Creating Multiple Threads in Java

Basically, when we need to perform several tasks at a time, we can create multiple threads to perform multiple tasks in a program.

Creating more than one thread to perform multiple tasks is called multithreading in Java. In multiple threading programming, multiple threads are executing simultaneously that improves the performance of CPU because CPU is not idle if other threads are waiting to get some resources.

Multiple threads share the same address space in the heap memory. Therefore, it is good to create multiple threads to execute multiple tasks rather than creating multiple processes.

Benefits of Creating multiple threads in Java for Multitasking

Suppose when we go to watch a movie in a theatre, generally, a person stays at door to check and cut the tickets. When we enter the hall, there is one more person that stays to show seats to us.

Suppose there is one person (1 thread) to perform these two tasks. In this case, he has to first cut ticket and then come along with us to show seats. Then, he will go back to door to cut second ticket and then again enter the hall to show seat for second ticket.

Like this, he is taking a lot of time to perform these tasks one by one. If theatre manager employs two persons (2 threads) to perform these two tasks, one person will cut the ticket and another person will show seat.

Thus, when the second person will be showing seat, the first person will cut the ticket. Like this, both persons will act simultaneously and wastage of time will not happen.

Let's create a Java program in which we will try to implement this realtime scenario. Look at the following source code.

```
// Two threads performing two tasks at a time.

public class MyThread extends Thread

{

// Declare a String variable to represent task.
```

```
  String task;
MyThread(String task)
{
  this.task = task;
}
public void run()
{
 for(int i = 1; i <= 5; i++)
 {
  System.out.println(task+ " : " +i);
  try
 {
    Thread.sleep(1000); // Pause the thread execution for 1000 milliseconds.
 }
 catch(InterruptedException ie) {
   System.out.println(ie.getMessage());
  }
 } // end of for loop.
} // end of run() method.
public static void main(String[] args)
{
// Create two objects to represent two tasks.
  MyThread th1 = new MyThread("Cut the ticket"); // Passing task as an
argument to its constructor.
  MyThread th2 = new MyThread("Show your seat number");
// Create two objects of Thread class and pass two objects as parameter to
constructor of Thread class.
  Thread t1 = new Thread(th1);
  Thread t2 = new Thread(th2);
  t1.start();
  t2.start();
  }
}
```

Output:

```
        Cut the ticket : 1
        Show your seat number : 1
        Cut the ticket : 2
        Show your seat number : 2
        Cut the ticket : 3
        Show your seat number : 3
        Cut the ticket : 4
        Show your seat number : 4
        Cut the ticket : 5
        Show your seat number : 5
```

Explanation:

1. In the preceding example, we have created two threads on two objects of MyThread class. Here, we created two objects to represent two tasks.

When we will run the above program, the main thread starts running immediately. Two threads will generate from the main thread that will perform two different tasks.

2. When t1.start(); is executed by JVM, it starts execution of code inside run() method and print the statement “Cut the ticket” on the console.

3. When JVM executes Thread.sleep(1000); inside the try block, it pauses the thread execution for 1000 milliseconds. Here. sleep() method is a static method that is used to pause the execution of thread for a specified amount of time.

For example, Thread.sleep(1000); will pause the execution of thread for 1000 milliseconds (1 sec). Since sleep() method can throw an exception named InterruptedException, we will catch it into catch block.

4. Meanwhile, JVM executes t2.start(); and second thread starts execution of code inside the run() method almost simultaneously. It will print the statement “Show your seat number”. Now, the second thread will undergo to sleep for 1000 milliseconds.

5. When the pause time period of the first thread is elapsed, it will reenter into running state and starts the execution of code inside run() method. The same process will also happen for second thread. In this manner, both threads will perform two tasks almost simultaneously.

Note:

1. In the above output, you will notice that the program is displaying two messages “Cut the ticket” and “Show your ticket” in shuffled manner.

This is because the CPU shares time between two threads. CPU executes the run() method of t1 for time slice and prints one message.

Meanwhile, CPU control goes to execute the run() method of t2 for next time slice and prints the second message. This process goes on.

2. The process of allotting time slots to threads is called time slice.

Therefore, we can say that thread behaviour is unpredictable in Java. When you will run code second time, you may get different output.

Multiple Threads acting on Single object

It is also possible to create two or more threads on a single object. Let’s create a Java program in which three threads will share the same object (same run() method).

Program:

```
public class MultipleThread1 implements Runnable{
   String task;
   MultipleThread1(String task)
   {
      this.task = task;
   }
   public void run()
   {
      for(int i = 1; i <= 5; i++)
      {
         System.out.println(task+ ":" +i);
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException e)
         {
            e.printStackTrace();
         }
      }
   }
```

```
  public static void main(String[] args)
  {
   System.out.println("Name of thread: " +Thread.currentThread().getName());
     // Multiple child threads acting on single object.
     MultipleThread1 mt = new MultipleThread1("Hello Java");
     Thread t1 = new Thread(mt);
     Thread t2 = new Thread(mt);
     Thread t3 = new Thread(mt);
     t1.start();
     t2.start();
     t3.start();
     int count = Thread.activeCount();
     System.out.println("No of active threads: " +count);
  }
}
```

Output:

```
   Name of thread: main
   No of active threads: 4
   Hello Java:1
   Hello Java:1
   Hello Java:1
   Hello Java:2
   Hello Java:2
   . . . . . .
   . . . . . .
   Hello Java:5
```

Explanation:

As you can observe that three threads are sharing the same run method. That is, three threads are doing the same task. The job of all three threads is to display “Hello Java” five times. The output may be unpredictable when you will run multiple times this program. But, the general pattern of the output will be the same.

Let’s take one more program where we will perform two different tasks by two different threads. Look at the following source code.

```
class MyThread1 implements Runnable
{
  public void run() // Entry point of Thread1
  {
    for(int i = 0; i < 5; i++)
    {
      System.out.println("First Child Thread: " +i);
    }
    System.out.println("\t First child existed");
  }
}
class MyThread2 implements Runnable
{
  public void run() // Entry point of Thread2
  {
    for(int i = 0; i < 5; i++)
    {
      System.out.println("Second Child Thread: " +i);
    }
```

```
        System.out.println("Second child existed");
    }
}
public class MyClass {
    public static void main(String[] args)
    {
        MyThread1 th1 = new MyThread1();
        Thread t1 = new Thread(th1);
        t1.start(); // Execution of first thread is started.
        MyThread2 th2 = new MyThread2();
        Thread t2 = new Thread(th2);
        t2.start(); // Execution of second thread is started.
        int j = 0;
        while(j < 4)
        {
            System.out.println("Main Thread: " +j);
            j = j + 1;
        }
        System.out.println("\t Main thread existing");
            }
}
```

Output:

```
       First Child Thread: 0
       First Child Thread: 1
       First Child Thread: 2
       First Child Thread: 3
```

```
First Child Thread: 4
First child existed
Main Thread: 0
Second Child Thread: 0
Main Thread: 1
Second Child Thread: 1
Main Thread: 2
Second Child Thread: 2
Main Thread: 3
Main thread existing
Second Child Thread: 3
Second Child Thread: 4
Second child existed
```

Explanation:

When you will execute the above program, the main thread immediately starts running. Two child threads generated from the main thread will perform two different tasks.

Now, three threads (Thread1, Thread2, and Main thread) will run independently altogether whenever CPU time is available to them. But there will be no specific order of execution. Hence, the program may give different output every time you run it.