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)

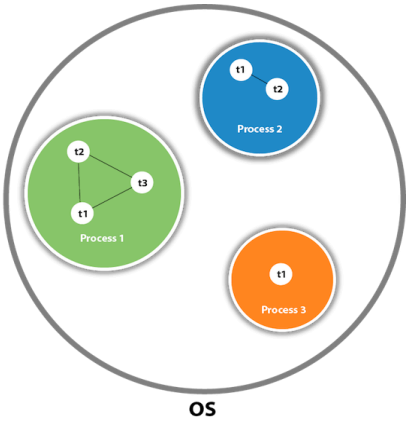
- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists,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

8.1 Introduction of Thread



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.

8.2 Creating a 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)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **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().

1. **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.

1) Java Thread Example by extending Thread class

FileName: Multi.java

```
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

FileName: Multi3.java

```
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);   // Using the constructor Thread(Runnable r)
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 the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

FileName: MyThread1.java

```
1. public class MyThread1
2. {
3. // Main method
4. public static void main(String argvs[])
5. {
```

```
6.  // creating an object of the Thread class using the constructor Thread(String name)
7.  Thread t= new Thread("My first thread");
8.
9.  // the start() method moves the thread to the active state
10. t.start();
11. // getting the thread name by invoking the getName() method
12. String str = t.getName();
13. System.out.println(str);
14. }
15. }
```

Output:

My first thread

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

Observe the following program.

FileName: MyThread2.java

```
1.  public class MyThread2 implements Runnable
2.  {
3.  public void run()
4.  {
5.  System.out.println("Now the thread is running ...");
6.  }
7.
8.  // main method
9.  public static void main(String argvs[])
10. {
11. // creating an object of the class MyThread2
12. Runnable r1 = new MyThread2();
13.
14. // creating an object of the class Thread using Thread(Runnable r, String name)
15. Thread th1 = new Thread(r1, "My new thread");
16.
17. // the start() method moves the thread to the active state
18. th1.start();
19.
20. // getting the thread name by invoking the getName() method
21. String str = th1.getName();
22. System.out.println(str);
23. }
24. }
```

Output:

My new thread
Now the thread is running ...

8.3 Thread Priorities

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler 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. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the 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).

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:

FileName: ThreadPriorityExample.java

```
// Importing the required classes
import java.lang.*;

public class ThreadPriorityExample extends Thread
{

// Method 1
// Whenever the start() method is called by a thread
// the run() method is invoked
public void run()
{
// the print statement
System.out.println("Inside the run() method");
}

// the main method
public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

// We did not mention the priority of the thread.
// Therefore, the priorities of the thread is 5, the default value

// 1st Thread
// Displaying the priority of the thread
// using the getPriority() method
System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 2nd Thread
// Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 3rd Thread
// // Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
```

```
39.
40.  // Setting priorities of above threads by
41.  // passing integer arguments
42.  th1.setPriority(6);
43.  th2.setPriority(3);
44.  th3.setPriority(9);
45.
46.  // 6
47.  System.out.println("Priority of the thread th1 is : " + th1.getPriority());
48.
49.  // 3
50.  System.out.println("Priority of the thread th2 is : " + th2.getPriority());
51.
52.  // 9
53.  System.out.println("Priority of the thread th3 is : " + th3.getPriority());
54.
55.  // Main thread
56.
57.  // Displaying name of the currently executing thread
58.  System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());
59.
60.  System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
61.
62.  // Priority of the main thread is 10 now
63.  Thread.currentThread().setPriority(10);
64.
65.  System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
66.  }
67.  }
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

FileName: ThreadPriorityExample1.java

```
1.  // importing the java.lang package
2.  import java.lang.*;
3.
4.  public class ThreadPriorityExample1 extends Thread
5.  {
6.
7.  // Method 1
8.  // Whenever the start() method is called by a thread
9.  // the run() method is invoked
10. public void run()
```

```
{
// the print statement
System.out.println("Inside the run() method");
}


// the main method
public static void main(String argvs[])
{

// Now, priority of the main thread is set to 7
Thread.currentThread().setPriority(7);

// the current thread is retrieved
// using the currentThread() method

// displaying the main thread priority
// using the getPriority() method of the Thread class
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

// creating a thread by creating an object of the class ThreadPriorityExample1
ThreadPriorityExample1 th1 = new ThreadPriorityExample1();

// th1 thread is the child of the main thread
// therefore, the th1 thread also gets the priority 7

// Displaying the priority of the current thread
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
}
}
```

Output:

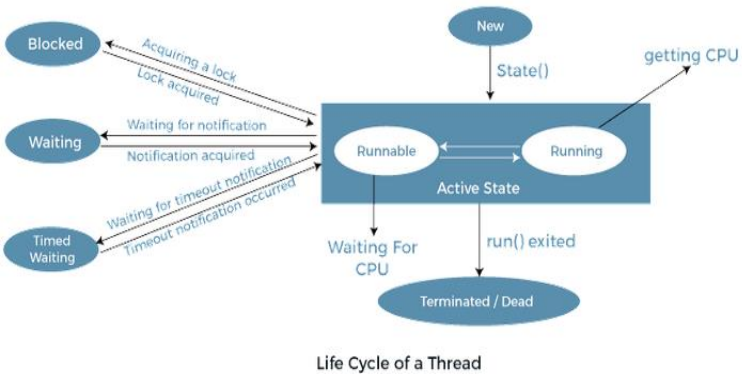
Priority of the main thread is : 7
Priority of the thread th1 is : 7

Explanation: If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

8.4 Life cycle of a Thread (Thread stated)

The *java.lang.Thread* class contains a *static State enum* – which defines its potential states. During any given point of time, the thread can only be in one of these states:

1. ***NEW*** **–** a newly created thread that has not yet started the execution
2. ***RUNNABLE*** **–** either running or ready for execution but it's waiting for resource allocation
3. ***BLOCKED*** **–** waiting to acquire a monitor lock to enter or re-enter a synchronized block/method
4. ***WAITING*** **–** waiting for some other thread to perform a particular action without any time limit
5. ***TIMED_WAITING*** **–** waiting for some other thread to perform a specific action for a specified period
6. ***TERMINATED*** **–** has completed its execution



Life Cycle of a Thread

All these states are covered in the diagram above; let's now discuss each of these in detail.

8.4.1. New

A *NEW Thread* (or a Born *Thread*) is a thread that's been created but not yet started. It remains in this state until we start it using the *start()* method.

The following code snippet shows a newly created thread that's in the *NEW* state:

```
Runnable runnable = new NewState();
Thread t = new Thread(runnable);
System.out.println(t.getState());
```

Since we've not started the mentioned thread, the method *t.getState()* prints:

```
NEW
```

8.4.2. Runnable

When we've created a new thread and called the *start()* method on that, it's moved from *NEW* to *RUNNABLE* state. **Threads in this state are either running or ready to run, but they're waiting for resource allocation from the system.**

In a multi-threaded environment, the Thread-Scheduler (which is part of JVM) allocates a fixed amount of time to each thread. So it runs for a particular amount of time, then relinquishes the control to other *RUNNABLE* threads.

For example, let's add *t.start()* method to our previous code and try to access its current state:

```
Runnable runnable = new NewState();
Thread t = new Thread(runnable);
t.start();
System.out.println(t.getState());
```

This code is **most likely** to return the output as:

```
RUNNABLE
```

Note that in this example, it's not always guaranteed that by the time our control reaches *t.getState()*, it will be still in the *RUNNABLE* state.

It may happen that it was immediately scheduled by the *Thread-Scheduler* and may finish execution. In such cases, we may get a different output.

8.4.3. Blocked

A thread is in the *BLOCKED* state when it's currently not eligible to run. **It enters this state when it is waiting for a monitor lock and is trying to access a section of code that is locked by some other thread.**

Let's try to reproduce this state:

```
public class BlockedState {
   public static void main(String[] args) throws InterruptedException {
      Thread t1 = new Thread(new DemoBlockedRunnable());
      Thread t2 = new Thread(new DemoBlockedRunnable());

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

      Thread.sleep(1000);
```

```
        System.out.println(t2.getState());
        System.exit(0);
    }
}

class DemoBlockedRunnable implements Runnable {
    @Override
    public void run() {
        commonResource();
    }

    public static synchronized void commonResource() {
        while(true) {
            // Infinite loop to mimic heavy processing
            // 't1' won't leave this method
            // when 't2' try to enter this
        }
    }
}
```

In this code:

1. We've created two different threads – *t1* and *t2*
2. *t1* starts and enters the synchronized *commonResource()* method; this means that only one thread can access it; all other subsequent threads that try to access this method will be blocked from the further execution until the current one will finish the processing
3. When *t1* enters this method, it is kept in an infinite while loop; this is just to imitate heavy processing so that all other threads cannot enter this method
4. Now when we start *t2*, it tries to enter the *commonResource()* method, which is already being accessed by *t1*, thus, *t2* will be kept in the *BLOCKED* state

Being in this state, we call *t2.getState()* and get the output as:

```
BLOCKED
```

8.4.4. Waiting

A thread is in *WAITING* state when it's waiting for some other thread to perform a particular action. According to JavaDocs, any thread can enter this state by calling any one of the following three methods:

1. *object.wait()*
2. *thread.join()* or
3. *LockSupport.park()*

Note that in *wait()* and *join()* – we do not define any timeout period as that scenario is covered in the next section.

We have a separate tutorial that discusses in detail the use of *wait(), notify()* and *notifyAll()*.

For now, let's try to reproduce this state:

```
public class WaitingState implements Runnable {
    public static Thread t1;

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

```
    public void run() {
        Thread t2 = new Thread(new DemoWaitingStateRunnable());
        t2.start();

        try {
            t2.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}

class DemoWaitingStateRunnable implements Runnable {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }

        System.out.println(WaitingState.t1.getState());
    }
}
```

Let's discuss what we're doing here:

1. We've created and started the *t1*
2. *t1* creates a *t2* and starts it
3. While the processing of *t2* continues, we call *t2.join()*, this puts *t1* in *WAITING* state until *t2* has finished execution
4. Since *t1* is waiting for *t2* to complete, we're calling *t1.getState()* from *t2*

Output:
WAITING

8.4.5. Timed Waiting

A thread is in *TIMED_WAITING* state when it's waiting for another thread to perform a particular action within a stipulated amount of time.

According to JavaDocs, there are five ways to put a thread on *TIMED_WAITING* state:

1. *thread.sleep(long millis)*
2. *wait(int timeout)* or *wait(int timeout, int nanos)*
3. *thread.join(long* millis*)*
4. *LockSupport.parkNanos*
5. *LockSupport.parkUntil*

To read more about the differences between *wait()* and *sleep()* in Java, have a look at this dedicated article here.

For now, let's try to quickly reproduce this state:

```
public class TimedWaitingState {
    public static void main(String[] args) throws InterruptedException {
        DemoTimeWaitingRunnable runnable= new DemoTimeWaitingRunnable();
        Thread t1 = new Thread(runnable);
        t1.start();
```

```
        // The following sleep will give enough time for ThreadScheduler
        // to start processing of thread t1
        Thread.sleep(1000);
        System.out.println(t1.getState());
    }
}

class DemoTimeWaitingRunnable implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Here, we've created and started a thread *t1* which is entered into the sleep state with a timeout period of 5 seconds; the output will be:

TIMED_WAITING

8.4.6. Terminated

This is the state of a dead thread. **It's in the *TERMINATED* state when it has either finished execution or was terminated abnormally.**

Let's try to achieve this state in the following example:

```
public class TerminatedState implements Runnable {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new TerminatedState());
        t1.start();
        // The following sleep method will give enough time for
        // thread t1 to complete
        Thread.sleep(1000);
        System.out.println(t1.getState());
    }

    @Override
    public void run() {
        // No processing in this block
    }
}
```

Here, while we've started thread *t1*, the very next statement *Thread.sleep(1000)* gives enough time for *t1* to complete and so this program gives us the output as:

TERMINATED

In addition to the thread state, we can check the *isAlive()* method to determine if the thread is alive or not. For instance, if we call the *isAlive()* method on this thread:

```
Assert.assertFalse(t1.isAlive());
```

It returns *false.* Put simply, **a thread is alive if and only if it has been started and has not yet died.**