

Java Threads: Thread Life Cycle and Threading Basics

By Ravikiran A S

Last updated on Dec 24, 2024

173122

Share This Article:



What is a Thread in Java?

A thread in [Java](#) is the direction or path that is taken while a program is being executed. Generally, all the programs have at least one thread, known as the main thread, that is provided by the JVM or [Java Virtual Machine](#) at the starting of the program's execution. At this point, when the main thread is provided, the `main()` method is invoked by the main thread.

A thread is an execution thread in a program. Multiple threads of execution can be run concurrently by an application running on the Java Virtual Machine. The priority of each thread varies. Higher priority threads are executed before lower priority threads.

Thread is critical in the program because it enables multiple operations to take place within a single method. Each thread in the program often has its own program counter, stack, and local variable.

Thread in Java enables concurrent execution, dividing tasks for improved performance. It's essential for handling operations like I/O and network communication efficiently. Understanding threads is crucial for responsive Java applications. Enroll in a [Java Course](#) to master threading and create efficient multithreaded programs.

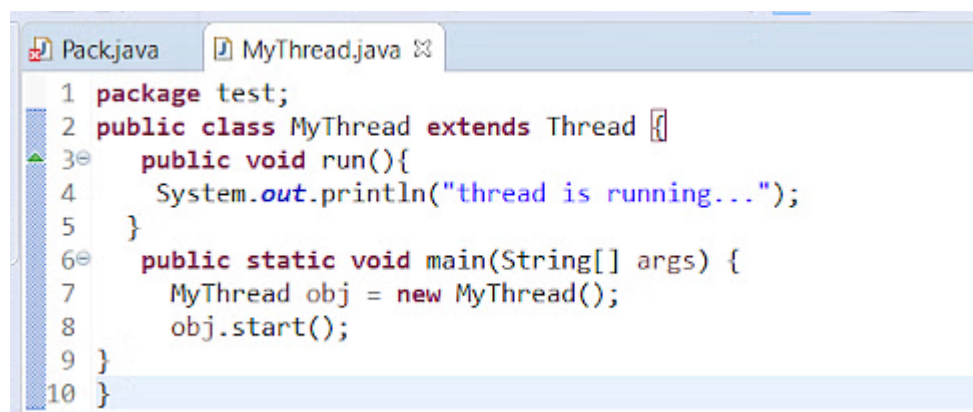
Creating a Thread in Java

A thread in Java can be created in the following two ways:

- **Extending java.lang.Thread class**

In this case, a thread is created by a new [class](#) that extends the Thread class, creating an instance of that class. The run() method includes the functionality that is supposed to be implemented by the Thread.

Below is an example to create a thread by extending java.lang.Thread class.



```
1 package test;
2 public class MyThread extends Thread {
3     public void run(){
4         System.out.println("thread is running...");
5     }
6     public static void main(String[] args) {
7         MyThread obj = new MyThread();
8         obj.start();
9     }
10 }
```

Output



```
<terminated> MyThread [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (17-Mar-2021, 11:39:01 am – 11:39:06 am)
thread is running...
```

Here, start() is used to create a new thread and to make it runnable. The new thread begins inside the void run() method.

- **Implementing Runnable interface**

This is the easy method to create a thread among the two. In this case, a class is created to implement the runnable interface and then the run() method.

The code for executing the Thread should always be written inside the run() method.

Here's a code to make you understand it.

```
package test;
public class MyThread implements Runnable {
    public void run(){
        System.out.println("thread is running..");
    }
    public static void main(String[] args) {
        Thread t = new Thread(new MyThread());
        t.start();
    }
}
```

Output

```
<terminated> MyThread [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (17-Mar-2021, 11:49:41 am – 11:49:44 am)
thread is running..
```

The start() method is used to call the void run() method. When start() is called, a new stack is given to the thread, and run() is invoked to introduce a new thread in the program.

Want a Top Software Development Job? Start Here!

Full Stack Developer - MERN Stack

EXPLORE PROGRAM

Lifecycle of a Thread in Java

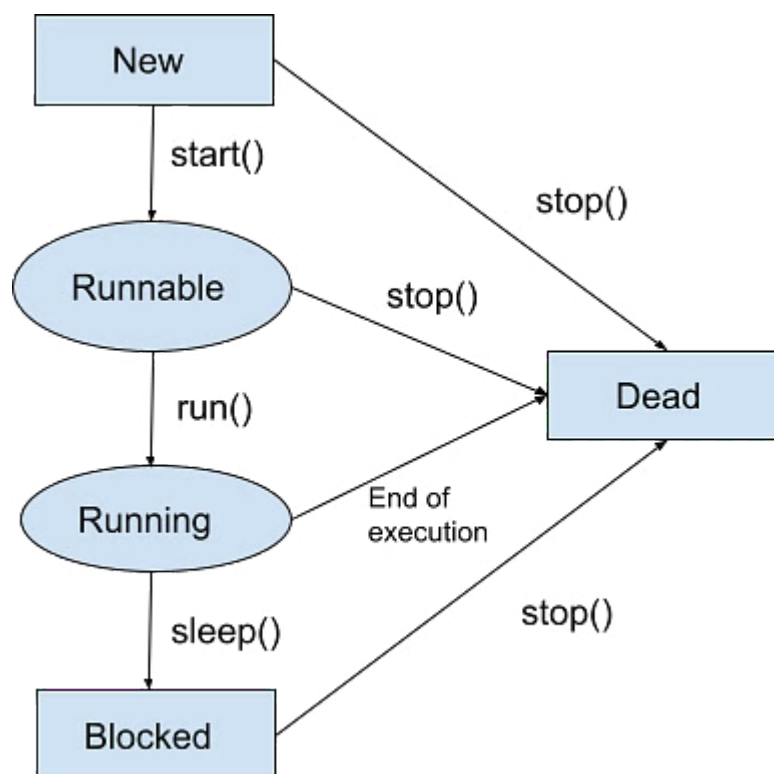
The Life Cycle of a Thread in Java refers to the state transformations of a thread that begins with

its birth and ends with its death. When a thread instance is generated and executed by calling the `start()` method of the `Thread` class, the thread enters the runnable state. When the `sleep()` or `wait()` methods of the `Thread` class are called, the thread enters a non-runnable mode.

Thread returns from non-runnable state to runnable state and starts statement execution. The thread dies when it exits the `run()` process. In [Java](#), these thread state transformations are referred to as the Thread life cycle.

There are basically 4 stages in the lifecycle of a thread, as given below:

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead



- **New State**

As we use the `Thread` class to construct a thread entity, the thread is born and is defined as being in the New state. That is, when a thread is created, it enters a new state, but the `start()` method on the instance has not yet been invoked.

- **Runnable State**

A thread in the runnable state is prepared to execute the `code`. When a new thread's `start()` function is called, it enters a runnable state.

In the runnable environment, the thread is ready for execution and is awaiting the processor's availability (CPU time). That is, the thread has entered the queue (line) of threads waiting for execution.

- **Running State**

Running implies that the processor (CPU) has assigned a time slot to the thread for execution. When a thread from the runnable state is chosen for execution by the thread scheduler, it joins the running state.

In the running state, the processor allots time to the thread for execution and runs its run procedure. This is the state in which the thread directly executes its operations. Only from the runnable state will a thread enter the running state.

- **Blocked State**

When the thread is alive, i.e., the thread class object persists, but it cannot be selected for execution by the scheduler. It is now inactive.

- **Dead State**

When a thread's `run()` function ends the execution of sentences, it automatically dies or enters the dead state. That is, when a thread exits the `run()` process, it is terminated or killed. When the `stop()` function is invoked, a thread will also go dead.

Want a Top Software Development Job? Start Here!

Full Stack Developer - MERN Stack

EXPLORE PROGRAM

Java Thread Priorities

The number of services assigned to a given thread is referred to as its priority. Any thread generated in the JVM is given a priority. The priority scale runs from 1 to 10.

1 is known as the lowest priority.

5 is known as standard priority.

10 represents the highest level of priority.

The main thread's priority is set to 5 by default, and each child thread will have the same priority as its parent thread. We have the ability to adjust the priority of any thread, whether it is the main thread or a user-defined thread. It is advised to adjust the priority using the Thread class's constants, which are as follows:

1. Thread.MIN_PRIORITY;
2. Thread.NORM_PRIORITY;
3. Thread.MAX_PRIORITY;

Below is a program to understand the Thread Priority.

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

Output

```
<terminated> ThreadPriority [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (17-Mar-2021, 1:19:48 pm – 1:19:49 pm)
running thread priority is:1
running thread priority is:10
```

Most Commonly Used Constructors in Thread Class

The Thread class includes constructors and methods for creating and operating on threads. Thread extends Object and implements the Runnable interface.

- Thread()

The default Thread() constructor is used to create a new Thread class.

```
package test;

public class ThreadExample extends Thread {

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

        System.out.println("Thread has been created with name :- " + t1.getName());
    }
}
```

Output

```
<terminated> ThreadExample [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (17-Mar-2021, 4:35:28 pm – 4:35:31 pm)
Thread has been created with name :- Thread-0
```

- Thread (String str)

A thread object is created and a name is provided to the same.


```
package test;
public class ThreadExample extends Thread {

    public static void main(String[] args) {
        Thread t1 = new Thread("Your Name");
        t1.start();

        System.out.println("Thread has been created with name :- " + t1.getName());
    }
}
```

Output

```
<terminated> ThreadExample [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (17-Mar-2021, 4:38:34 pm – 4:38:35 pm)
Thread has been created with name :- Your Name
```

- Thread (Runnable r)

In this constructor type, Runnable reference is passed and a new Thread object is created.

```
1 package test;
2 class Testing implements Runnable {
3
4     @Override
5     public void run() {
6         System.out.println("Do Something");
7     }
8 }
9
10 }
11 public class ThreadExample extends Thread {
12
13     public static void main(String[] args) {
14         Thread t1 = new Thread(new Testing());
15         t1.start();
16
17         System.out.println("Thread has been created with name :- " + t1.getName());
18     }
19 }
20
21 }
22
23 }
```

Output

```
<terminated> ThreadExample [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (17-Mar-2021, 4:43:57 pm – 4:43:58 pm)
Thread has been created with name :- Thread-0
```



```
Do Something
```

- Thread (Runnable r, String r)

We may use this constructor to generate a new Thread object by passing a Runnable reference as the first parameter and also providing a name for the newly generated thread.

```
package test;

class Book implements Runnable {

    @Override
    public void run() {
        System.out.println("Do a Task");
    }
}

public class ThreadExample extends Thread {

    public static void main(String[] args) {
        Thread t1 = new Thread(new Book(), "book thread");
        t1.start();

        System.out.println("Thread has been created with name :- " + t1.getName());
    }
}
```

Output

```
<terminated> ThreadExample [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (17-Mar-2021, 4:46:47 pm – 4:46:48 pm)
Thread has been created with name :- book thread
Do a Task
```

Master Core Java 8 Concepts, Java Servlet & More!

Java Certification Training

[ENROLL NOW](#)

Multithreading in Java

In Java, multithreading is the method of running two or more threads at the same time to maximize CPU utilization. As a result, it is often referred to as Concurrency in Java. Each thread runs in parallel with the others. Since several threads do not assign different memory areas, they conserve memory. Furthermore, switching between threads takes less time.

In Java, [multithreading](#) enhances program structure by making it simpler and easier to navigate. These generalized threads can be used in high-server media applications to easily change or enhance the configuration of these complex structures.

Here is an example of Multithreading in Java.

```
1 package test;
2 public class Example1 implements Runnable{
3
4     public static void main(String[] args) {
5         Thread Example1 = new Thread("Demo1");
6         Thread Example2 = new Thread("Demo2");
7         Example1.start();
8         Example2.start();
9         System.out.println("Thread names are following:");
10        System.out.println(Example1.getName());
11        System.out.println(Example2.getName());
12    }
13    @Override
14    public void run() {
15    }
16
17 }
18
19
```

Output



```
<terminated> Example1 [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (17-Mar-2021, 5:06:40 pm – 5:06:42 pm)
Thread names are following:
Demo1
Demo2
```

How to Handle Thread Deadlock

A deadlock is a situation in which two or more threads are stuck waiting for each other indefinitely. When several threads need the same locks but receive them in separate orders, a deadlock occurs. In a Java multithreaded program, a deadlock condition may occur because the `synchronized` keyword causes the executing thread to block while waiting for the lock, associated with the specified object.

To prevent deadlock, make sure that when you obtain several locks, you always acquire them in the same order across all threads. Here is an example of code which may result in a deadlock.

```
package demo;

public class Example {

    private final String name;

    public Example(String name){
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public synchronized void call(Example caller){
        System.out.println(this.getName() + " has asked to call me " + caller.getName());
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        caller.callMe(this);
    }

    public synchronized void callMe(Example caller){
        System.out.println(this.getName() + " has called me " + caller.getName());
    }

    public static void main(String[] args) {
        Example caller1 = new Example("caller-1");
        Example caller2 = new Example("caller-2");

        new Thread(new Runnable() {
            public void run() { caller1.call(caller2); }
        }).start();

        new Thread(new Runnable() {
            public void run() { caller2.call(caller1); }
        }).start();
    }
}
```

Output

```
Example [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe (19-Mar-2021, 1:09:06 pm)
caller-1 has asked to call me caller-2
caller-2 has asked to call me caller-1
```

This code will probably result in a deadlock for the two generated threads. The lock on object caller1 and caller2 are already owned by respective threads forcing the threads to wait for each other to unlock such locks. This would trigger a deadlock and prevent the callMe() function from being called.

Become job-ready and get complete job assistance by opting for the decade's hottest career option. Score your dream job in no time by enrolling in our [Full Stack Java Developer Job Guarantee Program Today!](#)

Java is undoubtedly one of the most popular and promising [programming languages](#) today. And in case you wish to master it, enroll in Simplilearn's [Java Certification Training Course](#). Start learning today and make a mark in the programming world today!

FAQs

1) What is a thread in Java?

A thread denotes a distinct path of execution within a program. Within the Java Virtual Machine, applications can maintain multiple concurrently operating threads of execution. Each individual thread is assigned a priority, further influencing their order of execution.

2) What is an example of thread in Java?

An illustrative example of Java threads can be observed by employing the 'Runnable Interface'.

3) What is threading in Java?

A thread signifies a path of execution within a program. Within the Java Virtual Machine, an application can host numerous concurrently running threads of execution. Each thread possesses a priority, with those having higher priority being given precedence over threads with lower priority.