

Multithreading in Java

Example 1: Child Thread, main Thread using run() method (call by user).

```
1. class MyThread extends Thread
2. {
3.     public void run()//overridden method
4.     {
5.         for(int i=1;i<=10;i++)
6.         {
7.             System.out.println("Child Thread is running:"+i);
8.         }
9.     }
10.}
    //1-10 lines defining a thread
    //Lines 5-8 is the job of the child thread

class Test
{
    public static void main(String args[])//main thread
    {
        MyThread mt=new MyThread();//thread instance creation,main thread
        create a child thread object
        mt.start();//main thread start child thread, child thread is responsible for
        executing run()method line(5-8)after start method.
        for(int j=1;j<=20;j++)
        {
            System.out.println("Main Thread is running:"+j);
        }
        //main thread is responsible for running the remaining code
    }
}

// both jobs will be executed simultaneously, and we will get mixed output.
```

C:\Windows\System32\cmd.exe

```
F:\Java Code>javac Test.java
```

```
F:\Java Code>java Test
```

```
Main Thread is running:11
```

```
Child Thread is running:1
```

```
Main Thread is running:12
```

```
Child Thread is running:2
```

```
Main Thread is running:13
```

```
Main Thread is running:14
```

```
Main Thread is running:15
```

```
Child Thread is running:3
```

```
Child Thread is running:4
```

```
Main Thread is running:16
```

```
Child Thread is running:5
```

```
Child Thread is running:6
```

```
Child Thread is running:7
```

```
Child Thread is running:8
```

```
Main Thread is running:17
```

```
Child Thread is running:9
```

```
Child Thread is running:10
```

```
Main Thread is running:18
```

```
Main Thread is running:19
```

```
Main Thread is running:20
```

Example 2:Child Thread, main Thread using run() method.

```
class MyThread extends Thread
{
    public void run()//overridden method
    {
        for(int i=1;i<=10;i++)
        {
            System.out.println("Child Thread is running:"+i);
        }
    }
}
class Test
{
    public static void main(String args[])//main thread
    {
        MyThread mt=new MyThread();
        mt.run();
        for(int j=11;j<=20;j++)
        {System.out.println("Main Thread is running:"+j);
        }
    }
}
```

C:\Windows\System32\cmd.exe

```
F:\Java Code>java Test
Child Thread is running:1
Child Thread is running:2
Child Thread is running:3
Child Thread is running:4
Child Thread is running:5
Child Thread is running:6
Child Thread is running:7
Child Thread is running:8
Child Thread is running:9
Child Thread is running:10
Main Thread is running:11
Main Thread is running:12
Main Thread is running:13
Main Thread is running:14
Main Thread is running:15
Main Thread is running:16
Main Thread is running:17
Main Thread is running:18
Main Thread is running:19
Main Thread is running:20
```

Thread Scheduler in Java

- It is a part of JVM.
- It is responsible for scheduling the threads. If multiple threads are waiting for a chance to execute, the thread scheduler decides the order in which the threads will execute.
- We can't expect an exact algorithm followed by a thread scheduler; it varies from JVM to JVM. Hence, we can't expect thread execution order and exact output.
- Hence whenever a situation comes to multithreading, there is no guarantee for exact output, but we can provide several possible outputs.

Overloading of run() method:

Overloading the run method is possible, but the Thread class start() method can invoke no argument run method, the other overloaded method we have to call explicitly like a normal method call. Similarly, if we overload the main() method, but JVM always calls the String args[] method.

Example:

```
class MyThreadEx extends Thread
{
    public void run()
    {
        System.out.println("No argument run method:");
    }
    public void run(int i)
    {
        System.out.println("int argument run method:");
    }
}

class TestOverload
{
    public static void main(String arg[])
    {
        MyThreadEx t1=new MyThreadEx();

        t1.start(); // run() or run(int i)
    } }
```

```
F:\Java Code>javac TestOverload.java

F:\Java Code>java TestOverload
No argument run method
```

If we are not overriding run() method:

If you are not overriding the run method, then the Thread class run method will be executed with an empty implementation. Hence, we will not get any output.

It's highly recommended to override the run method. Otherwise, do not go for the multithreading concept.

Example:

```
class MyThreadEx extends Thread
{
//not override run() method.

}
class NoOverride
{
    public static void main(String arg[])
    {
        MyThreadEx t1=new MyThreadEx();

        t1.start(); ---reg---invoke run()(parent)
    }
}
```

```
F:\Java Code>javac NoOverride.java

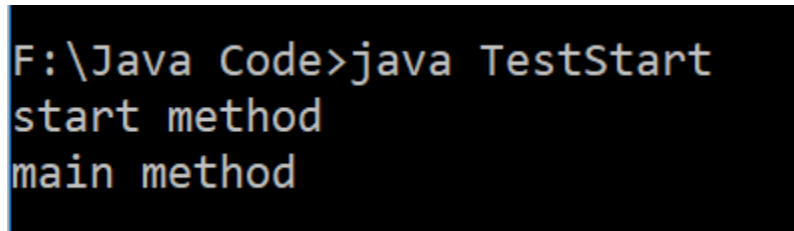
F:\Java Code>java NoOverride

F:\Java Code>
```

Overriding of start() method:

```
class MyThreadEx1 extends Thread
{
    public void start()
    {
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}

class TestStart
{
    public static void main(String args[])
    {
        MyThreadEx1 t=new MyThreadEx1();
        t.start();
        System.out.println("main method");
    }
}
```



```
F:\Java Code>java TestStart
start method
main method
```

Always get similar output if we run 1000 times.

Reason:

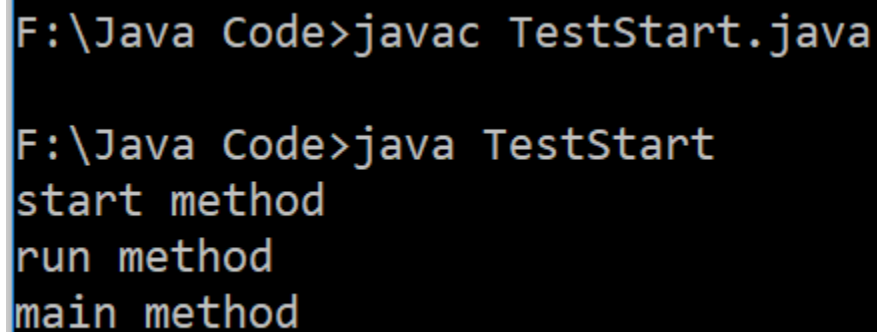
If we override the start method, our start method will be executed just like a normal method call, and a new thread will not be created. Total output will be produced by the main thread. So it is recommended **never to override the start method**; otherwise, don't go for the multithreading concept.

Let's do small change in the previous program.

```
class MyThreadEx1 extends Thread
{
    public void start()
    {
        super.start();
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
class TestStart
{
    public static void main(String args[])
    {
        MyThreadEx1 t=new MyThreadEx1();
        t.start();
        System.out.println("main method");
    }
}
```

We will get Mixed Output

Just because of super.start(),



```
F:\Java Code>javac TestStart.java

F:\Java Code>java TestStart
start method
run method
main method
```


Restart the same thread.

After starting the thread if we are trying to restart the same thread then we will get run time exception: Illegal Thread state exception.

```
mt1.start();  
System.out.println("main thread");  
mt1.start();
```

```
java.lang.IllegalThreadStateExceptionRunning Thread1:5  
    at java.lang.Thread.start(Thread.java:708)  
Running Thread1:6    at TestThread1.main(TestThread1.java:18)
```