Introduction

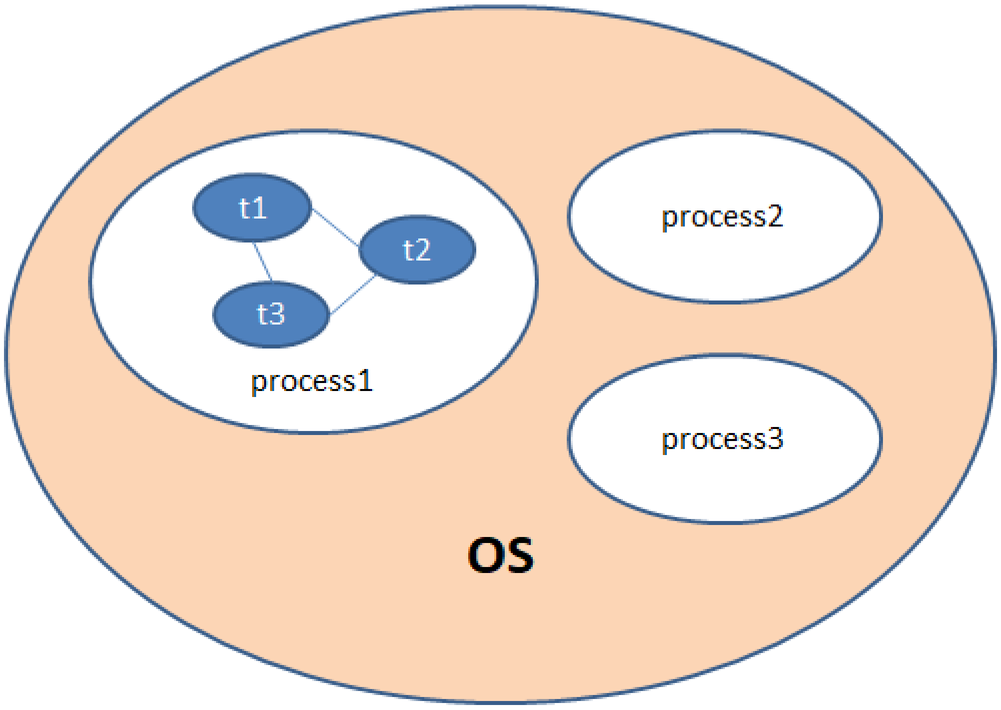
The process of executing multiple threads concurrently is called multithreading in Java. This article will explain Multithreading and how multithreading works in Java with code examples. In Java, threads use a shared memory location.

What is Thread in Java?

A thread is a lightweight small process or the smallest unit of the processing, which can run concurrently with the other threads of the same process.

Threads are independent because they all have a separate path of execution due to which if an exception occurs in one thread, it doesn't affect the execution of other threads. All the threads of a process share the common memory area. The process of executing multiple threads is called multithreading in Java.

Let's understand the figure, given below.



In the figure, a thread is executed inside the process and the context-switching between the threads. Here the multiple processes in the OS and the single process can have multiple threads but only one thread executed at a time.

Multithreading is a procedure of executing multiple threads at the same time. Multiprocessing and multithreading are used to achieve multitasking in Java. Saving memory and context-switching between the threads takes less time than the process. Multithreading is mainly used in games, animation, etc.

What are the Advantages of Multithreading?

- The threads are independent due to which we can perform multiple operations at the same time.
- We can perform many operations simultaneously, so it saves time.
- The Threads don't affect other threads if an exception occurs in a single thread.
- Better use of CPU resources.
- Decrease the cost of maintenance.
- Better use of cache storage by utilization of resources.

What are the Disadvantages of Multithreading?

- Multiple threads can interfere with each other when sharing hardware resources such as caches or transaction lookaside buffers(TLBs).
- The execution time of a single thread can be degraded, even when only one thread is executing.
- Complex debugging and testing process.
- Unpredictable results.

Why we use Multithreading in Java?

We use multithreading in Java for achieving Multitasking.

What is Multitasking?

Multitasking is the process of executing multiple tasks at the same time. We use multitasking to utilize the CPU.

Multitasking can be achieved in two ways. Which are:

Process-based Multitasking

Process-based multitasking means Multiprocessing, in this, every process has its own memory address and every process allocates a separate memory area.

The process is heavy-weight. The cost of communication between the process is very high. Switching from one process to another process requires some time to save and load the registers, memory maps, updates, lists, etc.

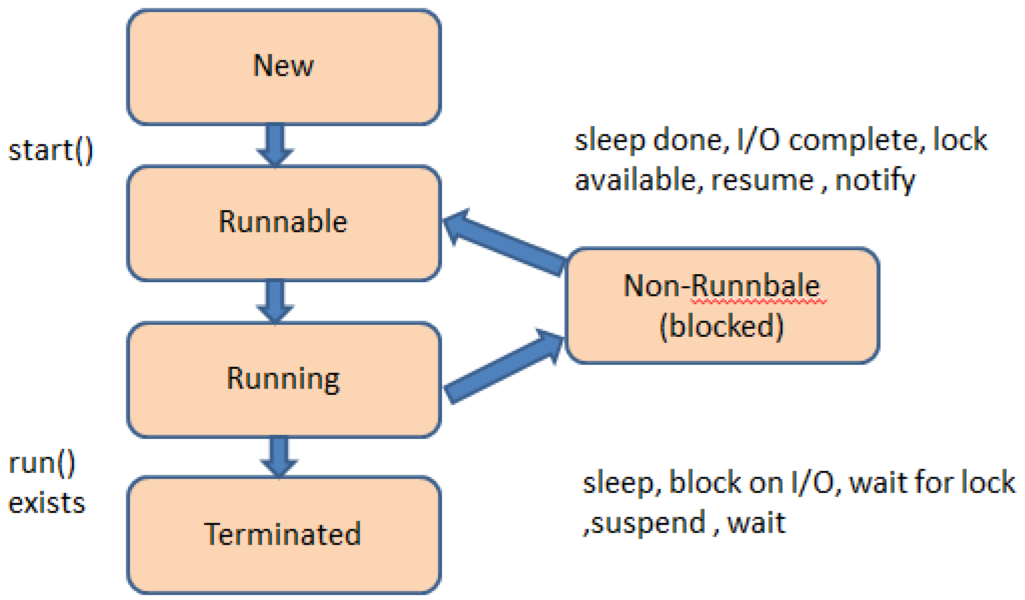
Thread-based Multitasking

Thread-based multitasking means Multithreading. in which all the threads share the same address space.

Thread is light-weight. The cost of communication between the threads is very low. At least one process is required for each thread.

What is thread life-cycle in Java?

A thread goes through many different stages in its life cycle. For example, a thread is born, thread start, thread run, and then thread die. The life cycle of the thread is controlled by JVM in Java. For a better understanding of the thread, we explain it in the five states.


New
start()
Runnable
sleep done, I/O complete, lock available, resume , notify
Non-Runnbale (blocked)
Running
run() exists
Terminated
sleep, block on I/O, wait for lock ,suspend , wait

New

A new state is the first state of the thread. The thread is in the new state if we create an object of the Thread class but before the invocation of start() method.

Runnable

After a new thread starts, the thread is in the second state after the invocation of the start() method, but the scheduler has not selected it to be in the running thread.

Running

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

Non-Runnable (Blocked)

The fourth state is non-runnable(blocked). This is the state when the thread is alive but is currently not eligible to run.

Terminated

The last state of a thread is terminated. A thread is in the terminated state when its run() method exits.

How threads are created in Java?

Threads are created in java in two ways, first by extending the Thread class and second by implementing the Runnable interface by any Java class.

How many ways create a thread in Java?

There are two ways to create a thread in Java multithreading.

1. By Extending Thread class
2. By implementing Runnable method

What is the Thread class in Multithreading?

Thread class provides many constructors and methods to create and perform many operations on a thread. Thread class extends an Objects class and implements Runnable interface in Java multithreading.

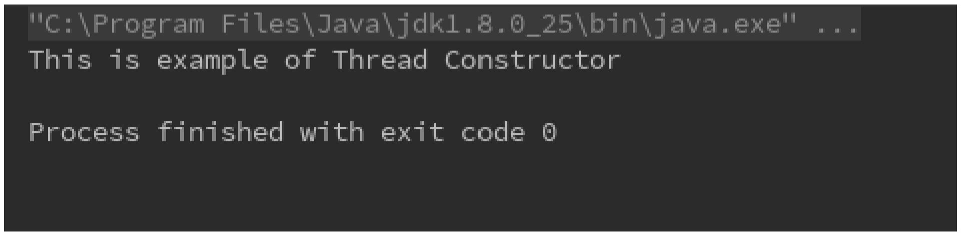
Constructors of Thread class

`1. Thread()`

Allocates a new Thread object. The complete program is listed below.

```java
public class ThreadConstructor extends Thread {
    public void run() {
        System.out.println("This is example of Thread Constructor");
    }

    public static void main(String args[]) {
        ThreadConstructor t = new ThreadConstructor();
        t.start();
    }
}
```

The output of the following code generates the following output.



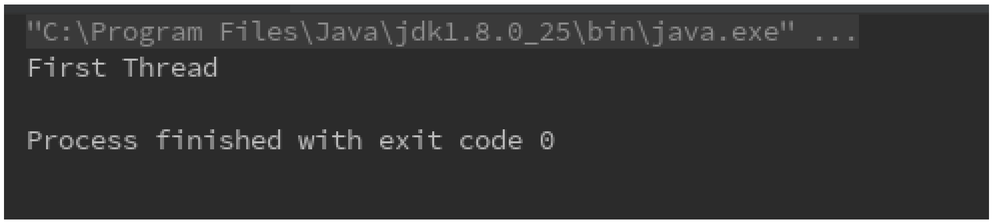
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
This is example of Thread Constructor

Process finished with exit code 0
```

```
2. Thread(String name)
```

Allocates a new Thread object. The complete program is listed below.

```java
public class ThreadConstructor1 extends Thread{
    String tname;

    ThreadConstructor1(String tname) {
        this.tname = tname;
    }

    public void run() {
        System.out.println(tname);
    }

    public static void main(String args[]) {
        ThreadConstructor1 t = new ThreadConstructor1("First Thread");
        t.start();
    }
}
```

The output of the following code generates the following output.



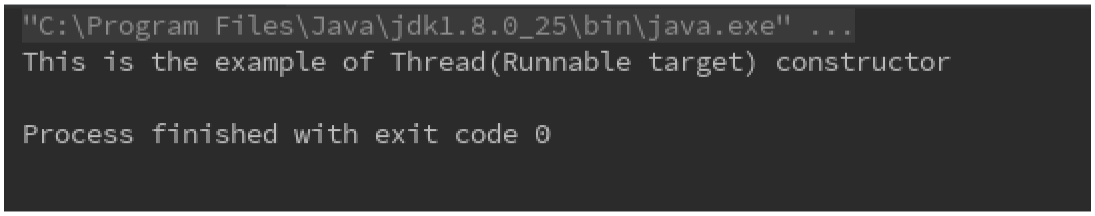
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
First Thread

Process finished with exit code 0
```

```
3. Thread(Runnable r)
```

Allocates a new Thread object. The complete program is listed below.

```java
public class ThreadConstructor2 implements Runnable {

    public void run() {
        System.out.println("This is the example of Thread(Runnable target)

    }

    public static void main(String args[]) {
        ThreadConstructor2 s = new ThreadConstructor2();
        Thread t = new Thread(s);
        Thread t = new Thread(s);
        t.start();
    }
}
```

The output of the following code generates the following output.



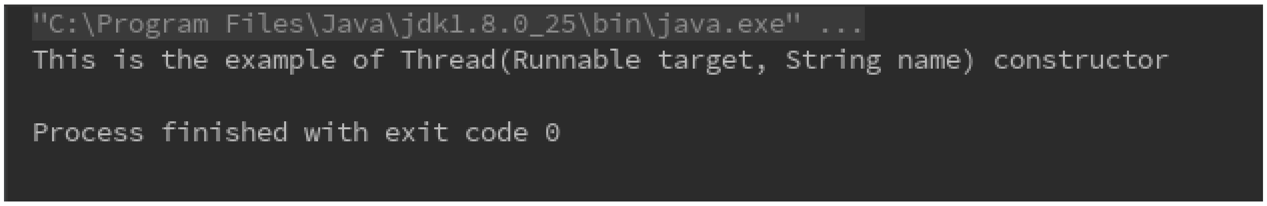
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
This is the example of Thread(Runnable target) constructor

Process finished with exit code 0
```

`4. Thread(Runnable r, String name)`

Allocates a new Thread object. The complete program is listed below.

```java
public class ThreadConstructor3 implements Runnable {

    ThreadConstructor3() {
    }

    public void run() {
        System.out.println("This is the example of Thread(Runnable target,
    }

    public static void main(String args[]) {
        ThreadConstructor3 s = new ThreadConstructor3();
        Thread t = new Thread(s, "CsharpCorner");

        t.start();
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
This is the example of Thread(Runnable target, String name) constructor

Process finished with exit code 0
```

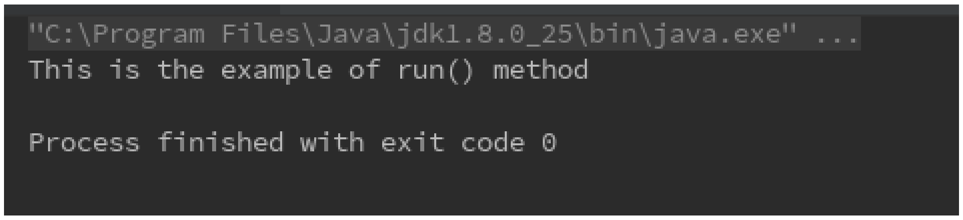
Methods of Thread class

`1. public void run()`

This method is used to perform an action for a thread. The complete program is listed below.

```java
public class ThreadRunMethod extends Thread {
    public void run() {
        System.out.println("This is the example of run() method");
    }

    public static void main(String args[]) {
        ThreadRunMethod t = new ThreadRunMethod();
        t.start();
    }
}
```

The output of the following code generates the following output.



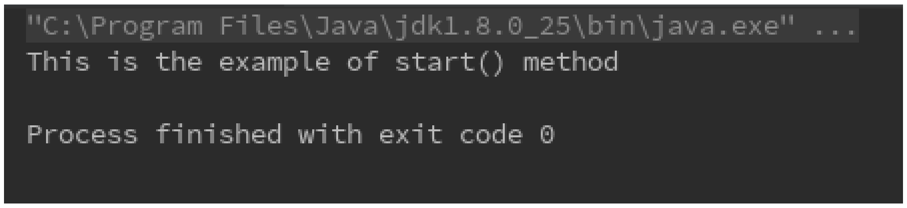
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
This is the example of run() method

Process finished with exit code 0
```

```
2. public void Thread.start()
```

This method is used to start the execution of the thread. JVM calls the run() method to start the thread. The complete program is listed below.

```java
public class ThreadStartMethod extends Thread {

    public void run() {
        System.out.println("This is the example of start() method");
    }

    public static void main(String args[]) {
        ThreadStartMethod s = new ThreadStartMethod();
        s.start();
    }
}
```

The output of the following code generates the following output.



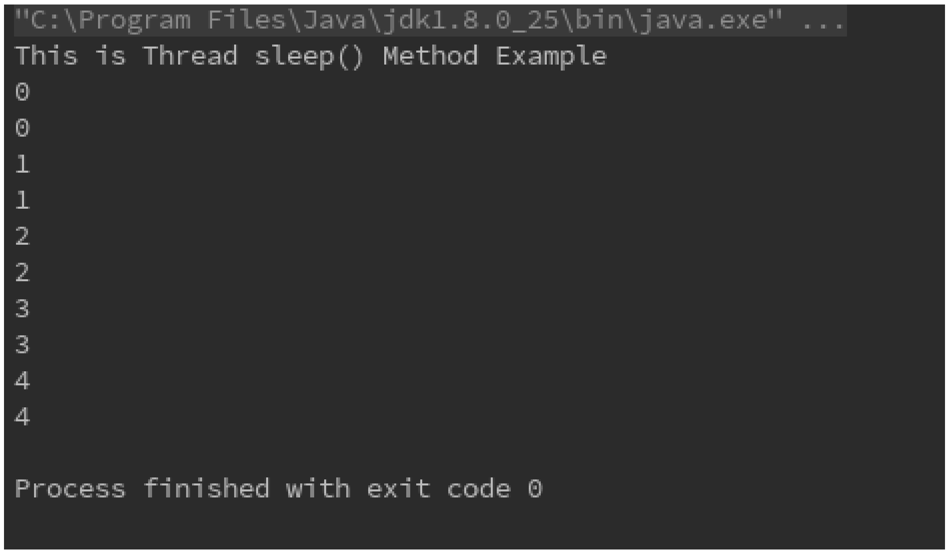
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
This is the example of start() method

Process finished with exit code 0
```

```
3. public void Thread.sleep(long milliseconds)
```

The Thread class sleep() method is used to sleep a thread for a particular duration of time. If one thread sleeps for the specified time then the thread scheduler picks up other threads and so on. The complete program is listed below.

```java
public class ThreadSleepMethod extends Thread {

    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }

    public static void main(String args[]) {
        ThreadSleepMethod t1 = new ThreadSleepMethod();
        ThreadSleepMethod t2 = new ThreadSleepMethod();
        t1.start();
        t2.start();
    }
}
```

The output of the following code generates the following output.



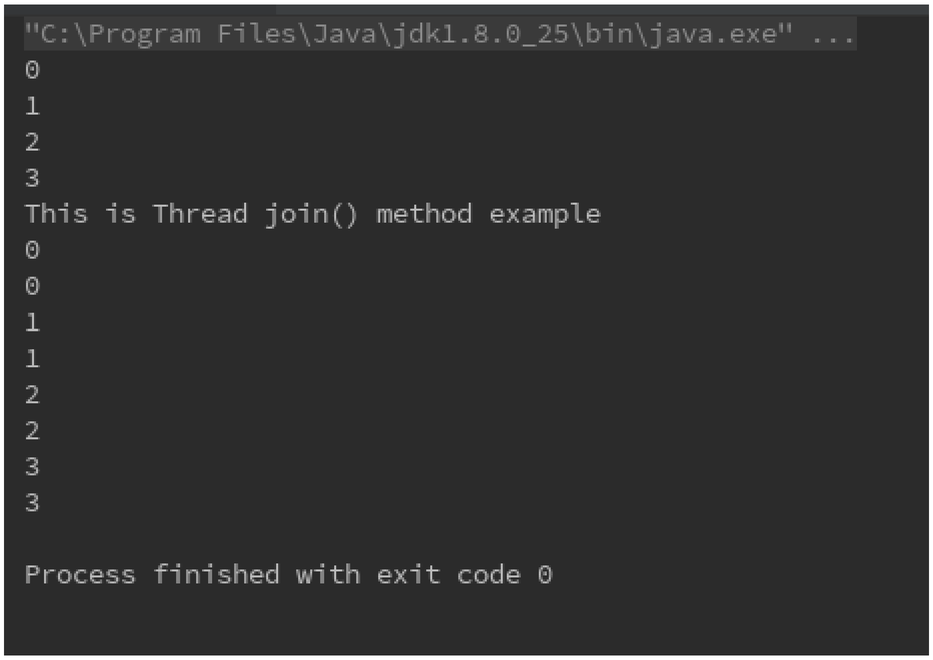
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
This is Thread sleep() Method Example
0
0
1
1
2
2
3
3
4
4

Process finished with exit code 0
```

`4. public void Thread.join()`

The join method is used to wait for a thread to die. The complete program is listed below.

```java
public class ThreadJoinMethod1 extends Thread {

    public void run() {
        for (int i = 0; i <= 3; i++) {
            try {
                Thread.sleep(1000);
            } catch (Exception e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }

    public static void main(String args[]) {
        ThreadJoinMethod1 t1 = new ThreadJoinMethod1();
        ThreadJoinMethod1 t2 = new ThreadJoinMethod1();
        ThreadJoinMethod1 t3 = new ThreadJoinMethod1();
        t1.start();
        try {
            t1.join();
        } catch (Exception e) {
            System.out.println(e);
        }

        t2.start();
        t3.start();

        System.out.println("This is Thread join() method example");
    }
}
```

The output of the following code is listed below.



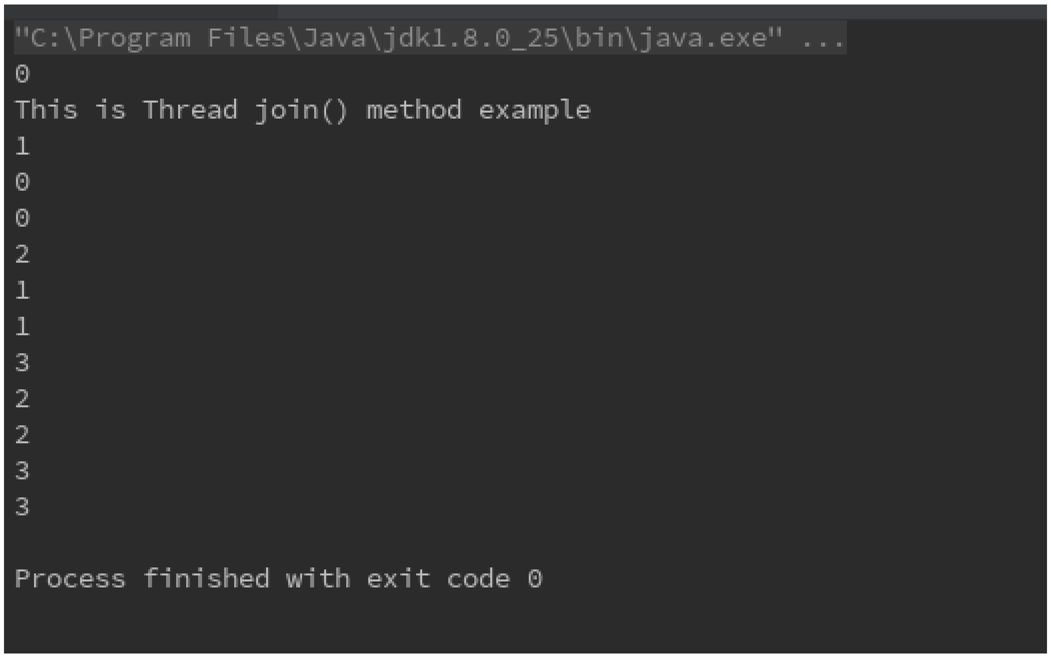
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
0
1
2
3
This is Thread join() method example
0
0
1
1
2
2
3
3

Process finished with exit code 0
```

`5. public void Thread.join(long milliseconds)`

This method is used to waits for a thread to die for a specified millisecond. The complete program is listed below.

```java
public class ThreadJoinMethod2 extends Thread {

    public void run() {
        for (int i = 0; i <= 3; i++) {
            try {
                Thread.sleep(700);
            } catch (Exception e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }

    public static void main(String args[]) {
        ThreadJoinMethod2 t1 = new ThreadJoinMethod2();
        ThreadJoinMethod2 t2 = new ThreadJoinMethod2();
        ThreadJoinMethod2 t3 = new ThreadJoinMethod2();
        t1.start();
        try {
            t1.join(1000);
        } catch (Exception e) {
            System.out.println(e);
        }
        t2.start();
        t3.start();
        System.out.println("This is Thread join() method example");

    }
}
```

The output of the following code generated the following output.



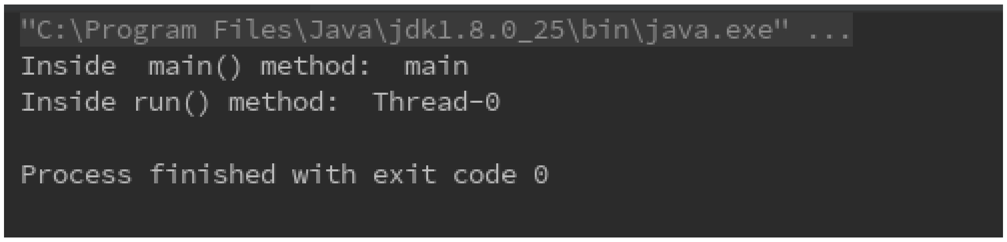
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
0
This is Thread join() method example
1
0
0
2
1
1
3
2
2
3
3

Process finished with exit code 0
```

```
6. public void Thread.currentThread()
```

The thread class static method currentThread() returns the reference of the Thread object that calls this method. The complete program is listed below.

```java
public class ThreadCurrentMethod extends Thread {

    @Override
    public void run() {
      Thread t = Thread.currentThread();
      String threadName = t.getName();
      System.out.println("Inside run() method:  " + threadName);
    }

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

      Thread t = Thread.currentThread();
      String threadName = t.getName();
      System.out.println("Inside  main() method:  " + threadName);
    }
  }
```

The output of the following code generates the following output.



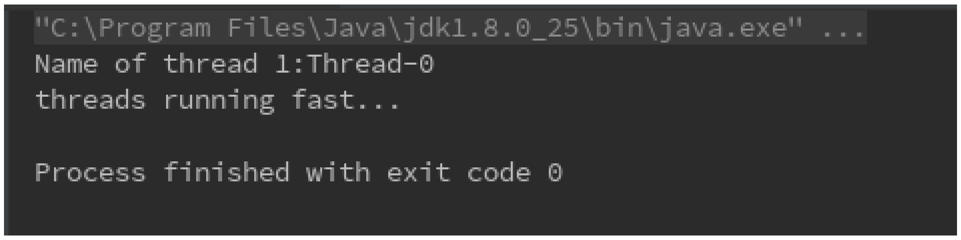
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Inside  main() method:  main
Inside run() method:  Thread-0

Process finished with exit code 0
```

`7. public String Thread.getName()`

This method is used to return the name of the thread. The complete program is listed below.

```java
public class ThreadGetnameMethod extends Thread {

  public void run() {
    System.out.println("threads running fast...");
  }

  public static void main(String args[]) {
    ThreadGetnameMethod t1 = new ThreadGetnameMethod();

    System.out.println("Name of thread 1:" + t1.getName());
    t1.start();

  }
}
```

The output of the following code generates the following output.



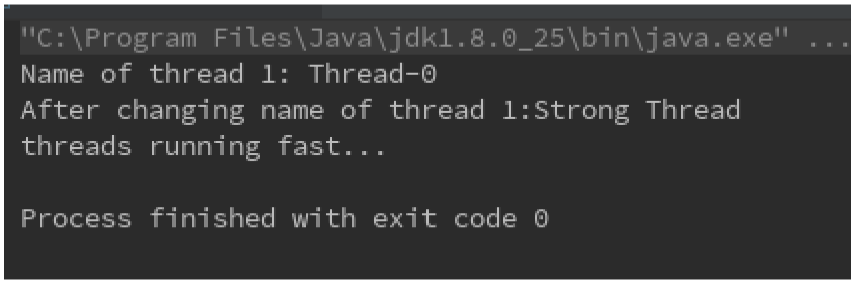
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Name of thread 1:Thread-0
threads running fast...

Process finished with exit code 0
```

```
8. public String Thread.setName(String name)
```

This method is used to change the name of the thread. The complete program is listed below.

```java
public class ThreadSetnameMethod extends Thread {

    public void run() {
        System.out.println("threads running fast...");
    }

    public static void main(String args[]) {
        ThreadSetnameMethod t1 = new ThreadSetnameMethod();
        System.out.println("Name of thread 1: " + t1.getName());
        t1.start();

        t1.setName("Strong Thread");
        System.out.println("After changing name of thread 1:" + t1.getNam
    }
}
```

The output of the code generates the following output.



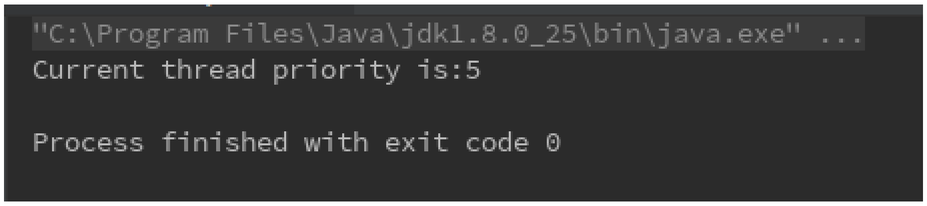
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Name of thread 1: Thread-0
After changing name of thread 1:Strong Thread
threads running fast...

Process finished with exit code 0
```

```
9. public int Thread.currentThread().getPriority()
```

This method is used to return the priority of the thread. The complete program is listed below.

```java
public class ThreadGetPriorityMethod extends Thread {

    public void run() {

        System.out.println("Current thread priority is:" + Thread.currentT
        System.out.println("Current thread priority is:" + Thread.current
    }

    public static void main(String args[]) {
        ThreadGetPriorityMethod g = new ThreadGetPriorityMethod();
        g.start();
    }
}
```

The output of the following code generates the following output.



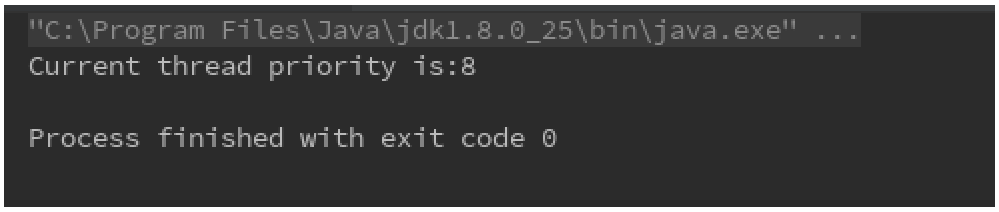
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Current thread priority is:5

Process finished with exit code 0
```

`10. public void Thread.currentThread().setPriority(int priority)`

This method is used to change the priority of the thread. The complete program is listed below.

```java
public class ThreadSetPriorityMethiod extends Thread {

    public void run() {
        System.out.println("Current thread priority is:" + Thread.current
    }

    public static void main(String args[]) {
        ThreadSetPriorityMethiod s = new ThreadSetPriorityMethiod();
        s.setPriority(8);
        s.start();
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Current thread priority is:8

Process finished with exit code 0
```

What is the Runnable interface in Java?

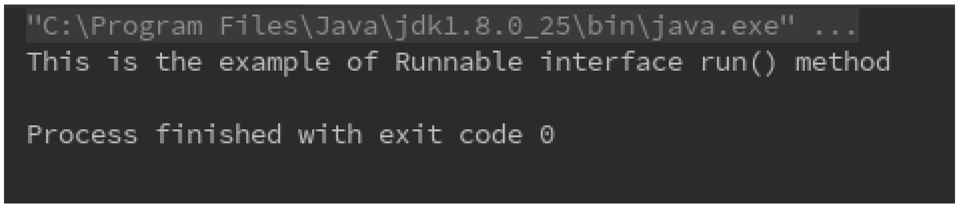
The runnable interface must be implemented by any class, whose instances are intended to be executed by a thread. The Runnable interface has only one run() method.

```
public void run()
```

This method is used to perform an action for a thread. The complete program is listed below.

```
public class RunnableRunMethod implements Runnable {

    @Override
    public void run() {
        System.out.println("This is the example of Runnable interface run
    }

    public static void main(String[] args) {
        RunnableRunMethod m = new RunnableRunMethod();
        Thread t = new Thread(m);
        t.start();
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
This is the example of Runnable interface run() method

Process finished with exit code 0
```

How to start a thread in Java?

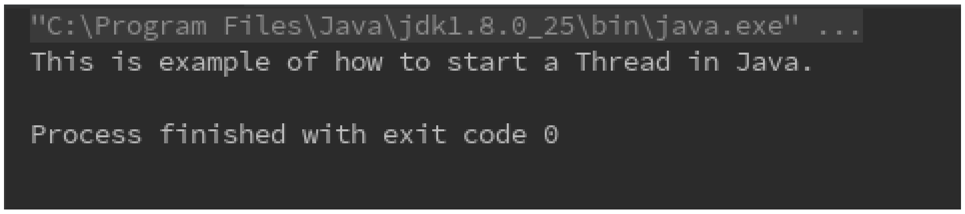
start() method is used to start a newly created thread. It performs may tasks, Which are-

- A new thread start.
- The thread shifts from the new state to the runnable state.
- If the thread gets a chance to execute, its target run() method will run.

The complete program is listed below.

```
public class ThreadExample extends Thread {

    public void run() {
        System.out.println("This is example of how to start a Thread in J
    }

    public static void main(String args[]) {
        ThreadExample t = new ThreadExample();
        t.start();
    }
}
```

The output of the following code generates the following output.



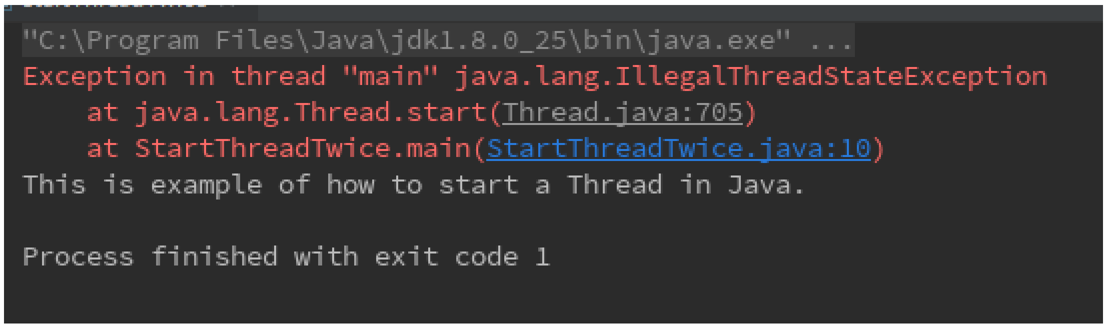
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
This is example of how to start a Thread in Java.

Process finished with exit code 0
```

Can We start a thread twice?

No, we cannot start a thread twice, after starting a thread, it can never be started again. If we do then all IllegalThreadStateException is thrown. In this case, the thread will run once but next time, it will throw an exception. The complete program is listed below.

```
01. public class StartThreadTwice extends Thread {
02.
03.     public void run() {
04.         System.out.println("we can not start a thread twice in a program"
05.     }
06.
07.     public static void main(String args[]) {
08.         ThreadExample t1 = new ThreadExample();
09.         t1.start();
10.         t1.start();  // Exception
11.     }
12. }
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Thread.java:705)
    at StartThreadTwice.main(StartThreadTwice.java:10)
This is example of how to start a Thread in Java.

Process finished with exit code 1
```

What is the Thread scheduler in Java?

The thread scheduler is the part of JVM, which decides which thread should run. There is no assurance that which runnable thread will be chosen to run by the scheduler. Any thread in the runnable state can be chosen by the scheduler to be the one and only running thread and if a thread is not in a runnable state, then it cannot be chosen to be the current running thread. Some methods can influence the scheduler to some extent and we can't control the behavior of the thread scheduler. Only one thread at a time can run in a single process in Java.

The thread scheduler generally uses preemptive scheduling or time-slicing scheduling to schedule the threads in Java.

Difference between Preemptive Scheduling and Time-slicing scheduling

Preemptive Scheduling	Time-slicing Scheduling
Under the preemptive scheduling, the highest priority tasks perform until it enters the waiting or dead state or a higher priority task comes into existence.	Under the time-slicing scheduling, a task performs for a predefined a slice of time and then reenters the pool of ready tasks. The thread scheduler then decides which task should execute next, based on the priority and other factors.

A priority of Thread in Java

Thread priorities are represented in the number of 1 to 10. Thread scheduler schedules the threads, according to their priority, which is referred to as preemptive scheduling. There is no guarantee because it fully depends on JVM specification that which scheduling, it chooses.

Each thread has a priority.

Three constants are defined in Thread class,

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

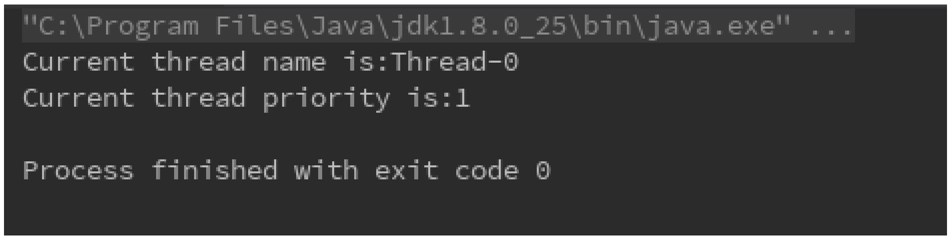
By default thread priority is NORM_PRIORITY(5)

The value of MIN_PRORITY is 0 and MAX_PRIORITY is 10.

The complete program is given below.

```java
public class ThreadPriority extends Thread {
    public void run() {
        System.out.println("Current thread name is:" + Thread.currentThre
        System.out.println("Current thread priority is:" + Thread.current
    }

    public static void main(String args[]) {
        ThreadPriority p1 = new ThreadPriority();
        p1.setPriority(Thread.MIN_PRIORITY);
        p1.start();
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Current thread name is:Thread-0
Current thread priority is:1

Process finished with exit code 0
```

What is Thread Pool in Java?

Thread Pool is a group of worker threads, which are waiting for the job and it can be reused many times.

A group of fixed size threads are created in a thread pool. A thread from the thread pool is selected and is assigned a job by the Service provider. After completion of his job, the thread is contained in the thread pool again.

What are the usages of the thread pool in Java?

Thread pool mainly used in JSP and Servlets in Java, which web or Servlets container creates a thread pool to process the request.

It saves time and better performance because there is no need to create a new thread. The complete program is listed below.

Step 1

First, we create a ThreadPoolExample Java class and implements the Runnable interface.

```
class ThreadPoolExample implements Runnable {
    private String thread;

    public ThreadPoolExample(String t) {
        this.thread = t;
    }

    public void run() {
        System.out.println(Thread.currentThread().getName() + " Start thre
        processmessage();//sleeps the thread for 2 seconds
        System.out.println(Thread.currentThread().getName() + " End ");//
    }

    private void processmessage() {
        try {
```

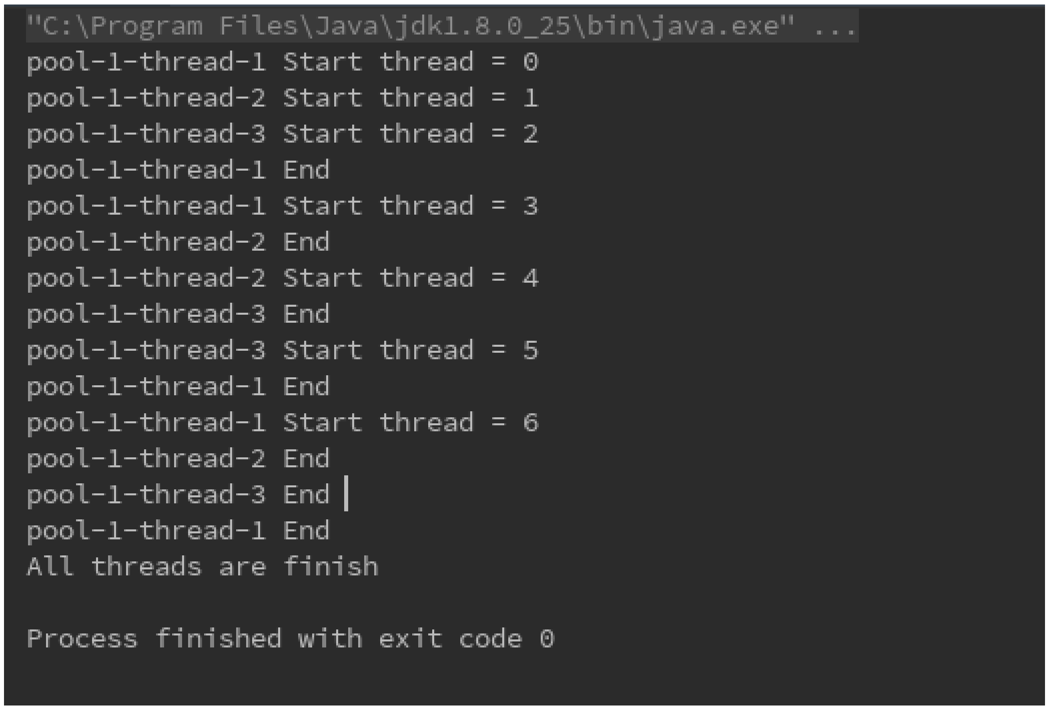
```java
            Thread.sleep(1500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Step 2

Second we create the ThreadPoolTest Java class for tesing the working of the threadpool.

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolTest {
    public static void main(String[] args) {
        ExecutorService e = Executors.newFixedThreadPool(3);//creating a
        for (int i = 0; i <= 6; i++) {
            Runnable t = new ThreadPoolExample("" + i);
            e.execute(t);
        }
        e.shutdown();
        while (!e.isTerminated()) {
        }
        System.out.println("All threads are finish");
    }
}
```

The output of the code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
pool-1-thread-1 Start thread = 0
pool-1-thread-2 Start thread = 1
pool-1-thread-3 Start thread = 2
pool-1-thread-1 End
pool-1-thread-1 Start thread = 3
pool-1-thread-2 End
pool-1-thread-2 Start thread = 4
pool-1-thread-3 End
pool-1-thread-3 Start thread = 5
pool-1-thread-1 End
pool-1-thread-1 Start thread = 6
pool-1-thread-2 End
pool-1-thread-3 End
pool-1-thread-1 End
All threads are finish

Process finished with exit code 0
```

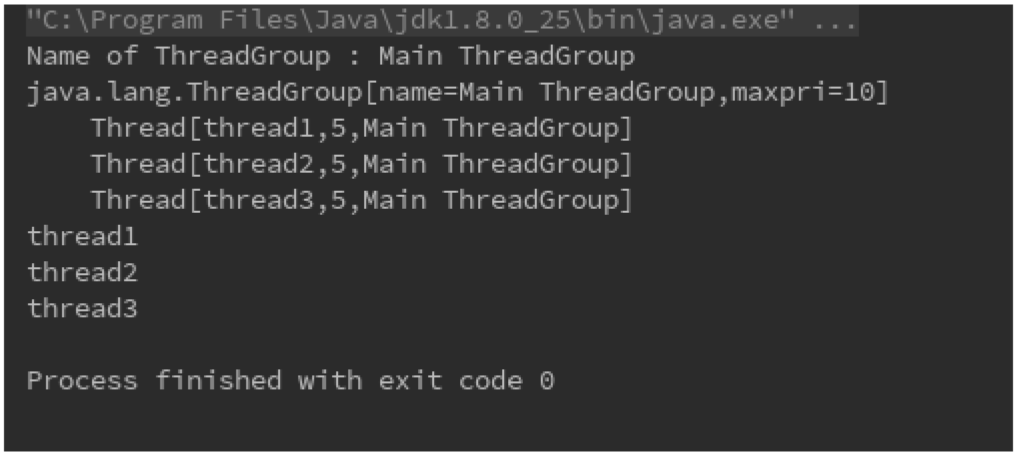
What is the ThreadGroup class in Java?

A group of multiple threads in a single object is called a ThreadGroup class in Java. With the help of ThreadGroup class, we can suspend, resume or interrupt a group of threads with a single method call.

ThreadGroup is implemented by *java.lang.ThreadGroup* class. The complete program is listed below.

```java
public class ThreadGroupExample implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        ThreadGroupExample obj = new ThreadGroupExample();
        ThreadGroup tg = new ThreadGroup("Main ThreadGroup");
        Thread t1 = new Thread(tg, obj, "thread1");
        t1.start();
        Thread t2 = new Thread(tg, obj, "thread2");
        t2.start();
        Thread t3 = new Thread(tg, obj, "thread3");
        t3.start();
        System.out.println("Name of ThreadGroup : " + tg.getName());
        tg.list();
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Name of ThreadGroup : Main ThreadGroup
java.lang.ThreadGroup[name=Main ThreadGroup,maxpri=10]
    Thread[thread1,5,Main ThreadGroup]
    Thread[thread2,5,Main ThreadGroup]
    Thread[thread3,5,Main ThreadGroup]
thread1
thread2
thread3

Process finished with exit code 0
```

Note

In this example, we create three threads that belong to one group. Here, tg is the ThreadGroup name, The ThreadGroupExample class implements Runnable interface and thread1, thread2, thread3 are the threads name.

What is Daemon Thread in Java?

Daemon thread is a service provider, which provides the services to the user thread. It is a low priority thread, which runs in the background to perform the tasks. Its life depends on the mercy of the user threads, i.e When all the threads die, JVM automatically terminates this thread.

There are many Java daemon threads running automatically such as garbage collection(gc) finalizer etc.

Some important points for Daemon Thread,

- It provides the services to the user thread for the background supporting tasks. It has no role in life than to serve the user thread.
- Its life fully depends on user threads.
- It is a low priority thread.

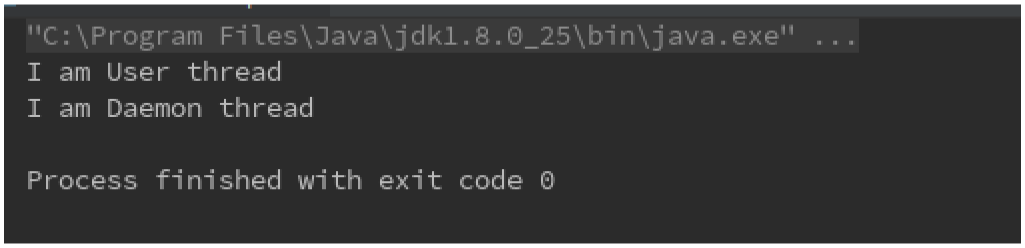
Why JVM terminates the Daemon thread if there is no user thread?

The main purpose of the daemon thread is providing the services to the user threads for the background supporting tasks due to which if there is no user thread, JVM terminates Daemon thread.

The complete program of a daemon thread is listed below.

```
public class DaemonThreadExample extends Thread {
        public void run() {
            if (Thread.currentThread().isDaemon()) {//checking for a daemo
                System.out.println("I am Daemon thread");
            } else {
                System.out.println("I am User thread");
            }
        }

        public static void main(String[] args) {
            DaemonThreadExample t1 = new DaemonThreadExample();
            DaemonThreadExample t2 = new DaemonThreadExample();
            t1.setDaemon(true);//it's a daemon thread
            t1.start();
            t2.start();
        }
    }
```

The output of the code generates the following output.

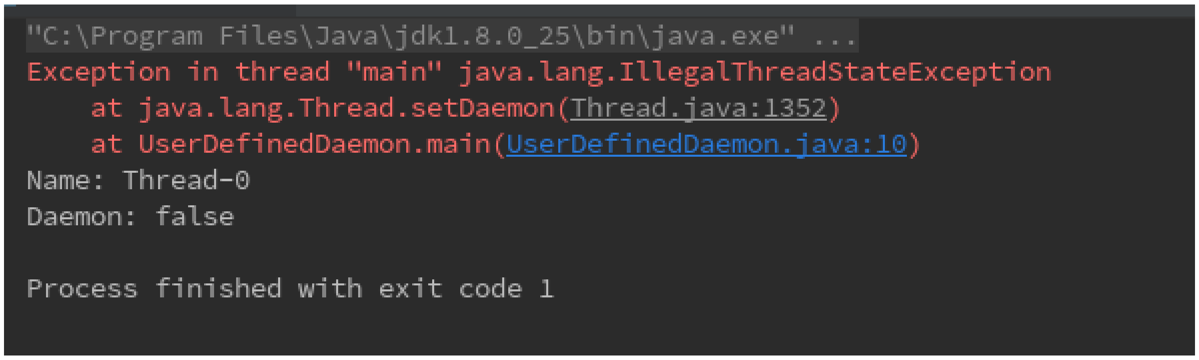


```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
I am User thread
I am Daemon thread

Process finished with exit code 0
```

If we want to create a User-defined Daemon thread, it should not be started, else it will throw an exception IllegalThreadStateException. The complete program of User-defined Daemon thread is listed below.

```java
public class UserDefinedDaemon extends Thread {
    public void run() {
        System.out.println("Name: " + Thread.currentThread().getName());
        System.out.println("Daemon: " + Thread.currentThread().isDaemon()
    }

    public static void main(String[] args) {
        UserDefinedDaemon t1 = new UserDefinedDaemon();
        t1.start();
        t1.setDaemon(true);//throw exception
    }
}
```

The output of the following code generates the following output.



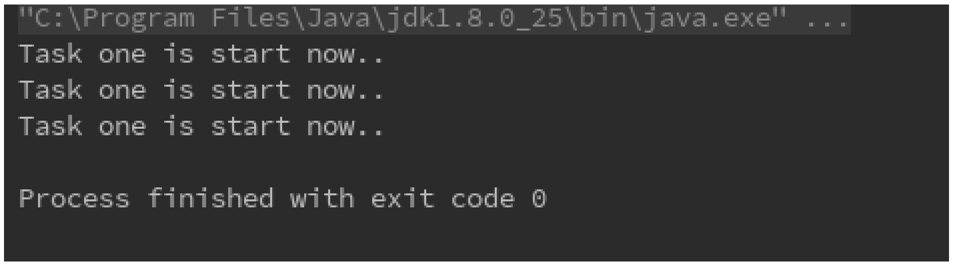
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.setDaemon(Thread.java:1352)
    at UserDefinedDaemon.main(UserDefinedDaemon.java:10)
Name: Thread-0
Daemon: false

Process finished with exit code 1
```

How to perform a single task by multiple threads?

If we want to perform a single task by many threads, we need to use only one run() method. The complete program is listed below.

```java
public class MultipleThreadsingleTask extends Thread{
    public void run() {
        System.out.println("Task one is start now..");
    }

    public static void main(String args[]) {
        MultipleThreadsingleTask mt1 = new MultipleThreadsingleTask();
        MultipleThreadsingleTask mt2 = new MultipleThreadsingleTask();
        MultipleThreadsingleTask mt3 = new MultipleThreadsingleTask();
        mt1.start();
        mt2.start();
        mt3.start();
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Task one is start now..
Task one is start now..
Task one is start now..

Process finished with exit code 0
```

Note

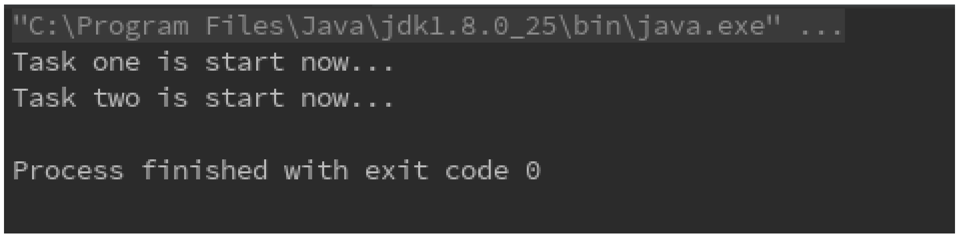
In the example, we created MultipleThreads class extends Thread class and performs a single task through the multiple threads.

How to perform multiple tasks by multiple threads?

If we want to perform multiple tasks by multiple threads, we need to use multiple run() method. The complete program is listed below.

```java
class Multi1 extends Thread {

    public void run() {
        System.out.println("Task one is start now...");
    }
}

class Multi2 extends Thread {
    public void run() {
        System.out.println("Task two is start now...");
    }
}

public class MultipleThreadsMultipleTask {
    public static void main(String args[]) {
        Multi1 t1 = new Multi1();
        Multi2 t2 = new Multi2();

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

The output of the following code generates the following output.



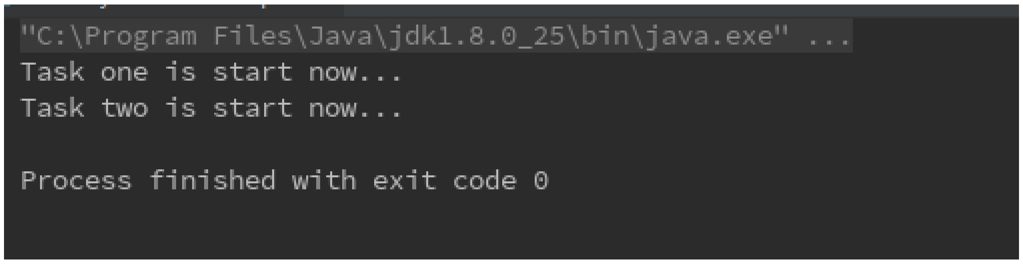
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Task one is start now...
Task two is start now...

Process finished with exit code 0
```

How to use Anonymous class with Runnable interface?

The classes which have no name are called Anonymous classes in Java. We can declare and initiate them at the same time. The complete program is listed below.

```
public class AnonymousClassExample {

    public static void main(String args[]) {

        Runnable r1 = new Runnable() {

            public void run() {
                System.out.println("Task one is start now...");
            }
        };
        Runnable r2 = new Runnable() {

            public void run() {
                System.out.println("Task two is start now...");
            }
        };

        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

The output of the following code generates the following output.



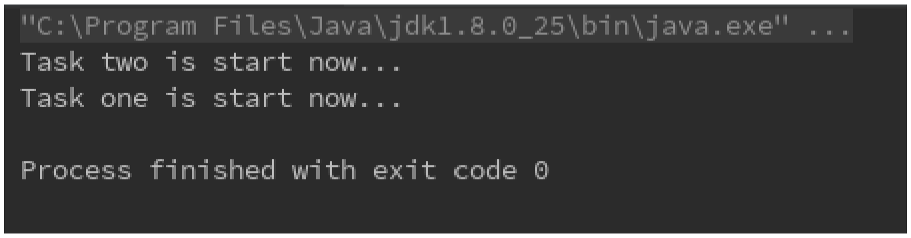
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Task one is start now...
Task two is start now...

Process finished with exit code 0
```

How to use Anonymous class that extends Thread class?

The complete program of the anonymous class that extends Thread class in Java.

```java
public class AnonymousClassExample {

    public static void main(String args[]) {

        Thread t1 = new Thread() {

            public void run() {
                System.out.println("Task one is start now...");
            }
        };
        Thread t2 = new Thread() {

            public void run() {
                System.out.println("Task two is start now...");
            }
        };

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

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Task two is start now...
Task one is start now...

Process finished with exit code 0
```

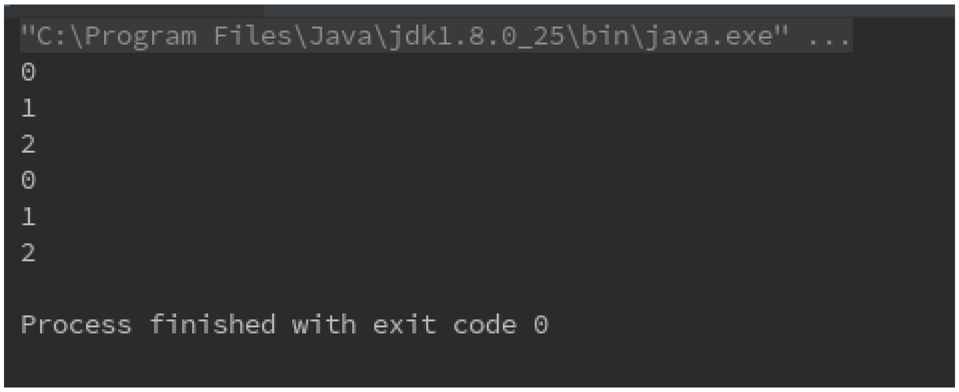
What happens when we call run() method instead of start() method in multithreading?

Every thread starts in a separate call stack. Calling the run() method from the main thread, followed by the run() method goes on the present call stack rather than at the beginning of the new call stack.

There will be no context-switching between the threads. The complete program is listed below.

```java
public class RunInsteadOfStart extends Thread {
    public void run() {

        for (int i = 0; i < 3; i++) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }

    public static void main(String args[]) {

        RunInsteadOfStart t1 = new RunInsteadOfStart();
        RunInsteadOfStart t2 = new RunInsteadOfStart();
        t1.run();
        t2.run();
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
0
1
2
0
1
2

Process finished with exit code 0
```

Note

In this example, we can see that there is no context-switching because here t1 and t2 will treat as a normal object and not a thread object.

What is Thread Synchronization?

In Java, synchronization is the capacity to control the access of multiple threads to any shared resources. It is a good option when we want to allow one thread to access the shared resources.

There are 2 types of Thread synchronization.

1) Mutual exclusive

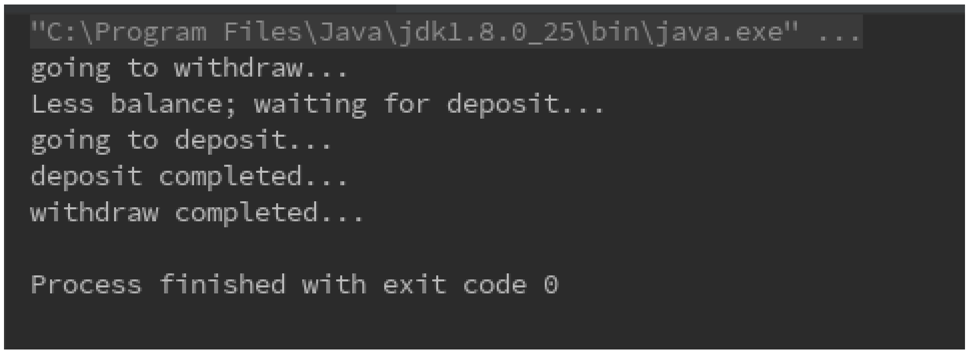
Mutual exclusive helps to keep the threads from a interfere with one another as a sharing data. This can be done in three ways.

`a. By synchronized method`

When we create any method as synchronized, it is called a synchronized method. The synchronized method locks an object for any shared resources. When a thread calls a synchronized method, it automatically gets the lock for that object and frees it when the thread completes its task. The complete program is listed below.

```java
class Customer {
    int amount = 10000;

    synchronized void withdraw(int amount) {
        System.out.println("going to withdraw...");  //Synchronized
Method

        if (this.amount < amount) {
            System.out.println("Less balance; waiting for deposit...");
            try {
                wait();
            } catch (Exception e) {

            }
        }
        this.amount -= amount;
        System.out.println("withdraw completed...");
    }

    synchronized void deposit(int amount) {
        System.out.println("going to deposit...");
        this.amount += amount;
        System.out.println("deposit completed... ");
        notify();
    }
}

public class SynchronizedMethodExample {
    public static void main(String args[]) {
        final Customer c = new Customer();

        new Thread() {
            public void run() {
                c.withdraw(15000);
            }
        }.start();

        new Thread() {
            public void run() {
                c.deposit(10000);
            }
        }.start();

    }
}
```

The output of the following code generates the following output.



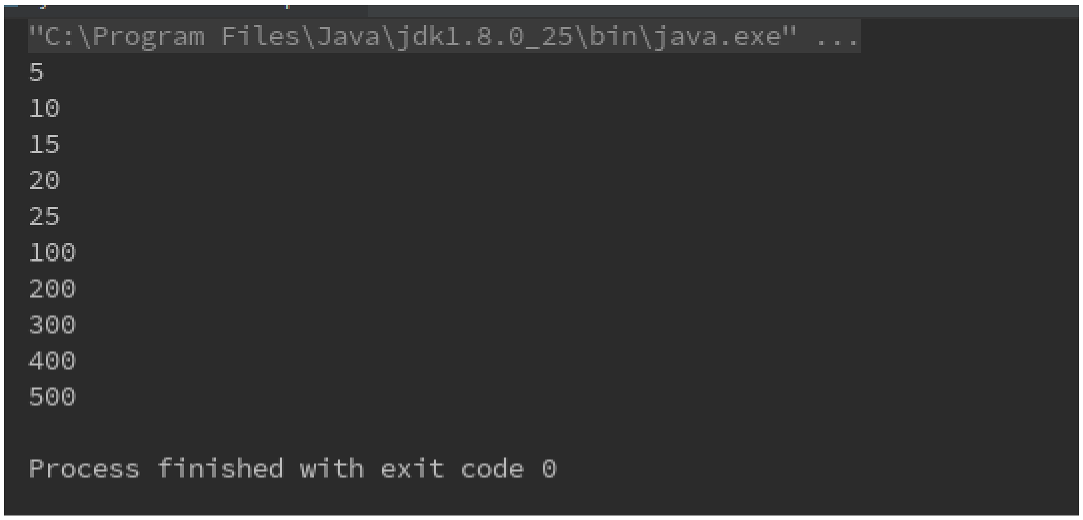
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed...

Process finished with exit code 0
```

`b. By synchronized block`

A block in java is a group of one or more statements enclosed in braces. when we used a synchronized keyword with a block, it is called a synchronized block. The complete program is listed below.

```java
class Table {

    void printTable(int n) {
        synchronized (this) {//synchronized block
            for (int i = 1; i <= 5; i++) {
                System.out.println(n * i);
                try {
                    Thread.sleep(400);
                } catch (Exception e) {
                    System.out.println(e);
                }
            }
        }
    }//end of the method
}

class MyThreadClass1 extends Thread {
    Table t;

    MyThreadClass1(Table t) {
        this.t = t;
    }

    public void run() {
        t.printTable(5);
    }

}

class MyThreadClass2 extends Thread {
    Table t;

    MyThreadClass2(Table t) {
        this.t = t;
    }

    public void run() {
        t.printTable(100);
    }
}

public class SynchronizedBlockExample {

    public static void main(String args[]) {
        Table obj = new Table();//only one object
        MyThreadClass1 t1 = new MyThreadClass1(obj);
        MyThreadClass2 t2 = new MyThreadClass2(obj);
        t1.start();
        t2.start();
    }
}
```

The output of the following code generates the following output.



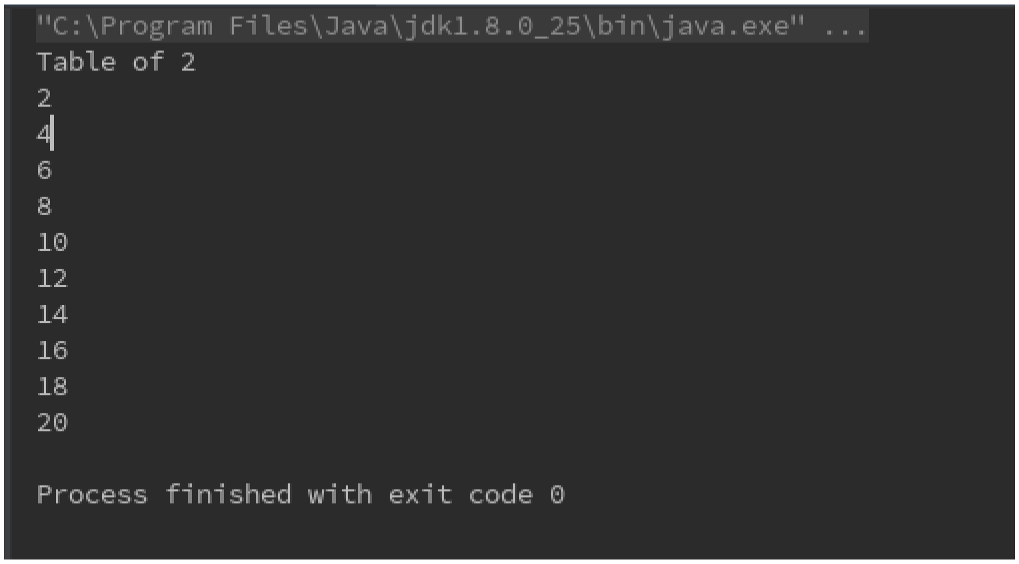
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
5
10
15
20
25
100
200
300
400
500

Process finished with exit code 0
```

`c. By static synchronized`

If we make any static method as synchronized, the lock will be on the class not on the object. The complete program is listed below.

```java
class PrintTable {
    public synchronized static void printTable(int n) {
        System.out.println("Table of " + n);       // static Synchroized

        for (int i = 1; i <= 10; i++) {
            System.out.println(n * i);
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
}

class MyThread1 extends Thread {
    public void run() {
        PrintTable.printTable(2);
    }
}

public class StaticSynchronizedExample {
    public static void main(String args[]) {

        //creating threads.
        MyThread1 t1 = new MyThread1();

        //start threads.
        t1.start();
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Table of 2
2
4
6
8
10
12
14
16
18
20

Process finished with exit code 0
```

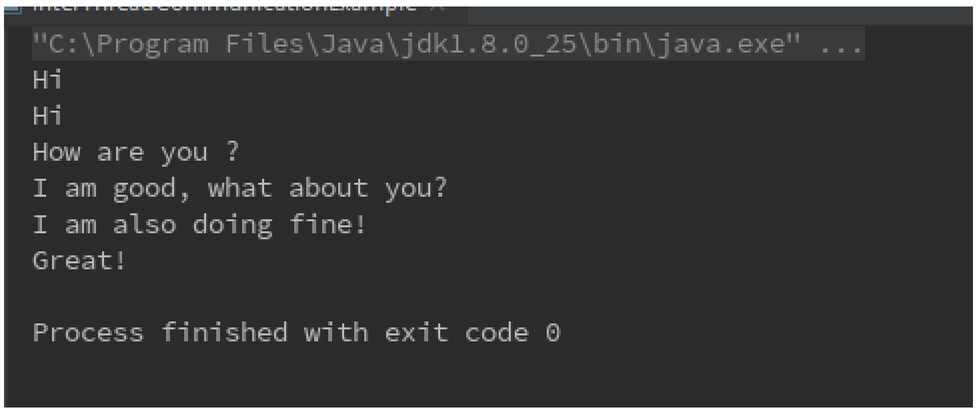
Cooperation (interthread communication)

Cooperation (interthread communication) is all about allowing synchronized threads to communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed.

Java provides benefits of avoiding thread pooling using interthread communication. The complete program of interthread communication is listed below.

```java
class Messages {
    boolean flag = false;

    public synchronized void Question(String msg) {
        if (flag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(msg);
        flag = true;
        notify();
    }

    public synchronized void Answer(String msg) {
        if (!flag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println(msg);
        flag = false;
        notify();
    }
}

class T1 implements Runnable {
    Messages m;
    String[] s1 = {"Hi", "How are you ?", "I am also doing fine!"};

    public T1(Messages m1) {
        this.m = m1;
        new Thread(this, "Question").start();
    }

    public void run() {
        for (int i = 0; i < s1.length; i++) {
            m.Question(s1[i]);
        }
    }
}

class T2 implements Runnable {
    Messages m;
    String[] s2 = {"Hi", "I am good, what about you?", "Great!"};

    public T2(Messages m2) {
        this.m = m2;
        new Thread(this, "Answer").start();
    }

    public void run() {
        for (int i = 0; i < s2.length; i++) {
            m.Answer(s2[i]);
        }
    }
}

public class InterThreadCommunicationExample {
    public static void main(String[] args) {
        Messages m = new Messages();
        new T1(m);
        new T2(m);
    }
}
```

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Hi
Hi
How are you ?
I am good, what about you?
I am also doing fine!
Great!

Process finished with exit code 0
```

What is Lock Concept in Multithreading?

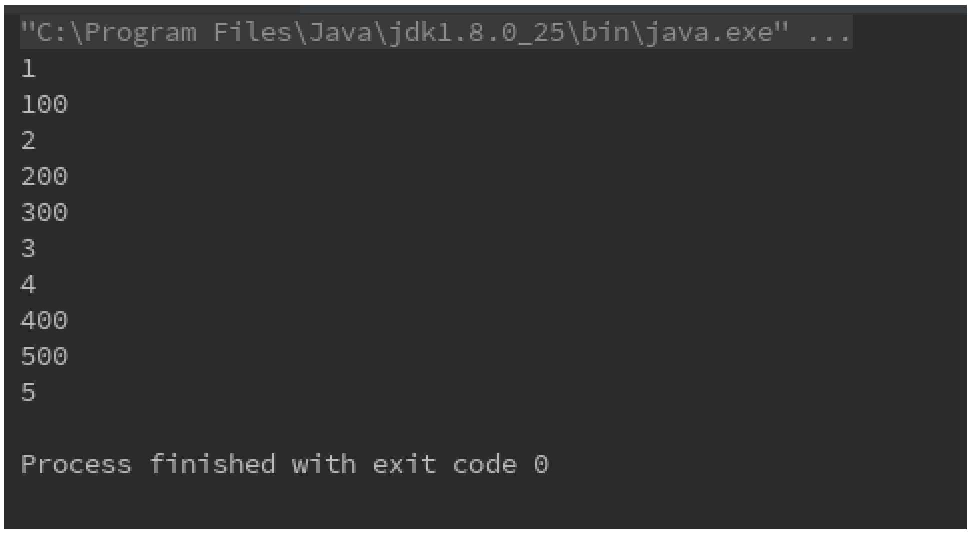
Synchronization is built around an internal entity called the lock or monitor. All objects have a lock associated with it. By rule, a thread that needs consistent access to an object's fields to get the object's lock before accessing them and release the lock, when it's done with them. Since Java 5 the package *java.util.concurrent.locks* contain many lock implementations.

The complete program is listed below.

```java
class A {
    void print(int n) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
            try {
                Thread.sleep(1000);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
}
```

```
class MyThreadLock1 extends Thread {
    A a;
    MyThreadLock1(A a) {
        this.a = a;
    }
    public void run() {
        a.print(1);
    }
}
class MyThreadLock2 extends Thread {
    A a;

    MyThreadLock2(A a) {
        this.a = a;
    }
    public void run() {
        a.print(100);
    }
}
public class LockConceptExample {
    public static void main(String args[]) {
        A obj = new A();
        MyThreadLock1 t1 = new MyThreadLock1(obj);
        MyThreadLock2 t2 = new MyThreadLock2(obj);
        t1.start();
        t2.start();
    }
}
```

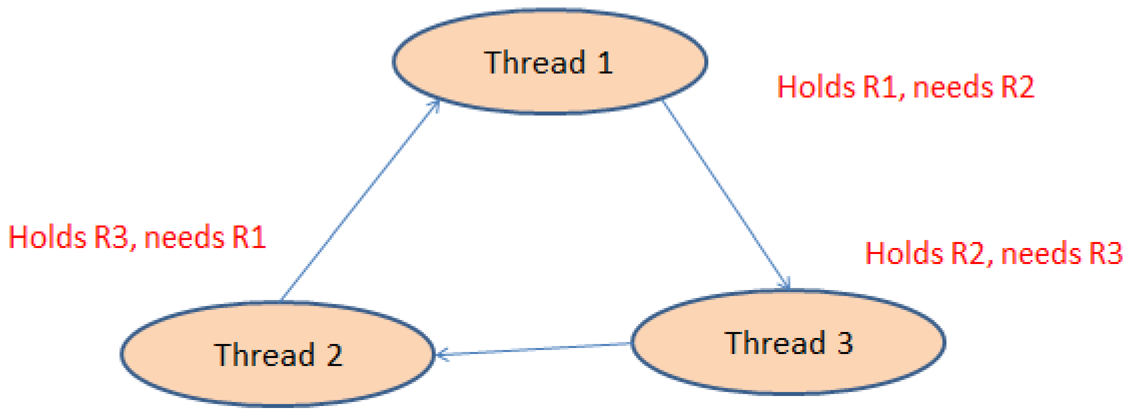
The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
1
100
2
200
300
3
4
400
500
5

Process finished with exit code 0
```

What is Thread Deadlock in Java?

Deadlock is a situation of complete Lock when no thread can complete its execution because of lack of resources. Deadlock occurs when multiple threads need the same locks but obtain them in a different order. Since both threads are waiting for each other to release the lock, this condition is called deadlock.

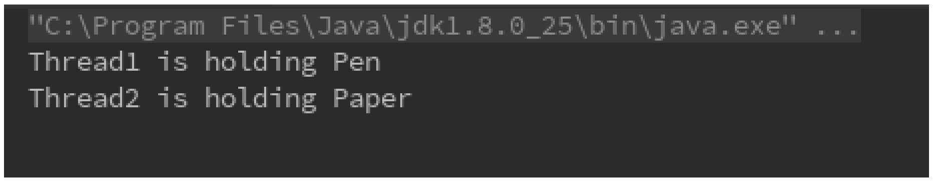


Deadlock Condition

The complete program is listed below.

```java
class Pen{}
class Paper{}

public class DeadLockExample {

    public static void main(String[] args)
    {
        final Pen pn =new Pen();
        final Paper pr =new Paper();

        Thread t1 = new Thread() {
            public void run()
            {
                synchronized(pn)
                {
                    System.out.println("Thread1 is holding Pen");
                    try{
                        Thread.sleep(1000);
                    }
                    catch(InterruptedException e){
                        // do something
                    }
                    synchronized(pr)
                    {
                        System.out.println("Requesting for Paper");
                    }
                }
            }
        };
        Thread t2 = new Thread() {
            public void run()
            {
                synchronized(pr)
                {
                    System.out.println("Thread2 is holding Paper");
                    try {
                        Thread.sleep(1000);
                    }
                    catch(InterruptedException e){
                        // do something
                    }
                    synchronized(pn)
                    {
                        System.out.println("requesting for Pen");
                    }
                }
            }
        };

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

The output of the following code generates the following output.



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Thread1 is holding Pen
Thread2 is holding Paper
```

Summary

In this tutorial, we learned about Multithreading in Java and how to achieve Multithreading in our Java programs.