
```

```
                Thread.sleep(500);
                t1.stop(); // stopping thread t1
                t2.stop(); // stopping thread t2
                Thread.sleep(500);
            }
            catch (InterruptedException e) {
                System.out.println("Caught:" + e);
            }
            System.out.println("Exiting the main Thread");
        }
}
```

Output:

```
New thread: Thread[First  thread, 5, main]
New thread: Thread[Second thread, 5, main]
First  thread: 0
Second thread: 0
First  thread: 1
Second thread: 1
First  thread: 2
Second thread: 2
First  thread: 3
Second thread: 3
First  thread: 4
Second thread: 4
First  thread: 5
Second thread Stopped.
First  thread Stopped.
Exiting the main Thread
```

Note: The output may vary every time.
By using a flag we can stop a thread whenever we want to and we can prevent unwanted run-time errors.

How do I determine if a thread is alive or not?

In JavaSW, a thread is considered to be alive when its start() method has been called. After the run() method finishes, the thread is considered not to be alive anymore. We can determine if a thread is currently alive or not by calling a Thread instance's isAlive() method. The getState() method can also be useful. It returns a Thread. State enum representing the current state of the thread.
We can demonstrate these methods with a simple example. First, we'll implement a job to be run by a thread called RunnableJob.

RunnableJob.java

```
package com.cakes;

public class RunnableJob implements Runnable {

	@Override
	public void run() {
		System.out.println("RunnableJob is running");
	}

}
```

Next we'll create a ThreadExample class. This class creates a Thread instance with a RunnableJob instance. After creating this thread, we call getState() and isAlive() on the thread. After this, we start the thread via its start() method. Once again, we call getState() and isAlive(). After that, we pause the current thread of execution for one second, and then we once again call getState() and isAlive().

ThreadExample.java

```
package com.cakes;

public class ThreadExample {

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

		RunnableJob runnableJob = new RunnableJob();
		Thread thread = new Thread(runnableJob);

		displayStateAndIsAlive(thread);
		thread.start();
		displayStateAndIsAlive(thread);
		Thread.sleep(1000);
		displayStateAndIsAlive(thread);
	}

	public static void displayStateAndIsAlive(Thread thread) {
		// java.lang.Thread.State can be NEW, RUNNABLE, BLOCKED, WAITING,
TIMED_WAITING, TERMINATED
		System.out.println("State:" + thread.getState());
		System.out.println("Is alive?:" + thread.isAlive());
	}

}
```

The console output of the execution of ThreadExample is shown here. Notice that when the thread has been created but not started, its state is NEW and it is not alive. After we start the thread, its state is RUNNABLE and it is alive. During the second-long pause, we can see from the console output that the run() method of our job has been called. After the pause, the state of the thread is TERMINATED and the thread is no longer alive.

Console Output

```
State:NEW



Is alive?:false



State:RUNNABLE



Is alive?:true



RunnableJob is running



State:TERMINATED



Is alive?:false
```

```
// Java program to illustrate
// isAlive()

public class oneThread extends Thread {
    public void run()
    {
        System.out.println("geeks ");
        try {
            Thread.sleep(300);
        }
        catch (InterruptedException ie) {
        }
        System.out.println("forgeeks ");
    }
    public static void main(String[] args)
    {
        oneThread c1 = new oneThread();
        oneThread c2 = new oneThread();
        c1.start();
        c2.start();
        System.out.println(c1.isAlive());
        System.out.println(c2.isAlive());
    }
}
```

Output:

```
geeks
true
```

```
true
geeks
forgeeks
forgeeks
```