***MULTI-THREADING***

**Introduction:**

**Multi-Tasking :** Executing multiple tasks simultaneously (at a time) is called as “Multi-tasking”. It is of 2 types.

1. Process based Multi-tasking
2. Thread based Multi-tasking

**Process based Multi-tasking:** Executing multiple tasks simultaneously where each task is an independent process is called as Process based Multi-tasking.

Ex: while writing java program, we can listen to songs and at the same time we can download some file from internet. All these 3 tasks are executing at a time , but all of them are separate and independent tasks. This is called as Process based Multi-tasking.

This type of Multi-tasking is best suitable at OS level.

**Thread based Multi-tasking:** Executing multiple tasks simultaneously where each task is separating independent part of the same program is called as Thread based Multi-tasking.

This is best suitable at Programatic level and each part of the same program is called as a “**Thread’**.

Ex: Need to write a java program which has to display time and temperature.

As time and temperature are independent components, we can write 2 threads where 1 thread will display time and other will display temperature.

Whether it is a Process based or Thread based Multi-tasking, the main goal is to reduce response time and utilize CPU resources properly.

Java has a rich API for Threads by providing Thread class and Runnable interface. So developing multi-threading program in java is very simple.

The main applications of Multi-threading are developing games and Graphics.

**The ways we can define, instantiate and start a new Thread:**

We can define the Thread either by extending the Thread class or by implementing Runnable interface.

Defining a Thread by extending Thread class:

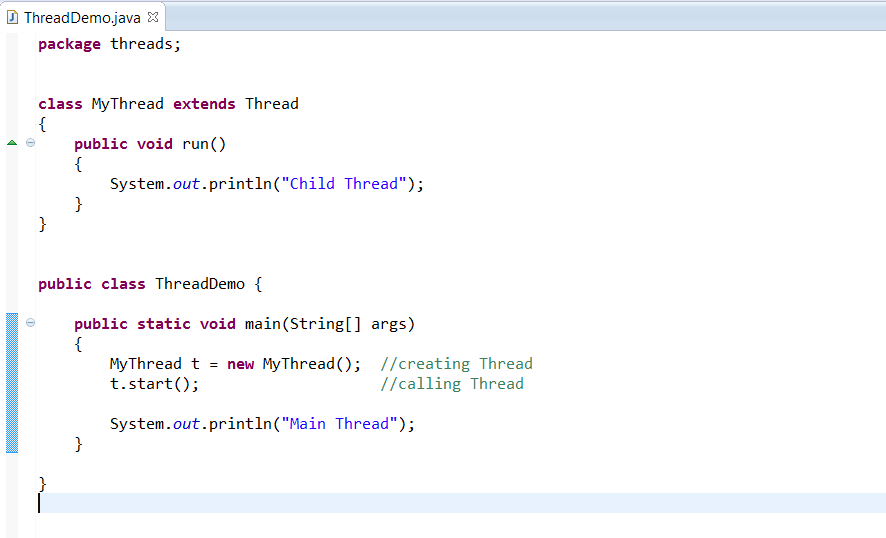
Thread class is present in lang package. java.lang.Thread

Signature of Thread class is

**public class java.lang.Thread implements java.lang.Runnable {**

**}**

Thread class implements Runnable interface.



The output may be anyone of this,

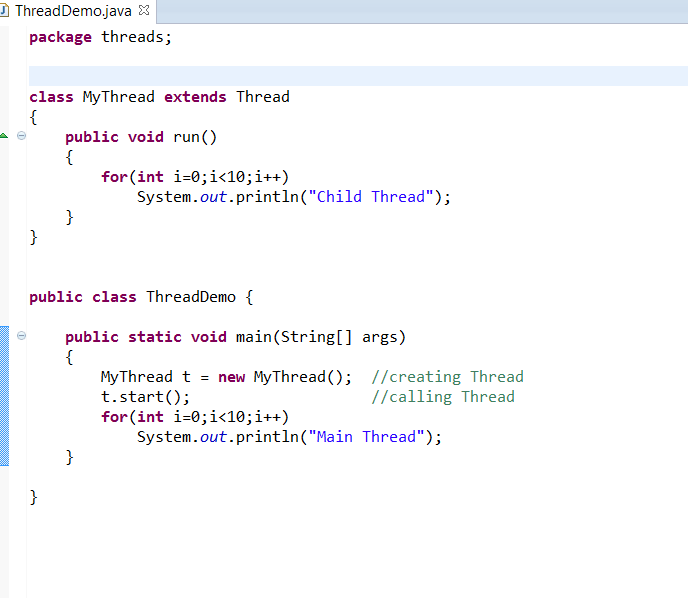
Main Thread

Child Thread

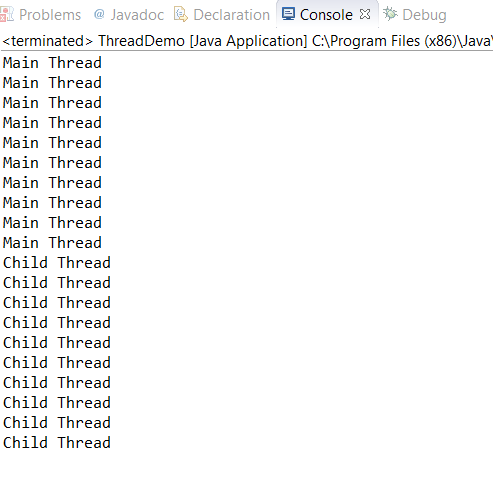
Or

Child Thread

Main Thread



Following outputs are possible.



Or

Child Thread (10 times)

Main Thread (10 times)

Or Main Thread and Child Thread alternatively.

**Thread Schedular:**

Why we are getting multiple possibilities of outputs in above program.

If we are having more than one Thread in a Program, then which thread will get executed first will be decided by Thread Schedular (this is a part of JVM).

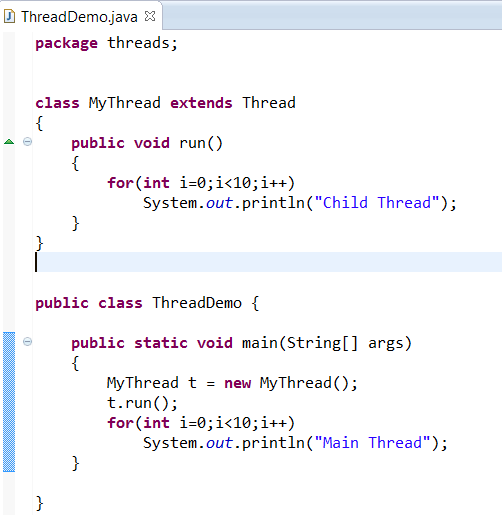
We can’t expect the exact behavior of Thread scheduler. It is a vendor dependent. So we can’t predict the output exactly.

So that’s the reason the code in Threads should be independent to each other. Don’t wait for the output of other Thread.

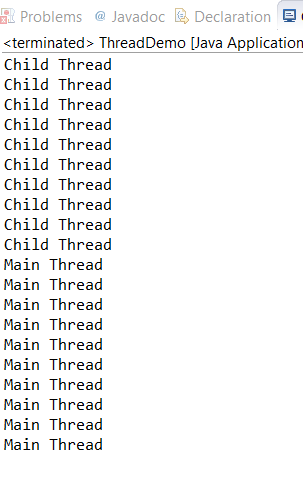
**Diff b/w t.start() and t.run()**

In case of t.start(), new thread will be created which is responsible for executing the run method.

But in case of t.run(), no thread will be created and run method will be executed just like a normal method call by main method. We need to call it explicitly to execute that code.



Output is :



Here t.run() is just called like a normal method and no thread will be created. So we can expect the exact output in this case.

**Specialty of Thread class start method t.start()**

The Thread class start method has the responsibility to fulfill the joining formalities of Thread like registering our Thread with Thread scheduler.

public synchronized void start()

{

1. Register our Thread with Thread scheduler
2. Invoke run method.

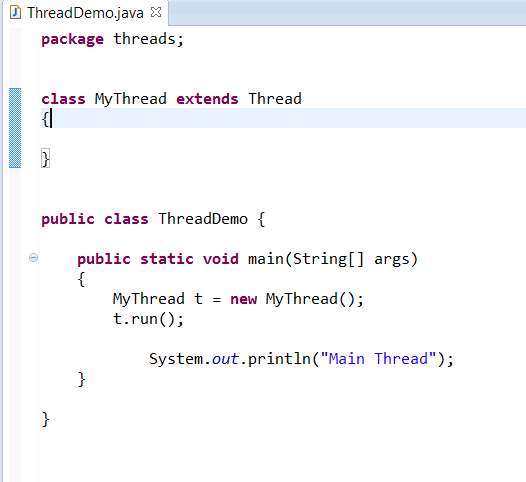
}

Because of start method, programmer will concentrate only on job of thread instead of low-level functionality (registering thread with thread scheduler etc.)

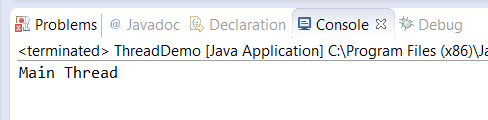
**If we are not overriding Thread class run method**

Then Thread class run method will execute which has empty implementation. Hence we won’t get any output.

Note: So it is highly recommended to override run method to define our job.

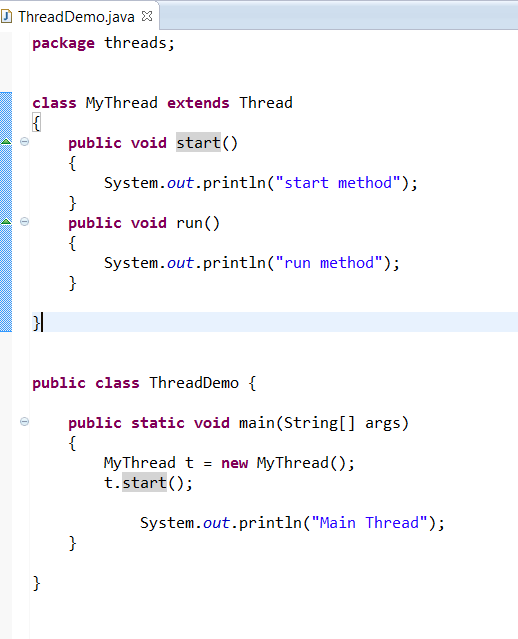


And output is

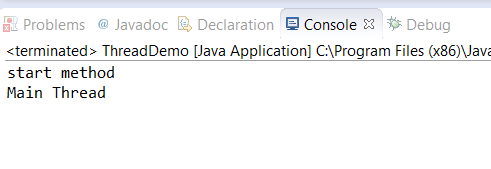


**If we are overriding Thread start method**

We can override Thread class start method, in that case no thread will be created and start method will be executed just like a normal method call.



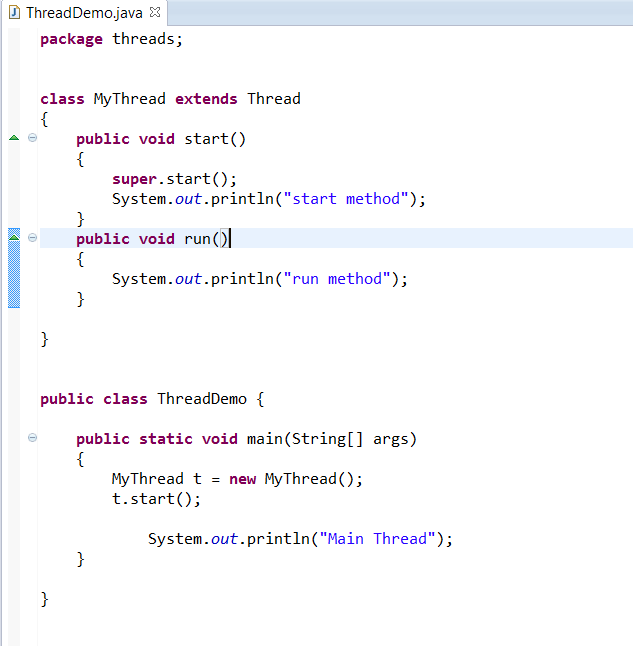
And output is



So clearly we can observe that start method didn’t call run method. That means no thread will be created if we are overriding start method. And it is called just like a normal method.

Note: It is not recommended to override start method in our class. We should utilize Thread class start method.

Case study:



In this program , we are overriding start method and at the same time we are calling Thread class start method (by using super.start() ), so thread will be created and run method will be invoked.

And the output possibilities are,

start method or start method

run method Main method

Main method run method

But this is not possible

Main method

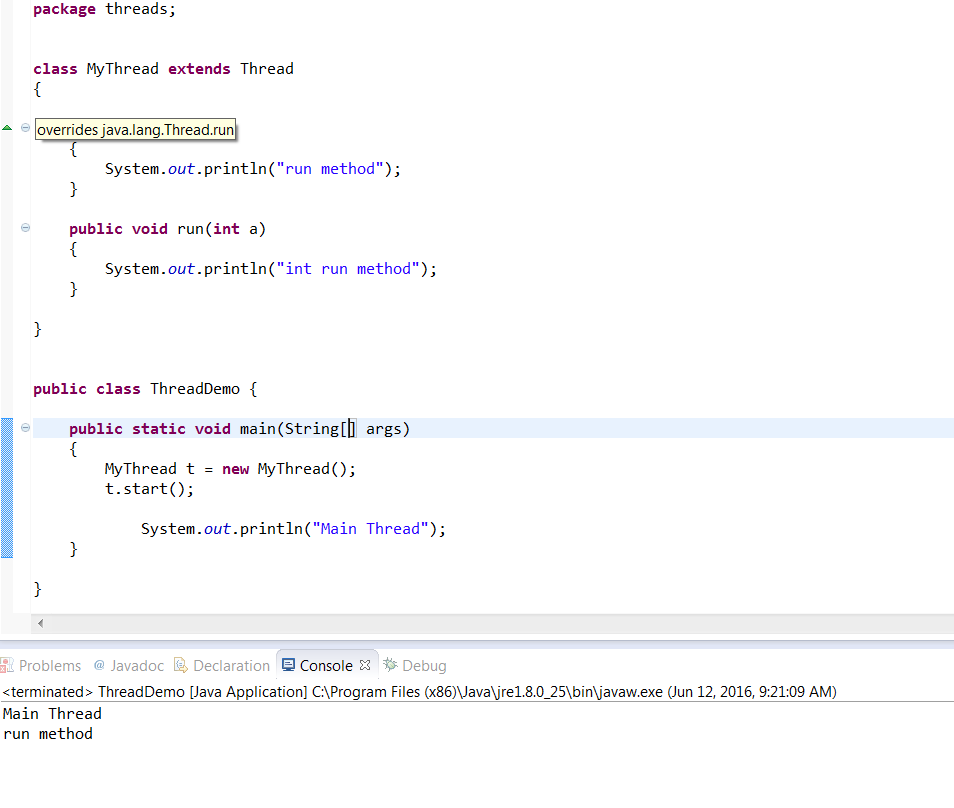
start method

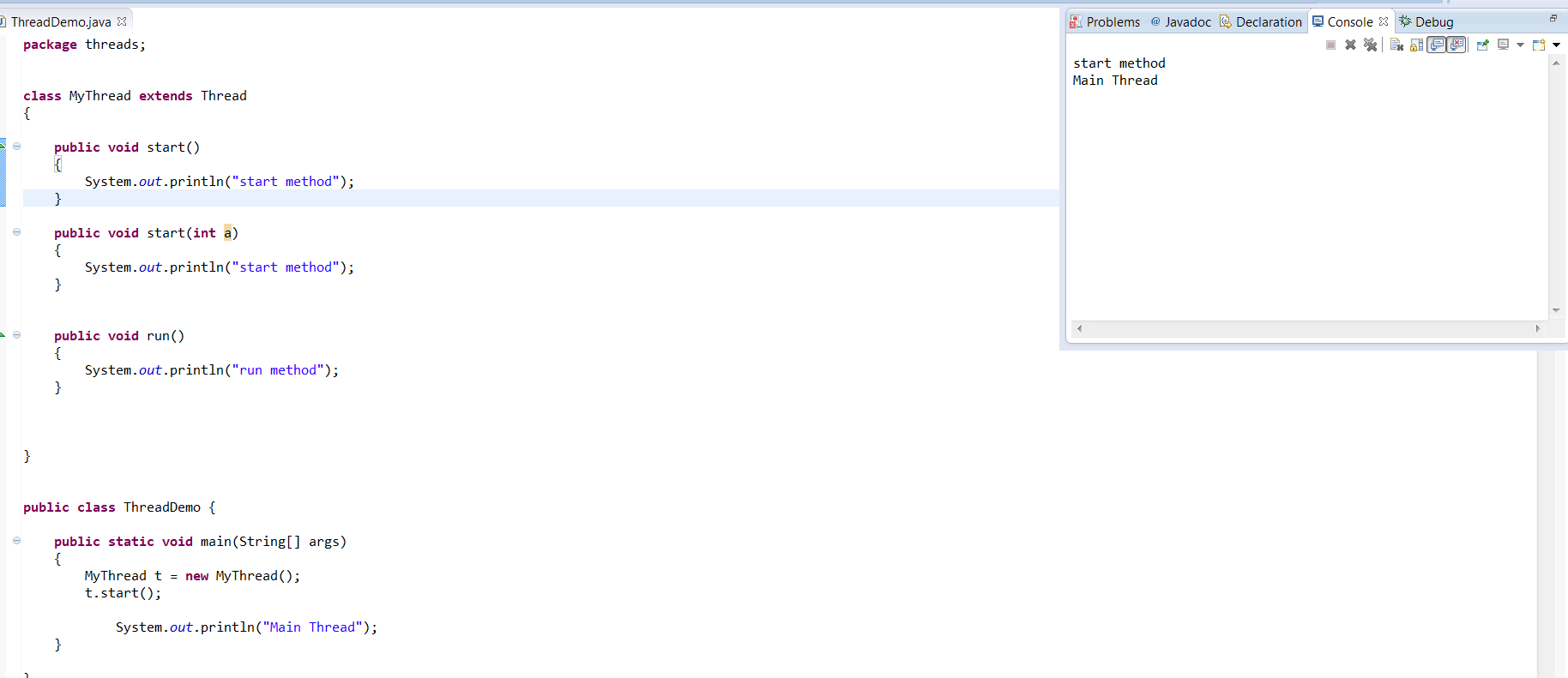
run method

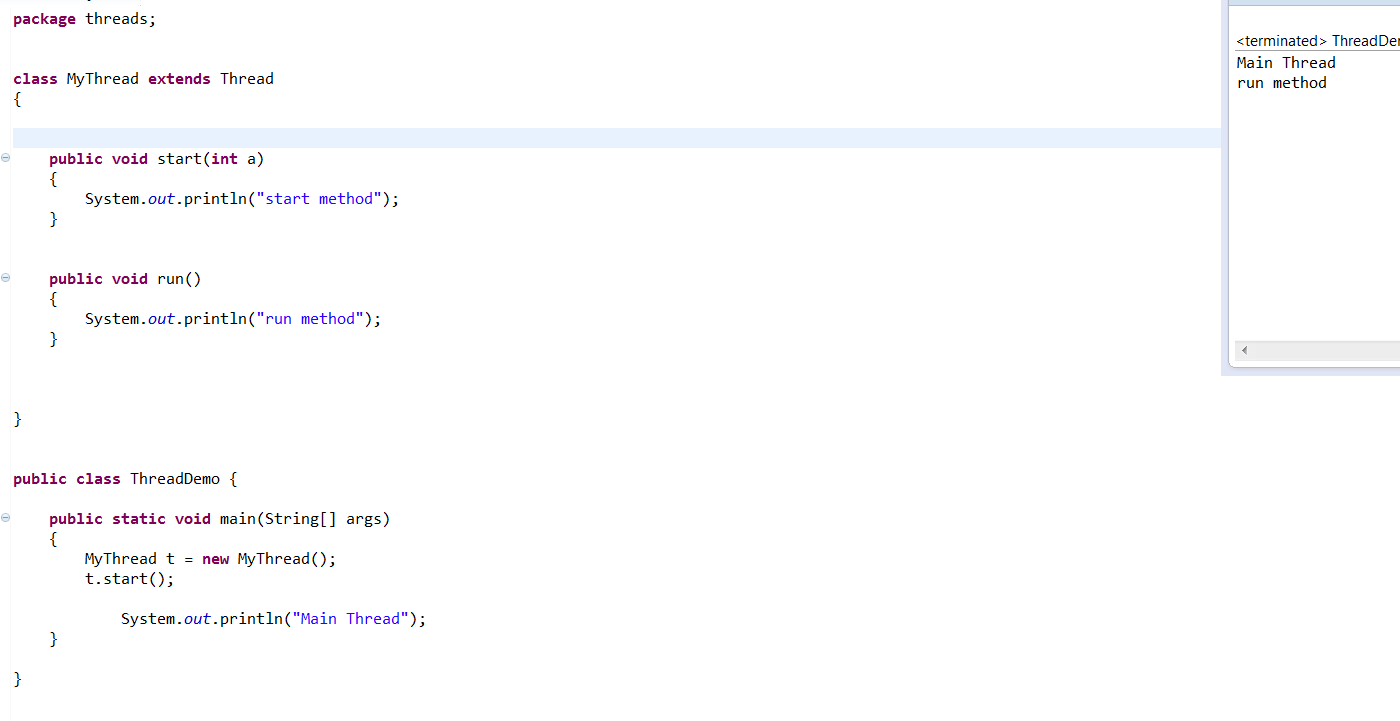
**Overloading a run method**

We can override run method but Thread class start method will always invoke no arguments run method.

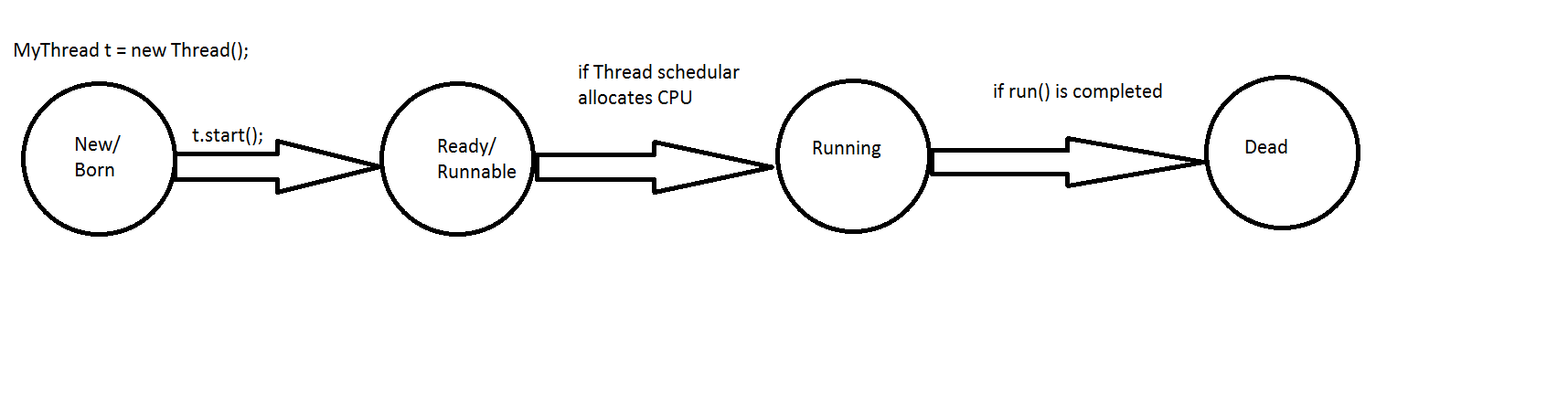
run method with parameters are just like normal methods and would be invoked separately.







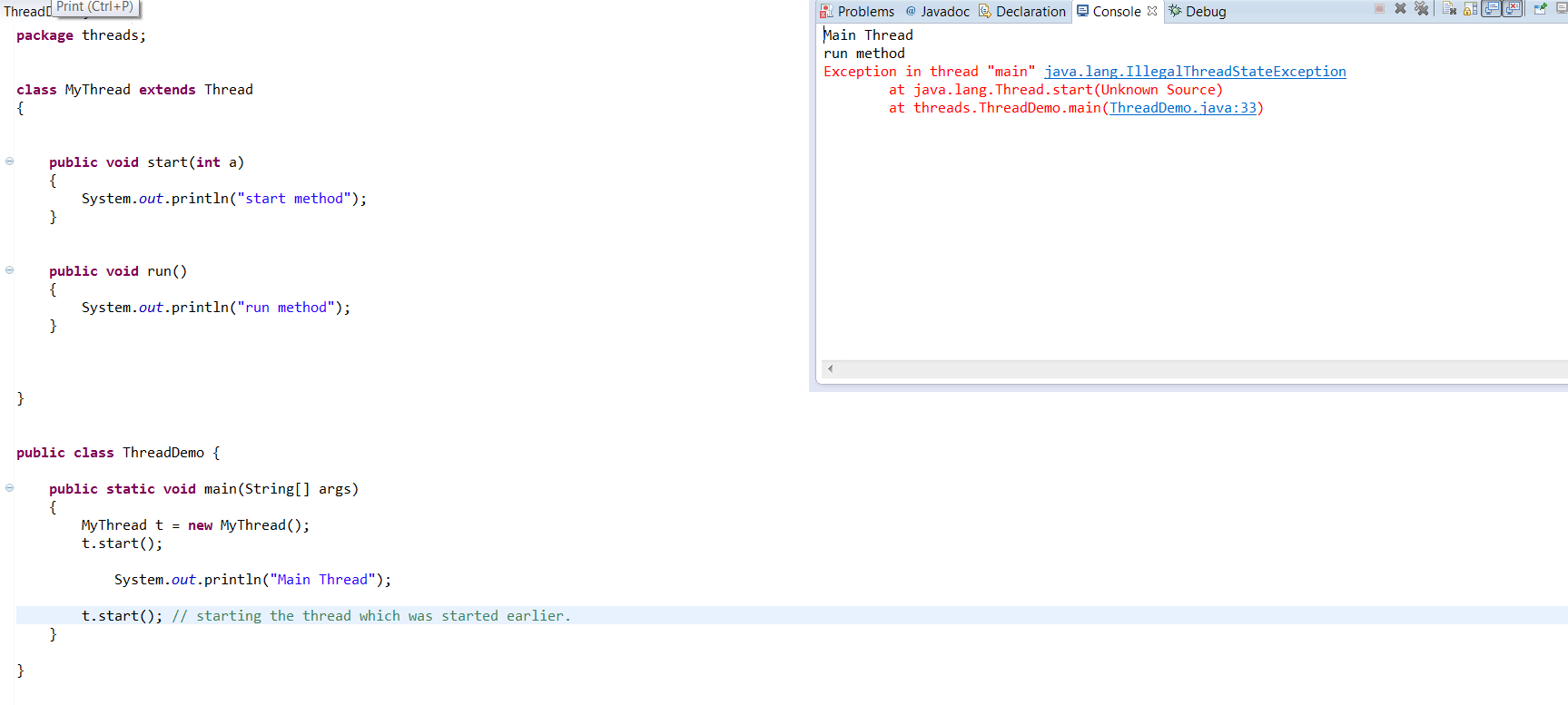
**Life cycle of a Thread**

****

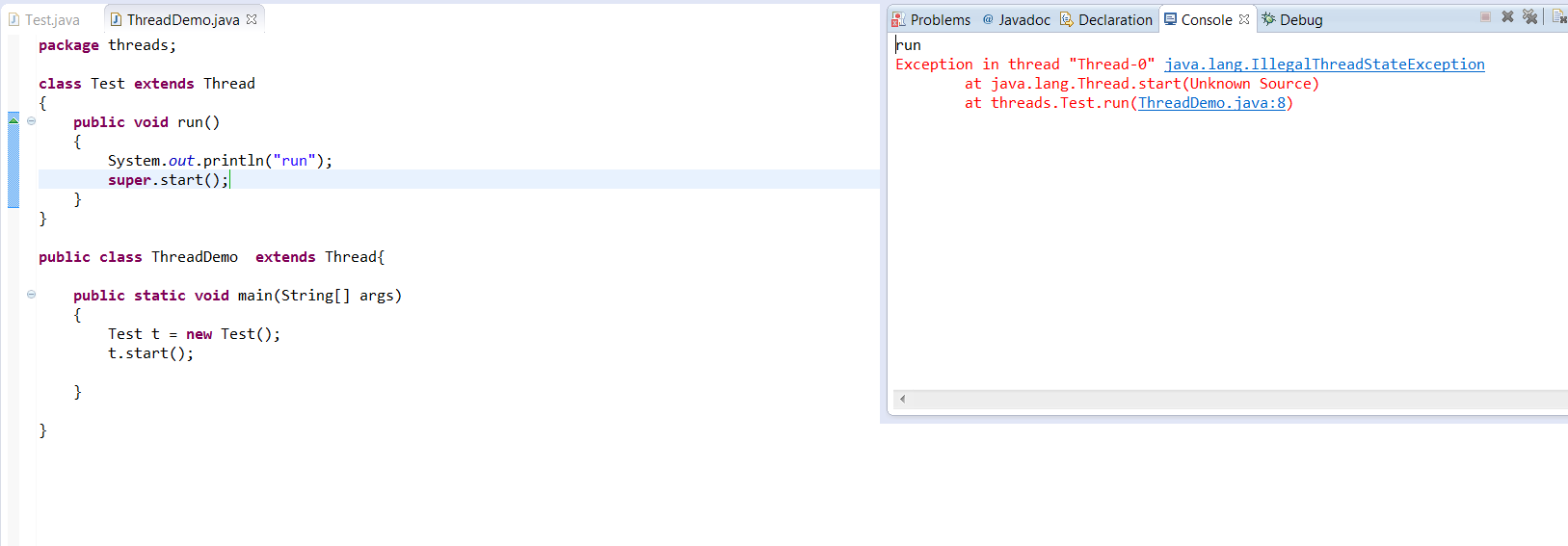
**IllegalThreadStateException:**

Once Thread is started, then there is no chance of restarting the same Thread again.

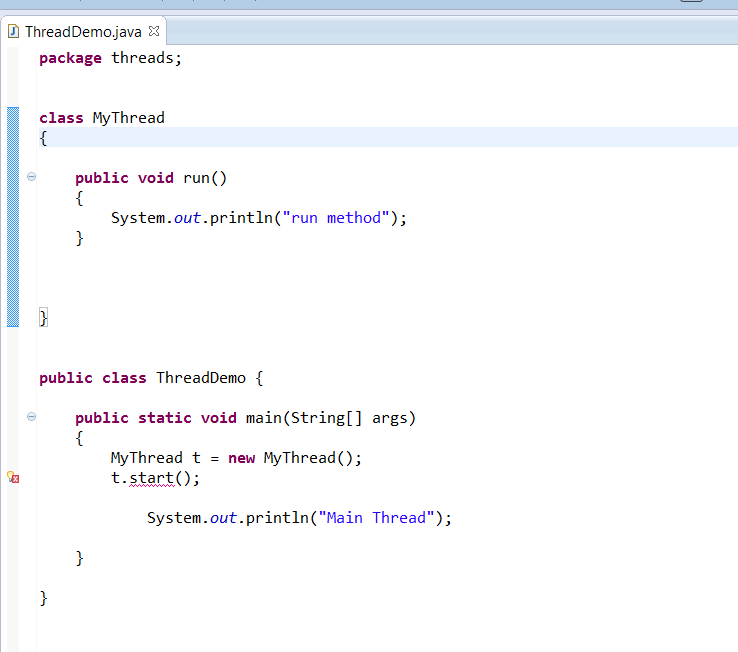
Violation leads to run time exception saying IllegalThreadStateException.

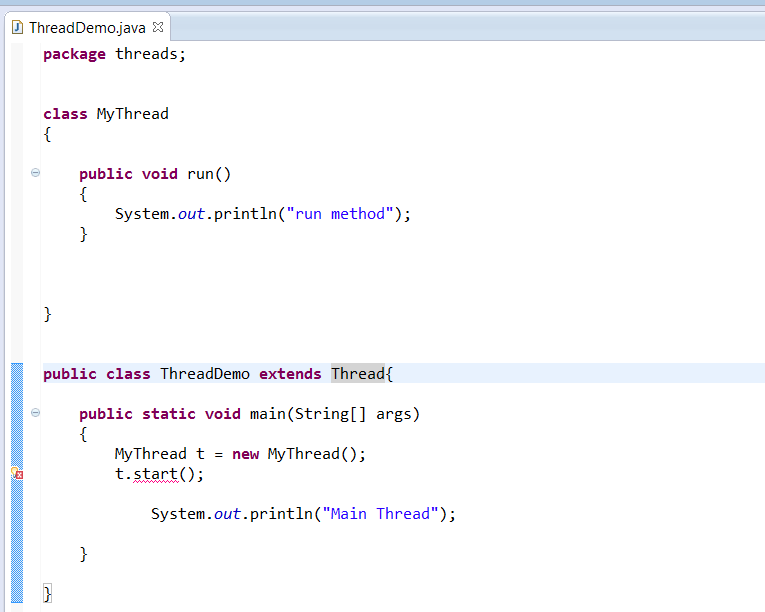


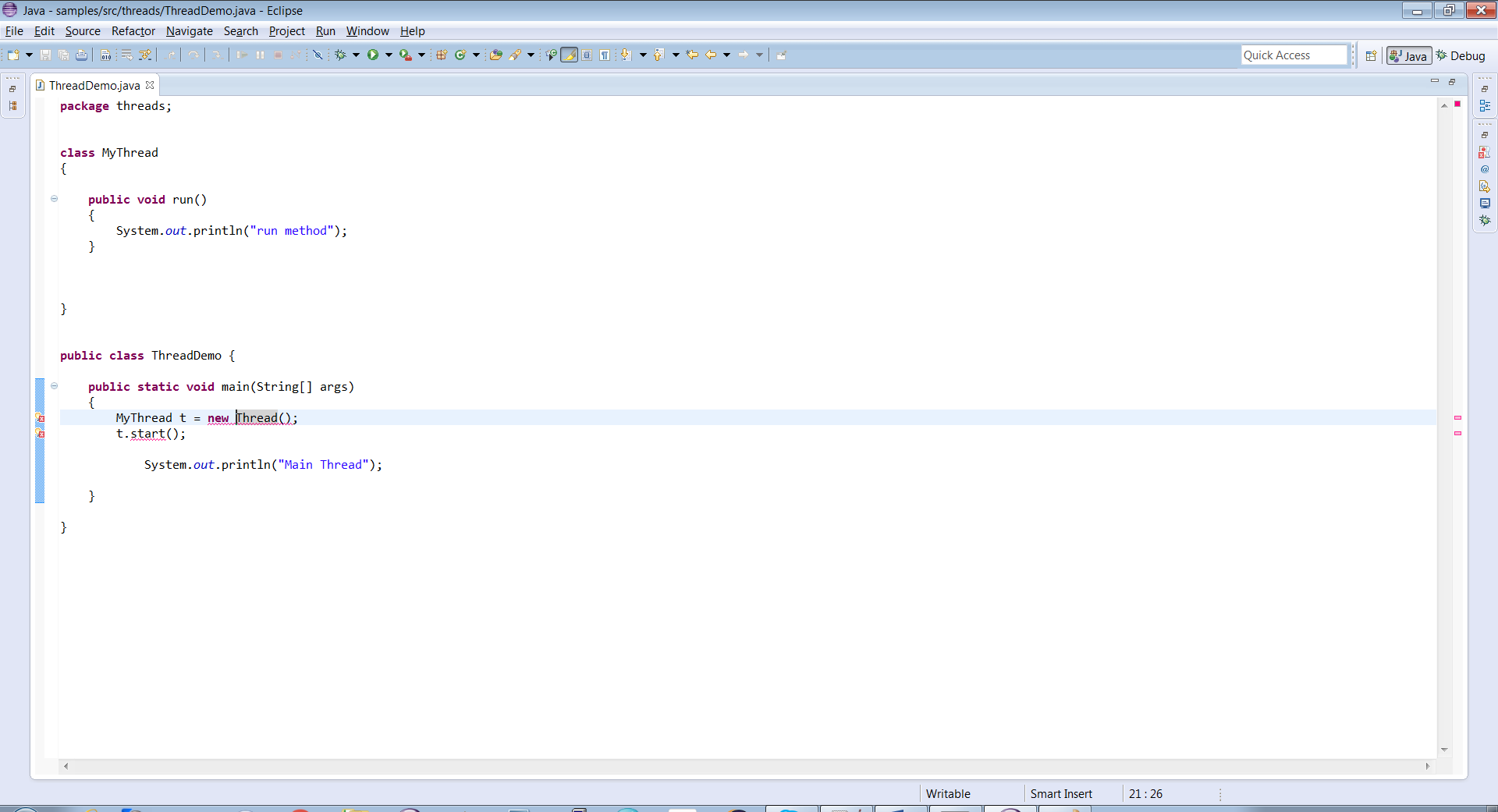
The same error will occur if we are trying to start Thread from run method.

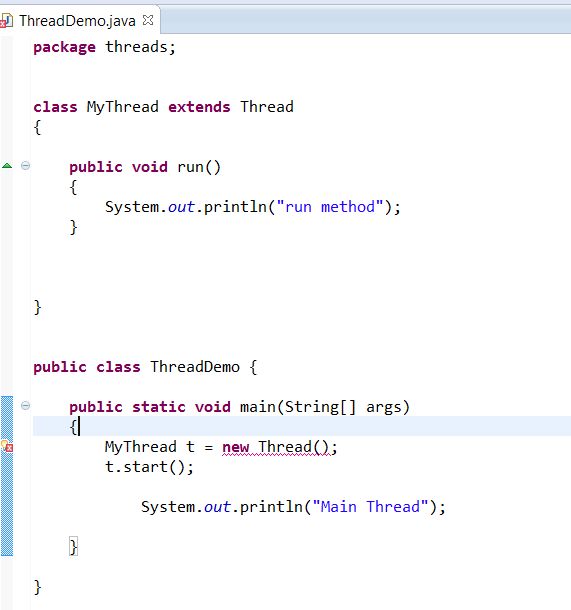


Examples programs:

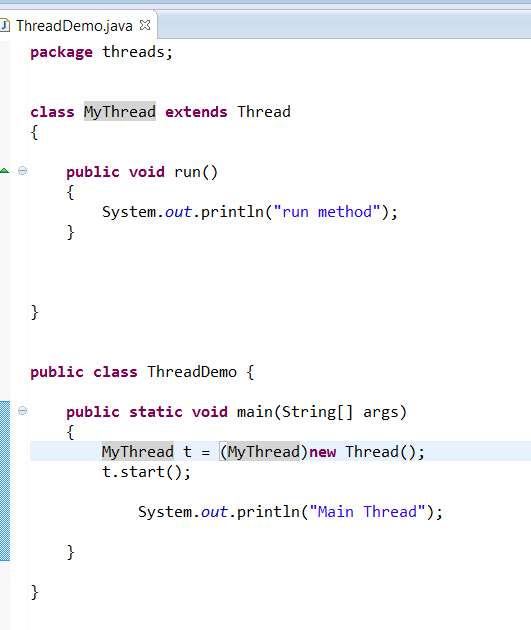




we can’t convert Thread type in to MyThread. (Compile time error)

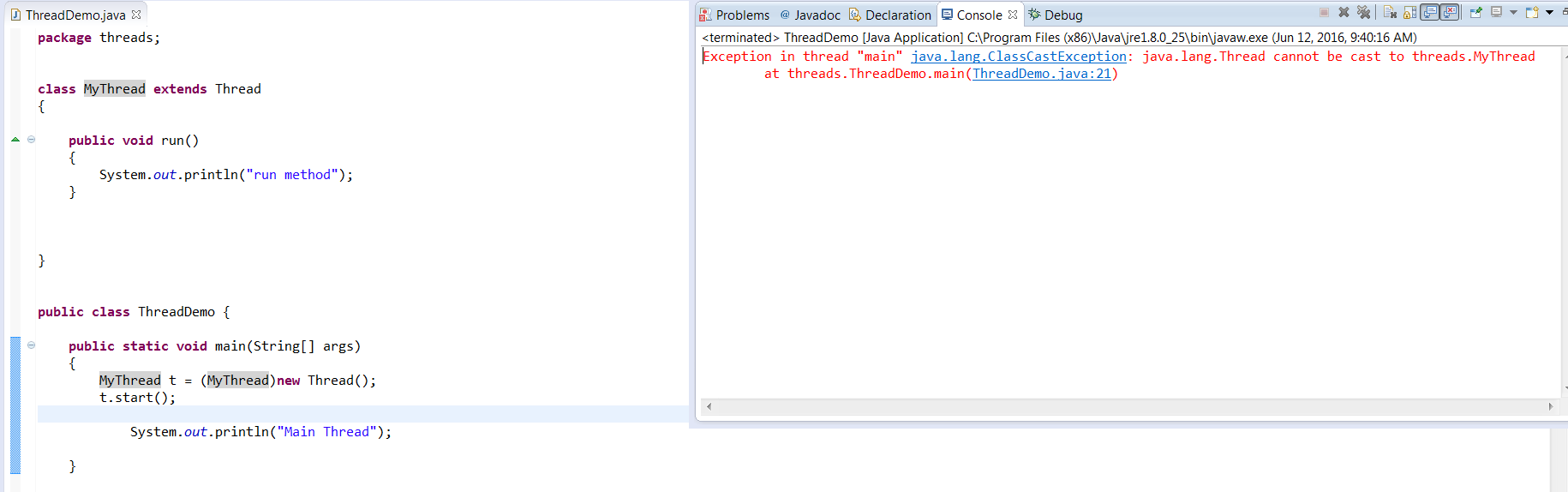


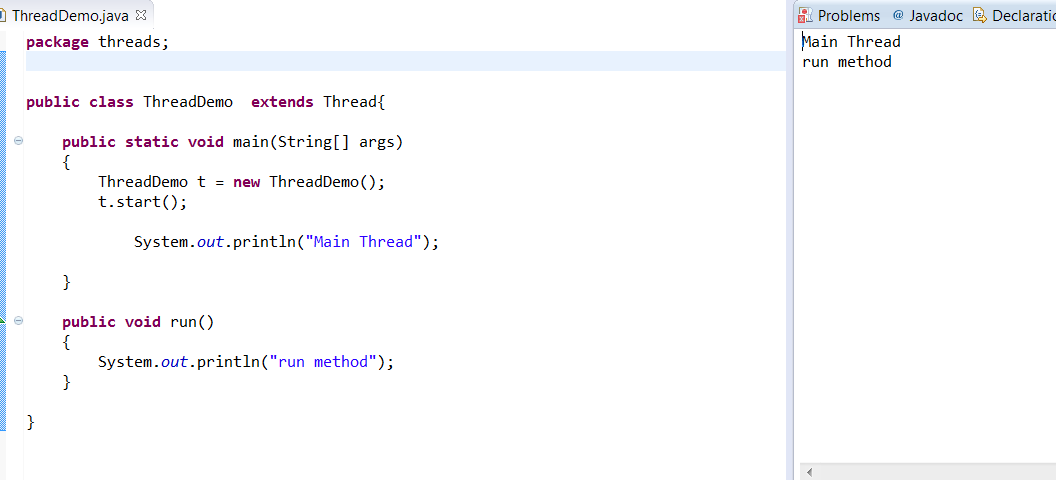
Here Thread is parent and MyThread is child. Child reference will not hold Parent object.



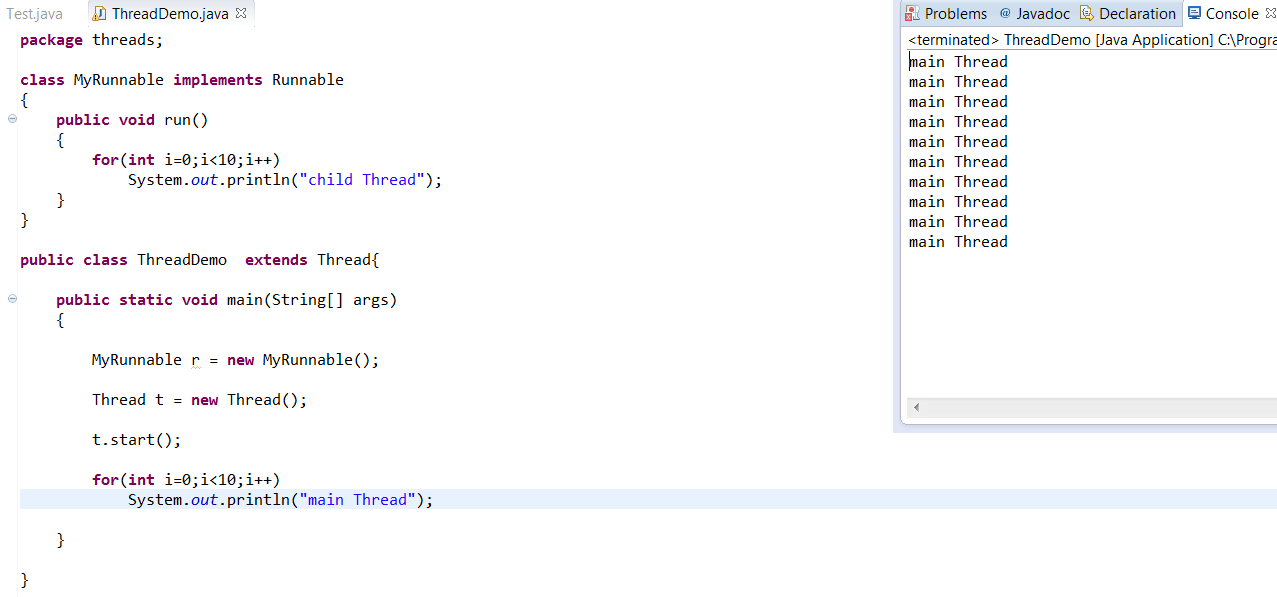
This is valid during compilation time as we are converting (type casting) parent object in to child type.

But during run time we will get ClassCastException



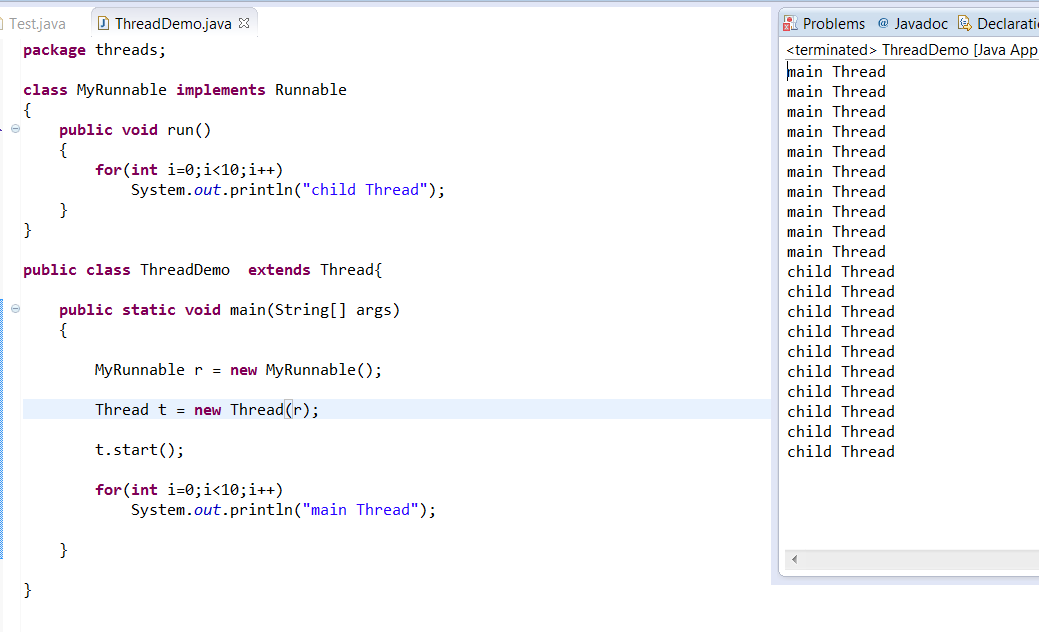


Defining Thread by implementing Runnable interface:



Here, we are not attaching MyRunnable reference to Thread.

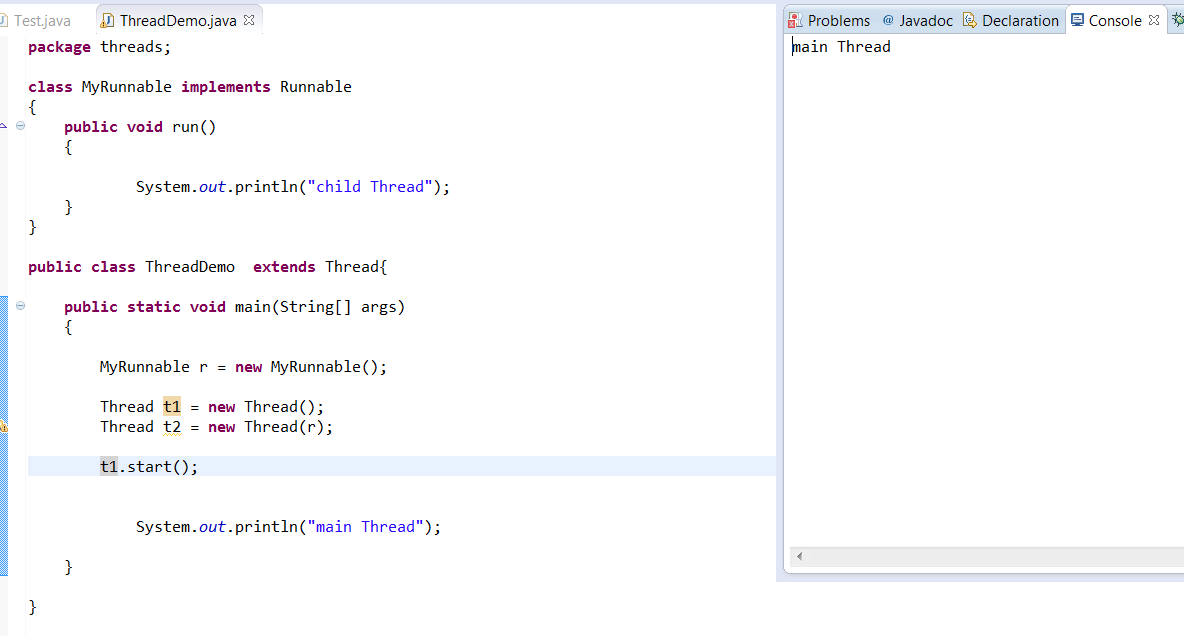
So Thread class run() method will execute which is of empty implementation.



Here MyRunnable run() method will execute.

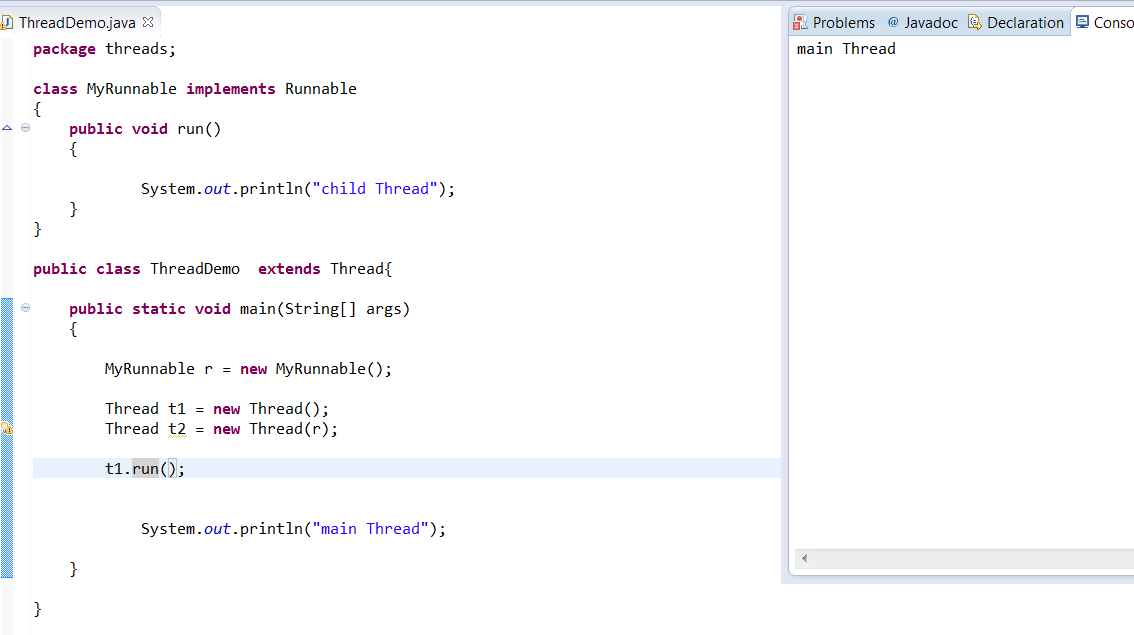
Case studies:

Case 1:



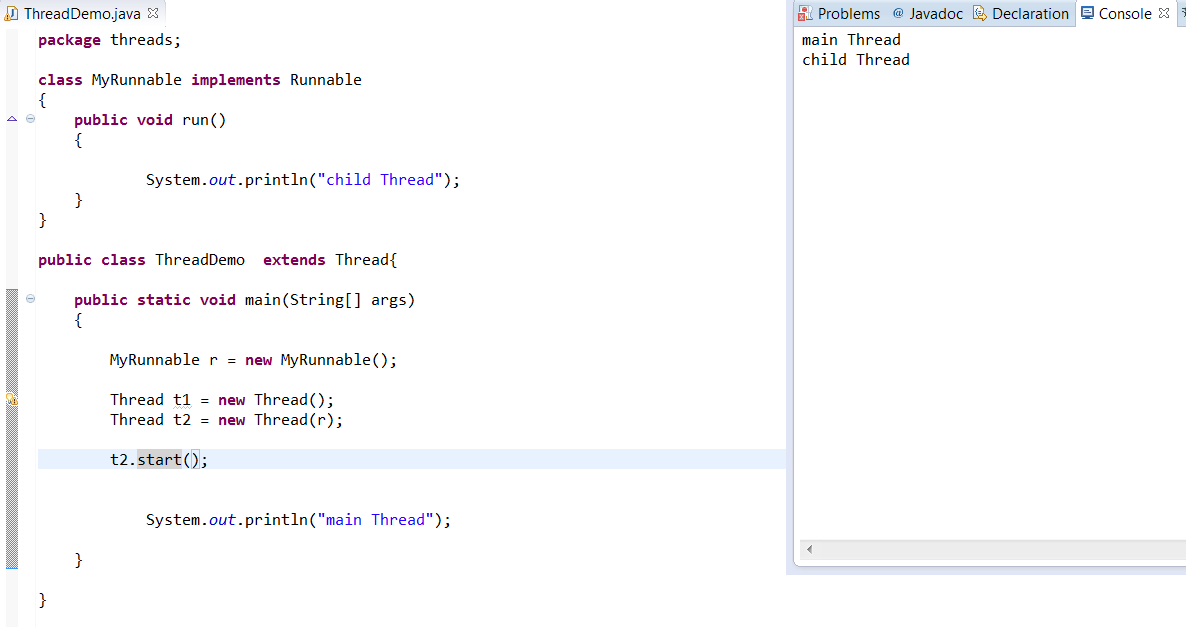
A new Thread will be created which is responsible of executing Thread class run() method.

Case 2:



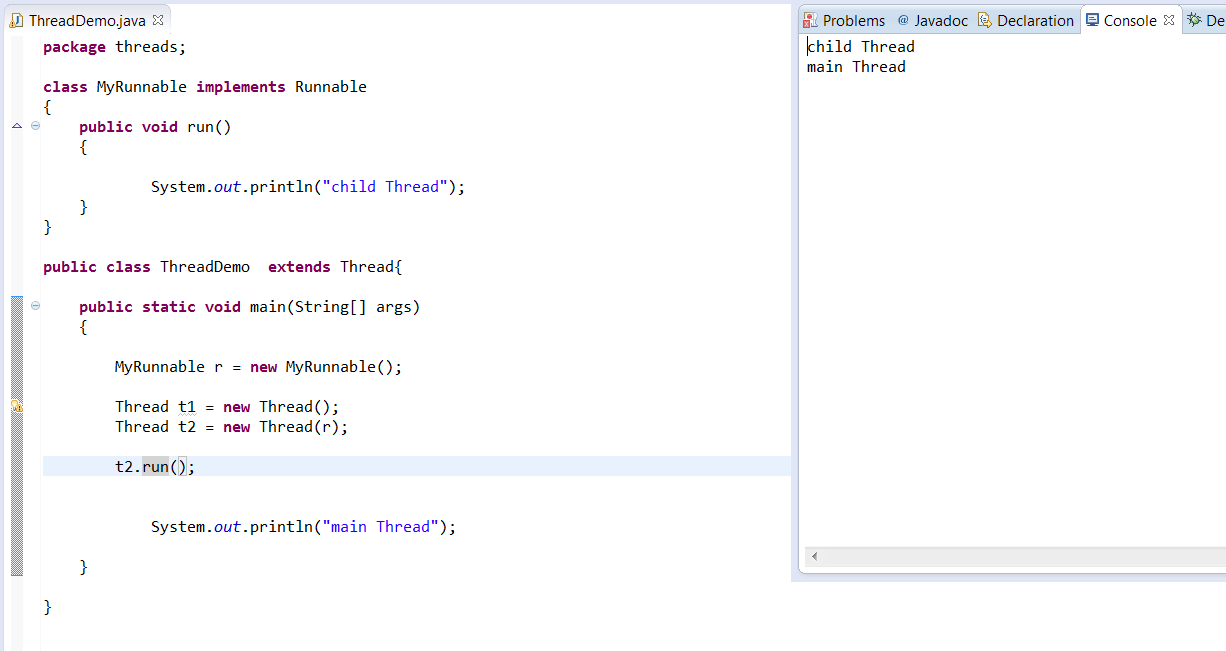
No Thread will be created and Thread class run() method will be executed just like a normal method call.

Case 3:



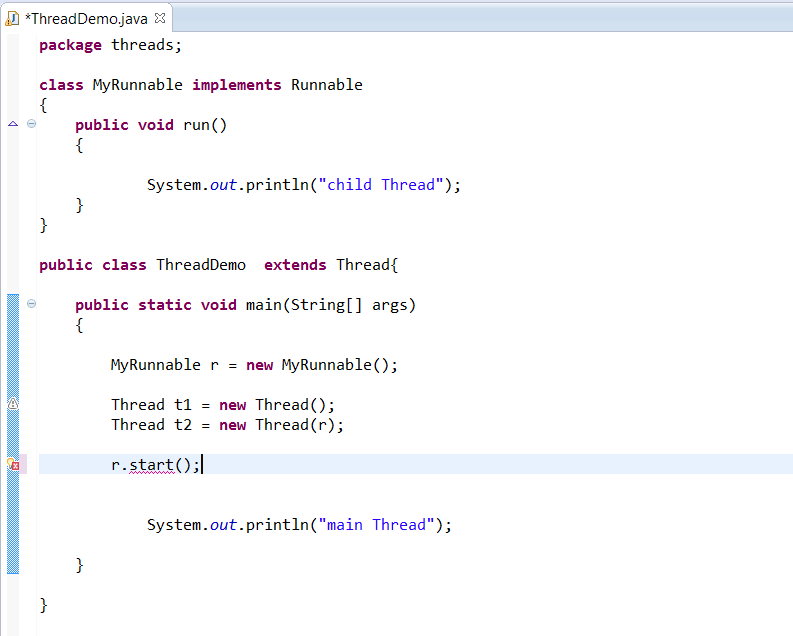
New Thread will be created which is responsible of executing MyRunnable run() method.

Case 4:



No new Thread will be created and MyRunnable run() method is executed just like a normal method call.

Case 5:



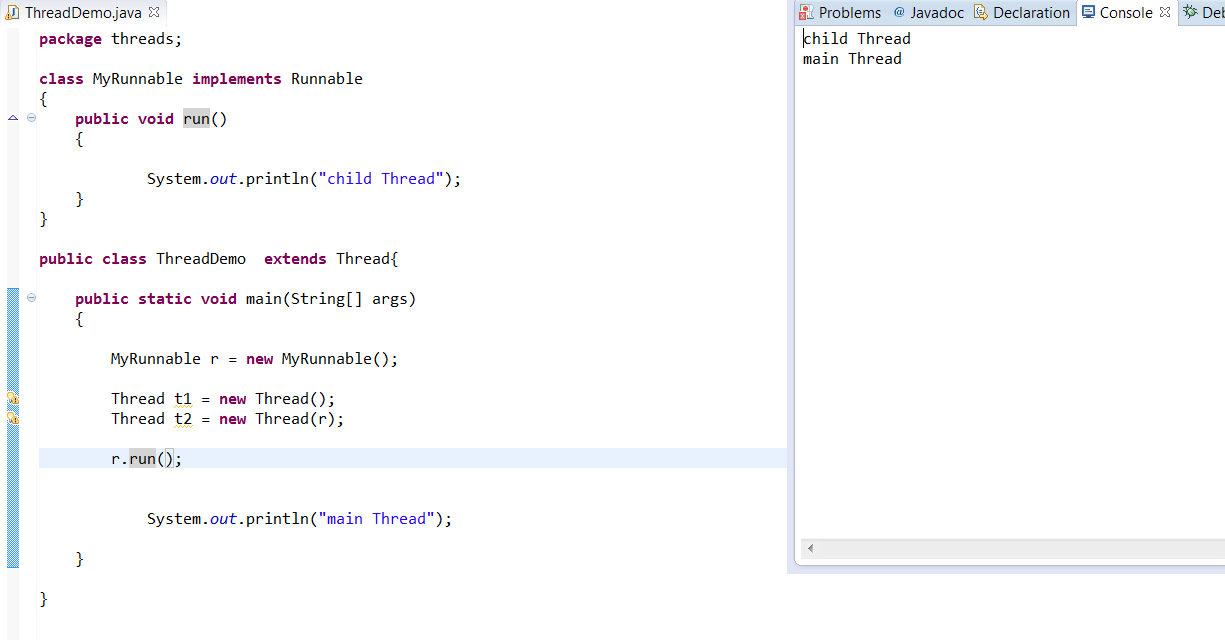
we will get compile time error saying start() is not available in MyRunnable class.

CE: cannot find symbol

symbol: method start()

location: class MyRunnable

Case 6:



No new Thread will be created and MyRunnable run() method will be executed just like a normal method call.

Q: In which of the above cases a new Thread will be created.

A: t1.start() and t2.start()

Q: In which of the above cases MyRunnable class run() method will be executed just like a normal method call.

A: t2.run() and r.run()

**Best Approach to define a Thread:**

Here we have two approaches to define Thread.

Approach 1: by extending Thread class

Class MyThread extends Thread

Approach 2: by implementing Runnable interface

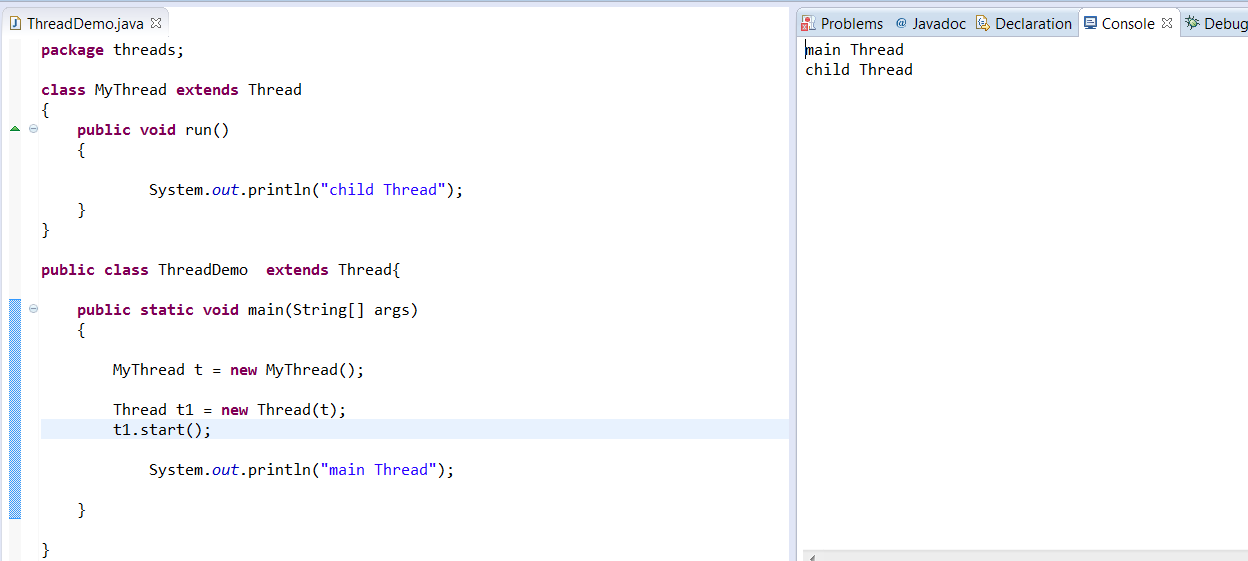
Class MyRunnable implements Runnable

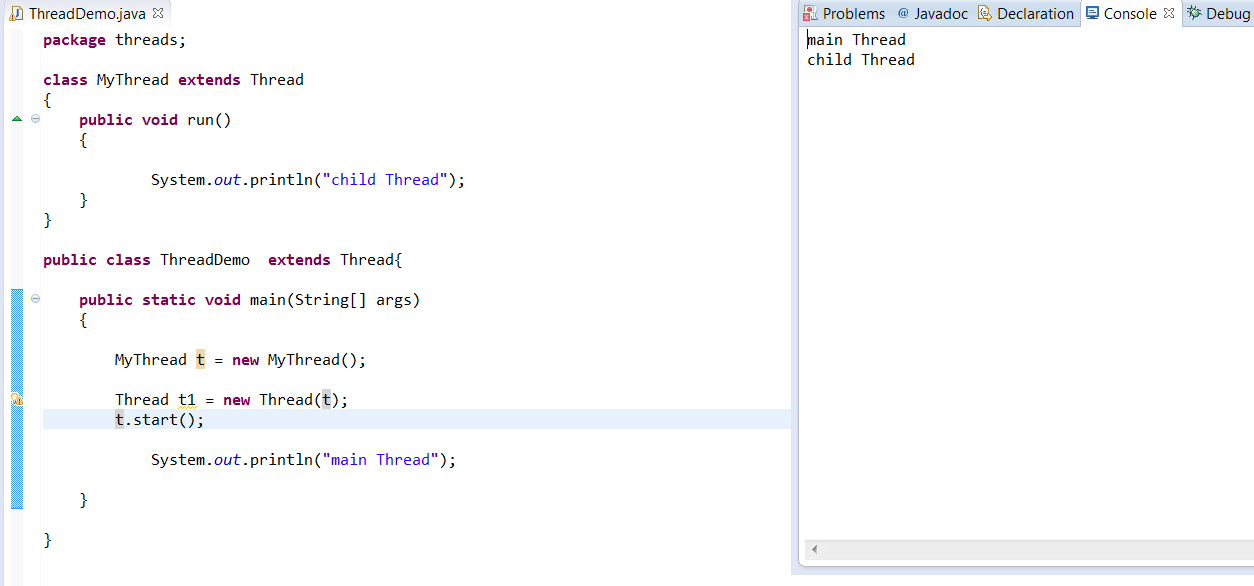
Among these two approaches , 2nd approach is recommended to use because,

In the 1st approach as our class is extending Thread class, there is no way of extending another class. (As multiple inheritance is not supported in Java).

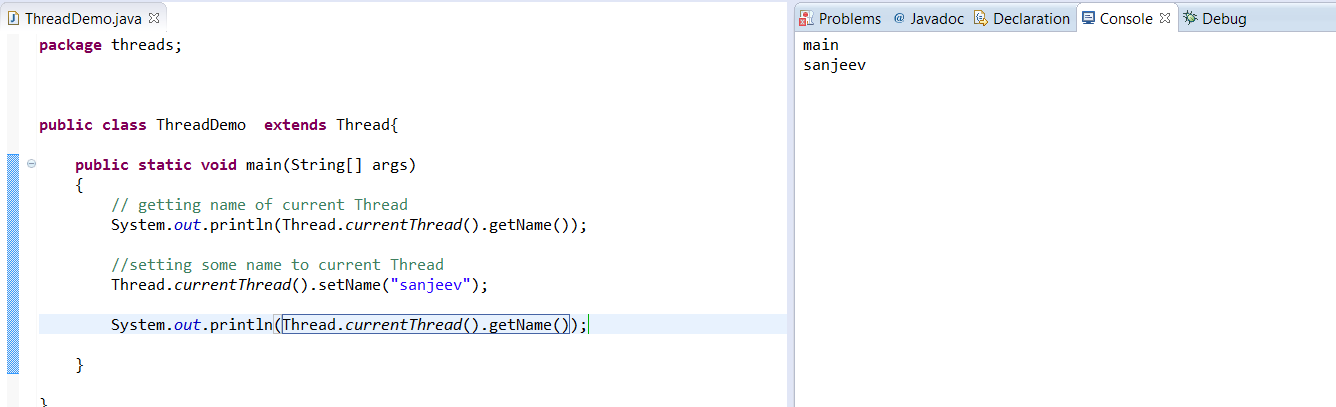
But in 2nd approach we can extend some other class also while implementing Runnable interface.

**Another way of creating Thread(Durga sir way ☺ , not recommended to use)**

****

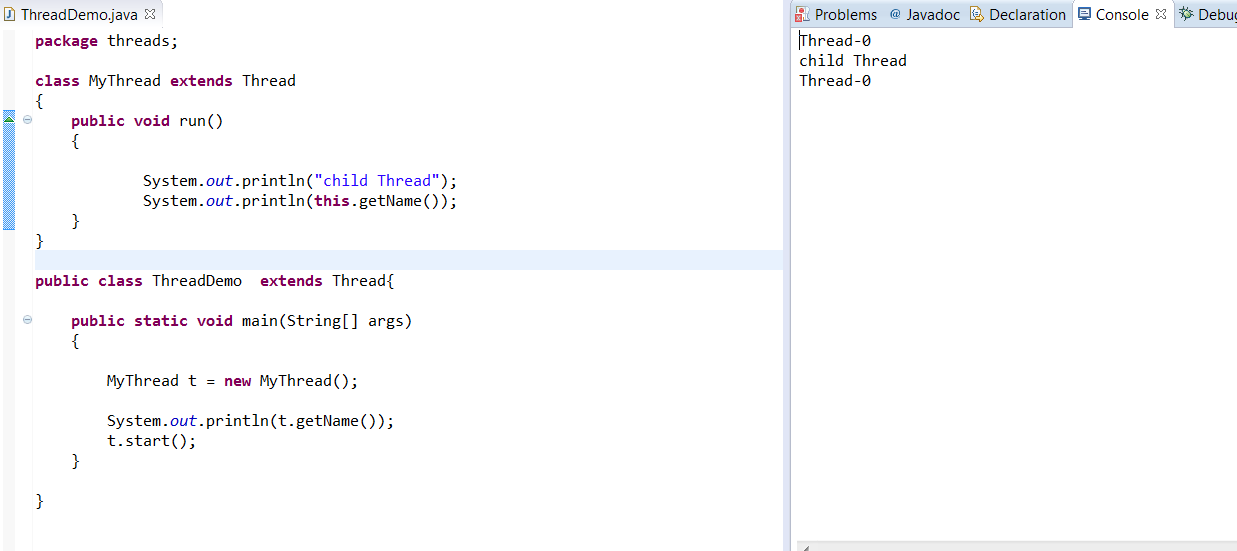
****

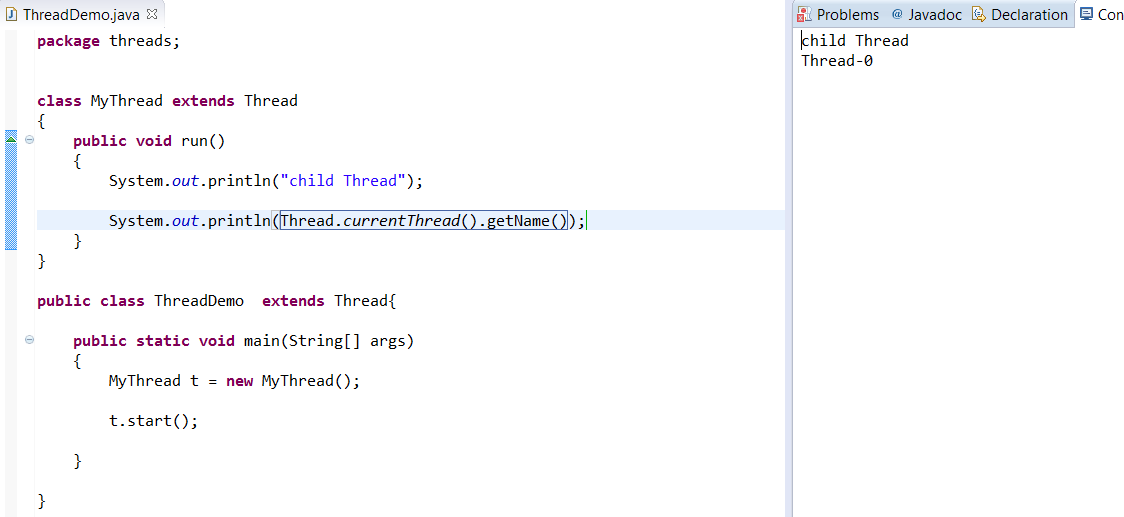
**Getting and Setting Thread name:**

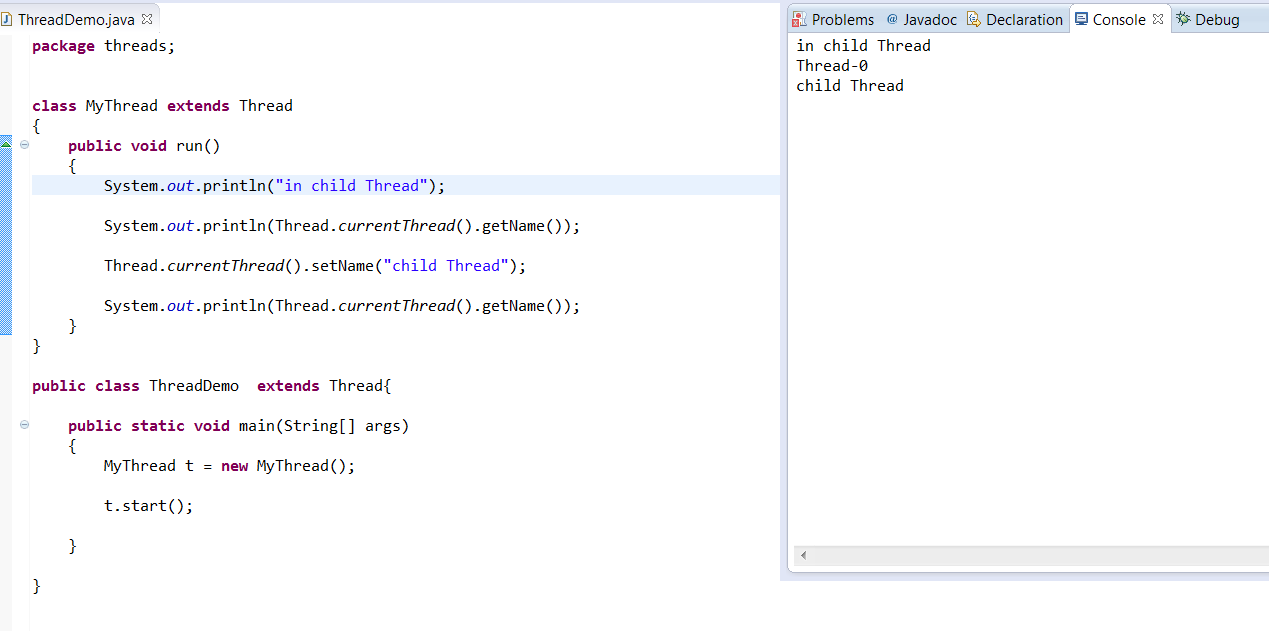
****

We can get current executing Thread reference by using the following method of Thread class

public static Thread currentThread();

****

****

****

**Thread Priority:**

Every Thread in Java has some priority. But the range of priorities are 1-10 (1 is least and 10 is highest).

Thread class defines the following constants to define some standard priorities.

1. Thread.MIN\_PRIORITY (1)
2. Thread.NORM\_PRIORITY (5)
3. Thread. MAX\_PRIORITY (10)

There are no constants like Thread.LOW\_PRIORITY and Thread.HIGH\_PRIORITY.

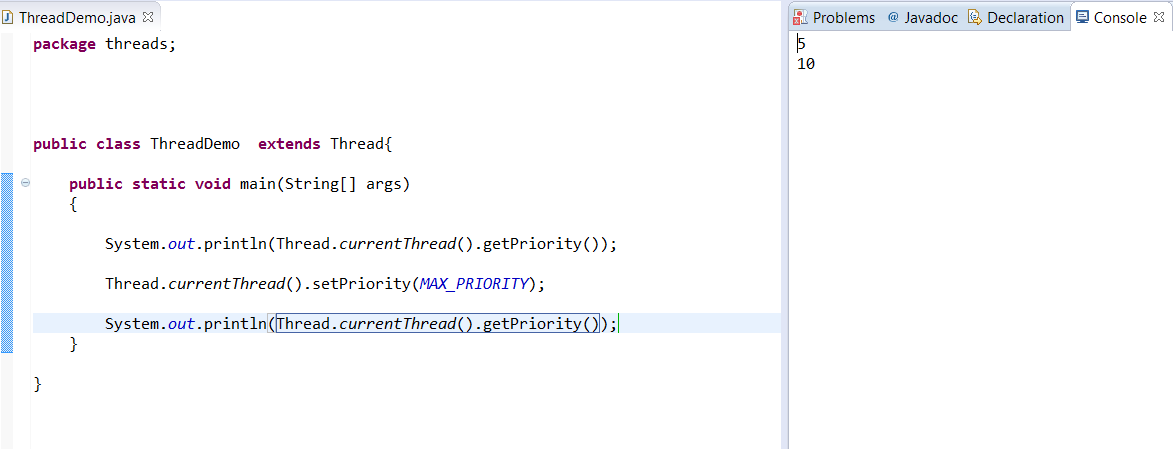
Thread scheduler will use these priorities while allocating CPU.

The Thread which is having high priority will get chance first.

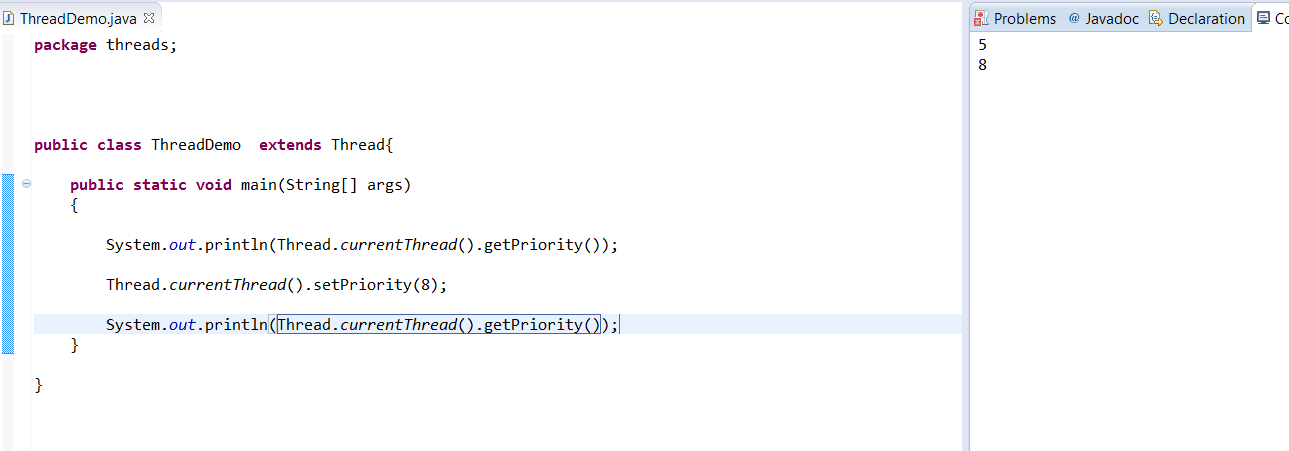
If two Threads having the same priority, then we can’t expect exact execution order and it is dependent on Thread scheduler.

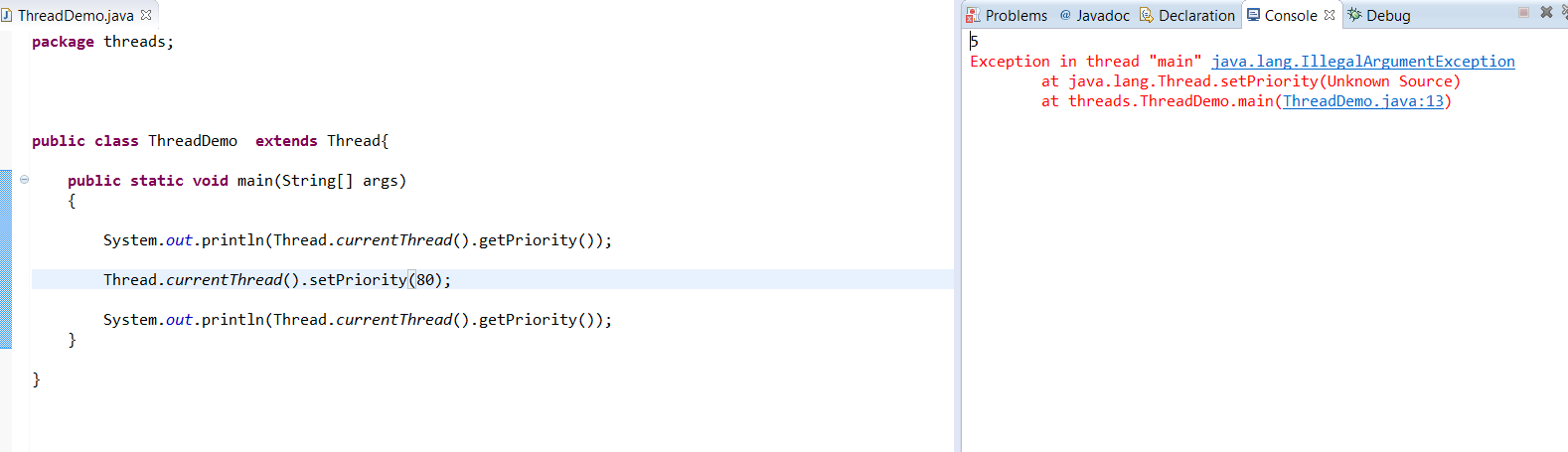
The default priority of only for main Thread is 5.

But for the remaining threads it is inheriting from parent i.e., whatever the priority does parent has the same will come to the child.









Priority is b/w 1-10, if we are giving other than this, we will get IllegalArgumentException.