Fundamental of Java Multi-threading

This article explains what threads are in Java. It elaborates on their uses and on how to implement them, while discussing their life cycle.

In a multi-tasking environment, having multiple threads helps the programmer to create efficient programs. In the context of Java, thread means two different things:

- An instance of *java.lang.Thread*
- A thread of execution

An instance of a thread is just an object like any other object in Java. It has variables and methods, and lives and dies on the heap, but a thread of execution is an individual process (a lightweight process) that has its own stack. In Java, there is one thread per stack or vice versa. Even if you don't create any thread in your program, there is a thread running in your program, by default, which is called the Main thread. The *main()* function of Java or the starting point of a Java program runs in the Main thread, or we can say that the Main thread controls the execution of the *main()* function of a Java program.

Why we need threads

In Java, a thread is required to fulfil the purpose of multi-tasking. When we have to perform multiple tasks simultaneously, we need multi-threading. There are two types of threads in Java:

- *User thread:* This is created by the Java programmer or user. Once all the user threads are complete, the JVM will shut down regardless of the state of any daemon threads.
- *Daemon thread:* This is created by the operating system. The JVM doesn't care about the completion of the daemon thread. The best example of a daemon thread is the garbage collector. The Java garbage collector is started by the JVM once any program completes its execution.

Thread implementation in Java

There are two ways to implement threads in Java.

Extending the thread class: We can create a thread

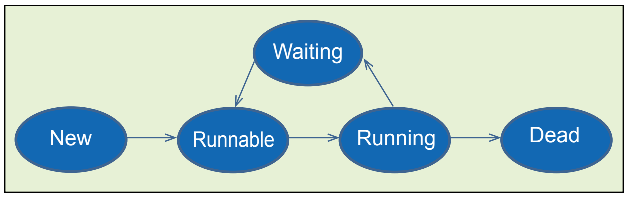

Figure 1: Life cycle of a thread

object in Java by extending the thread class of the *java.lang package*. For example:

```
class MyThread extends Thread{
public void run(){// line 2
System.out.println("Thread run method");// line 3

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

     MyThread t = new MyThread();// line 6
     t.start();// line 7

}
}
```

In the above code, at line number 6, we have created a thread object. The *run()* method in the above code is a pre-defined method and it is present inside the *java.lang.Thread* class. In line number 7, when we say *t.sart()*, it actually calls the *run()* method and hence the output of the program is *"Thread run method"* (line number 3).

Implementing the Runnable interface:

```
class MyRunnable implements Runnable{

     public void run(){// line 2
       System.out.println("Runnable run method");// line 3
     }

     public static void main(String [] args){

       MyRunnable r = new MyRunnable();// line 6
       Thread t = new Thread(r);// line 7
       t.start();// line 8
     }
}
```

In the above code, you can see how it differs from extending a thread class. Here, at line number 6, we have created an object of the *MyRunnable* class, which is not a thread object but just an object of a class that implements the Runnable interface; hence, the object becomes Runnable. Now, since the object 'r' is Runnable, we can pass this object to a thread class constructor at line number 7 and, finally, once the thread object is created at line number 7, we call the *start()* method to start the thread that eventually gives the call to the *run()* method. And the output is *"Runnable run method"* (line 3).

Note: We prefer implementing the Runnable interface over extending the thread class for thread implementation in Java, because the interface promotes multiple inheritance in Java.

Life cycle of a thread

Let us go over the above thread's life cycle diagram (Figure 1), step by step:

1. *New:* The thread reaches this when we create the object of the thread. A thread is not considered alive at this state.
2. *Runnable:* Runnable means the thread is eligible to run but until the scheduler selects the thread to run, it can't run. A thread enters into this state when the *start()* method is invoked. A thread can also enter into this stage after running and in the blocked state. A thread is considered alive in this state.
3. *Running:* This is the state where the thread is actually performing the task. The thread comes into this state when the CPU scheduler selects the thread and makes it run. There are several ways to get to the runnable state, but there is only one way to get into the running state.
4. *Waiting/Blocked/Sleeping:* In these states, the thread is not eligible to run but it is still alive. From this state, the thread may return to the runnable state and further, it may go to the running state. The thread comes into this state when we call the *sleep(),wait()* and *yield()* methods.
5. *Dead:* A thread comes in this state when the *run()* method finishes its execution. Once a thread is dead, it can't be started again; so if you try to invoke the *start()* method again, it will give you an exception.

Naming a thread

There are various ways to name a Java thread.

1. By using the thread class constructor:

```
public Thread(String)
public Thread(Runnable, String)

class MyThread extends Thread{

     public void run(){
       System.out.println(Thread.currentThread().getName());
     }

     public MyThread(String name){
       super(name);// line 6
     }
```

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

      MyThread t = new MyThread(“Vikas”);// line 9
      t.start();

    }

}
```

In the above code, we have used the thread class constructor that takes *java.lang.String* as a parameter (line number 9). This gives a call to the constructor at line number 6, which further sets the name to the super class thread constructor. The output of the program is *“Vikas”*.

```
class MyRunnable implements Runnable{

    public void run(){
      System.out.println(Thread.currentThread().getName());
    }

    public static void main(String[] args){

      MyRunnable r = new MyRunnable();
      Thread t = new Thread(r, “Vikas”);// line 7
      t.start();

    }

}
```

In the above code, we are trying to set the name of the thread at line number 7, which takes two parameters, *MyRunnable* object and the *java.lang.String* object. The output of the code is *“Vikas”*.

2. By the *setName()* method of the thread class:

```
Class MyThread extends Thread{

    public void run(){

      System.out.println(Thread.currentThread().getName());
    }

    Public static void main(String[] args){

      MyThread t = new MyThread();
      t.setName(“Vikas”);// line 7
      t.start();
    }
}
```

In the above code we are trying to set the name of the thread at line number 7 by using the *setName()* method of thread class. The output of the code is *“Vikas”*.

To summarise, a thread is a Java class that is present inside the *java.lang package*. There is one thread per call stack and vice versa. Every Java program has a thread running by default and it is called the Main thread, which controls the *main()* function of the Java program. There are two types of threads in Java — one is the user thread and the other is the daemon thread. The best example of a daemon thread is the garbage collector. The user thread is created by the programmer and when its execution is complete, the daemon thread is invoked by the JVM. Each Java thread has five life cycle states (Figure 1). A Java programmer can name a thread by using the thread class constructor and the *setName()* method of the thread class.

By: Vikas Kumar Gautam

The author is a mentor at Emertxe Information Technology (P) Ltd. His main areas of expertise include application development using Java/J2EE and Android for both Web and mobile devices. A Sun Certified Java Professional (SCJP), his interests include acquiring greater expertise in the application space by learning from the latest happenings in the industry. He can be reached at *vikash_kumar@emertxe.com*