**MultiTasking** : Executing tasks simultaneously is a concept of MultiTasking. There are two types of MultiTasking: 1. Process based 2. Thread based. Advantage is to increase performance by reducing processing time.

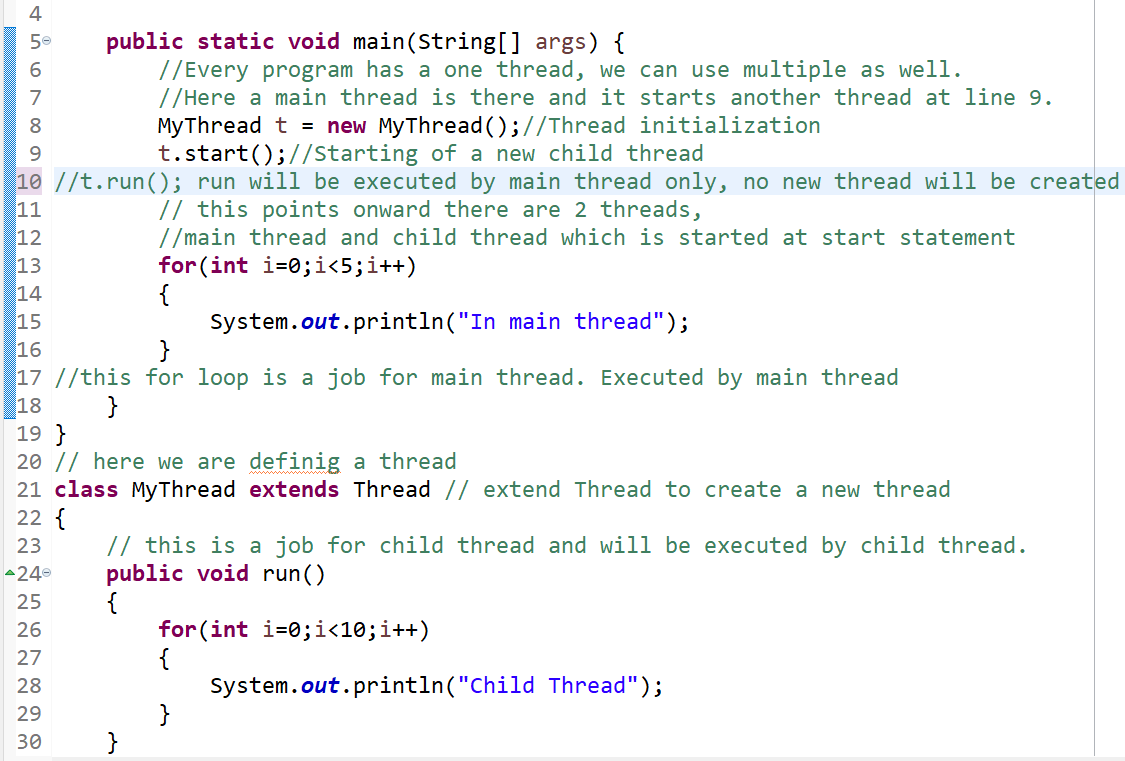
1. Process based : Executing several tasks simultaneously where each task is a separate, independent process is called Process based multitasking. Ex. : Writing a java program and simultaneously download a file from internet. Process based multitasking is best suitable at O.S. level.
2. Thread based : Executing several tasks simultaneously where each task is a separate, independent part of a **same program.** This type of multitasking is called thread-based multitasking. Here each independent part is called as **thread**. Thread based multitasking is best suitable at Programming level. Ex. : There is a programs of 10,000 lines. 1st 5k lines are independent of last 5k lines. Then two threads executes 1st and last 5k lines parallelly. The main important application areas of **multi threading** are : to develop multi media graphics, to develop animations, to develop video games, to develop web and application servers etc.

**Thread** : is a separate flow of execution. For every thread a separate independent job is there.

**How to create / define a thread** : We can define a thread by 2 ways :

1. By Extending Thread class
2. By implementing Runnable() interface

1.**By Extending Thread class** :



**Thread scheduler** : Is a part of JVM. It is responsible to schedule threads. i.e. if multiple threads are waiting to get the chance of execution then in which order threads will be executed is decided by thread scheduler.

We can’t expect exact algorithm followed by thread scheduler. It varies from JVM to JVM. Hence, we can’t expect thread execution order and exact output, hence whenever situation comes to multithreading, there is no guarantee for exact output. But we can provide several possible outputs.

**Difference between t.start() and t.run()** in above program : t.run() > A new thread will **not** be created, and run() method will be executed just like a normal method call by main thread only. If we use t.start(); > A new thread will be created which is responsible for the execution of Run() method.

**Importance of thread class Start() method** : Thread class start method is responsible to register the thread with thread scheduler and all other mandatory activities. Hence, without executing thread class start method, there is no chance of starting a new thread in Java. Due to this, thread class start method is considered as hart of multi-threading. It also invokes **run**() method. It is not recommended to override Start method.

Start()

{

1. Register this thread with thread scheduler

2. Perform all other mandatory activities

3. **Invoke run() method**

}

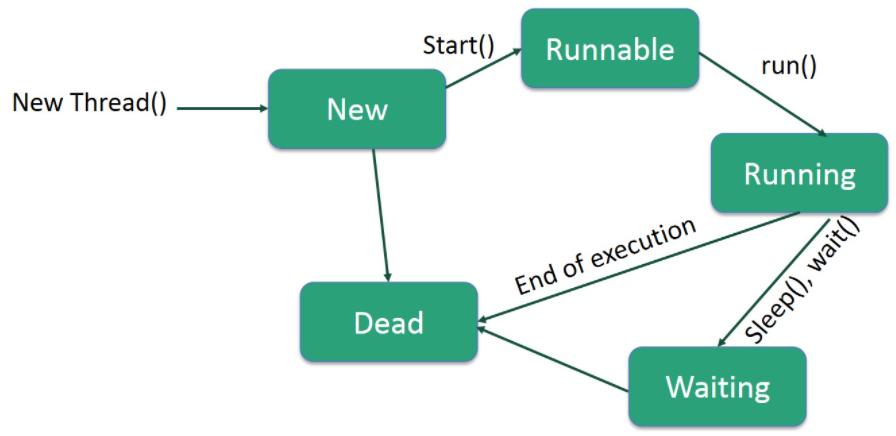
**IMP :**

**Overloading of Run() method** : Is always possible, but thread class start method can invoke no argument run method of thread class. The other overloaded method we have to call explicitly like a normal method call. If we are not overloading Run method, then Thread class run method will be executed which has **empty** implementation. Hence, we won’t get any output.

**Note** : It is highly recommended to override run method otherwise don’t go for multithreading concept.

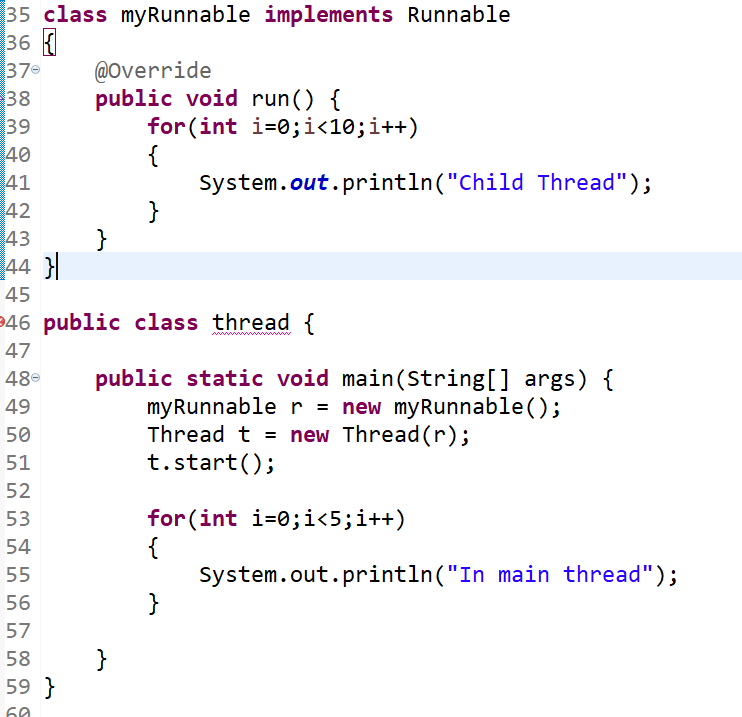
**Thread lifecycle** :

1. Mythread t = new mythread(); (thread is in **new / Born** state)
2. t.start(); (**Ready / Runnable** state)
3. Once thread scheduler allocates a process to thread (It enter in **Running** state)
4. Once run method completes, (it enters in **dead** state)



**Note** : After starting a thread, If we are trying to re-start a same thread, then we will get runtime exception saying illegal thread state exception.

**Implementing thread by implementing Runnable interface :** We can define a thread by implementing runnable interface. Runnable interface present in Java.lang package and it contains only one method : public void run() method.



Which approach is best to define a thread ?

Ans : Among two ways of defining a thread, **implements runnable approach is recommended**. In the first approach our class always extends Thread class, there is no chance of extending any other class. Hence, we are missing inheritance benefit. But in the 2nd approach, while implementing runnable interface, we can extend any other class.

Thread Methods

Following is the list of important methods available in the Thread class.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public void start()**  Starts the thread in a separate path of execution, then invokes the run() method on this Thread object. |
| 2 | **public void run()**  If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object. |
| 3 | **public final void setName(String name)**  Changes the name of the Thread object. There is also a  **getName()** method for retrieving the name. |
| 4 | **public final void setPriority(int p)**  Sets the priority of this Thread object. The possible values of **P** are between 1 and 10.  **getPriority()** It returns the priority of the thread. |
| 5 | **public final void setDaemon(boolean on) :** Thread that is executing at background called Daemon thread ex. : garbagecollector  A parameter of true denotes this Thread as a daemon thread.   |  |  | | --- | --- | | [**isDaemon()**](https://www.javatpoint.com/java-thread-isdaemon-method) | It tests if the thread is a daemon thread. | |
| 6 | **public final void join(long millisec)**  The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes. |
| 7 | **public void interrupt()**  Interrupts this thread, causing it to continue execution if it was blocked for any reason. |
| 8 | **public final boolean isAlive()**  Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion. |

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public static void yield()**  Causes the currently running thread to pause to any other threads of the same priority that are waiting to be scheduled. |
| 2 | **public static void sleep(long millisec)**  Causes the currently running thread to block for at least the specified number of milliseconds. |
| 3 | **public static boolean holdsLock(Object x)**  Returns true if the current thread holds the lock on the given Object. |
| 4 | **public static Thread currentThread()**  Returns a reference to the currently running thread, which is the thread that invokes this method. |
| 5 | **public static void dumpStack()**  Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application. |

We can prevent thread execution using following methods : 1. Yield 2. Join 3. Sleep

**Synchronization** : If multiple threads trying to operate on same java object simultaneously then there is a chance of data inconsistency problem. To overcome this problem, we should go for synchronized keyword. If a method or block is declared as synchronized, then at a time only one thread is allowed to execute that method or block on the given object. So that data inconsistency problem will be resolved.

The main advantage of synchronized keyword is we can resolve data inconsistency problems. But, the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problem, hence if there is no specific requirement, then it is not recommended to use synchronized keyword. Internally synchronization is implemented by using **LOCK**. Every object in java has a unique lock Whenever we are using synchronized keyword then only lock concept will come in picture. If a thread wants to execute synchronized method on the given object 1st it has to get lock of that object. Once thread got the lock, then it is allowed to execute any synchronized method on that object. Once method execution completes, automatically thread releases a lock. Acquiring and releasing lock is internally taken care by JVM and Program is not responsible for this activity.

While a thread executing synchronized method on the given object the remaining threads are not allowed to execute any synchronized method simultaneously on the same object. But, remaining threads are allowed to execute non-synchronized methods simultaneously.

Class x

{

Synch m1()

Synch m2()

m3()

}

If t1 starts execution of m1, t2 comes to execute m1(), it goes to waiting state. t3 comes to execute m2(), it goes to waiting state. If t4 comes to execute m3(), it will get the chance immediately since ot is not synchronized. Lock concept is implemented based on Object but not based on method.