

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing.

Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Process vs. threads

A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

A thread is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Advantage of Java Multithreading

1. It doesn't block the user because threads are independent and you can perform multiple operations at same time.
2. You can perform many operations together so it saves time.
3. Threads are independent so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

Multithreading

1. Process-based Multitasking (Multiprocessing)

- Each process has its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2. Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

According to sun, there is only 4 states in thread life cycle in java **new, runnable, non-runnable and terminated.**

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread ()
- Thread (String name)

- Thread (Runnable r)
- Thread (Runnable r, String name)

Runnable interface:

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

- **public void run ()**: is used to perform action for a thread.

Commonly used methods of Thread class:

1. **public void run()**: is used to perform action for a thread.
2. **public void start()**: starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void sleep(long milliseconds)**: Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join()**: waits for a thread to die.
5. **public void join(long milliseconds)**: waits for a thread to die for the specified milliseconds.
6. **public int getPriority()**: returns the priority of the thread.
7. **public int setPriority(int priority)**: changes the priority of the thread.
8. **public String getName()**: returns the name of the thread.
9. **public void setName(String name)**: changes the name of the thread.
10. **public Thread currentThread()**: returns the reference of currently executing thread.
11. **public int getId()**: returns the id of the thread.
12. **public Thread.State getState()**: returns the state of the thread.
13. **public boolean isAlive()**: tests if the thread is alive.
14. **public void yield()**: causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend()**: is used to suspend the thread(deprecated).
16. **public void resume()**: is used to resume the suspended thread(deprecated).
17. **public void stop()**: is used to stop the thread(deprecated).
18. **public boolean isDaemon()**: tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b)**: marks the thread as daemon or user thread.

20. **public void interrupt()**: interrupts the thread.
21. **public boolean isInterrupted()**: tests if the thread has been interrupted.
22. **public static boolean interrupted()**: tests if the current thread has been interrupted.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

The screenshot displays an IDE with the following components:

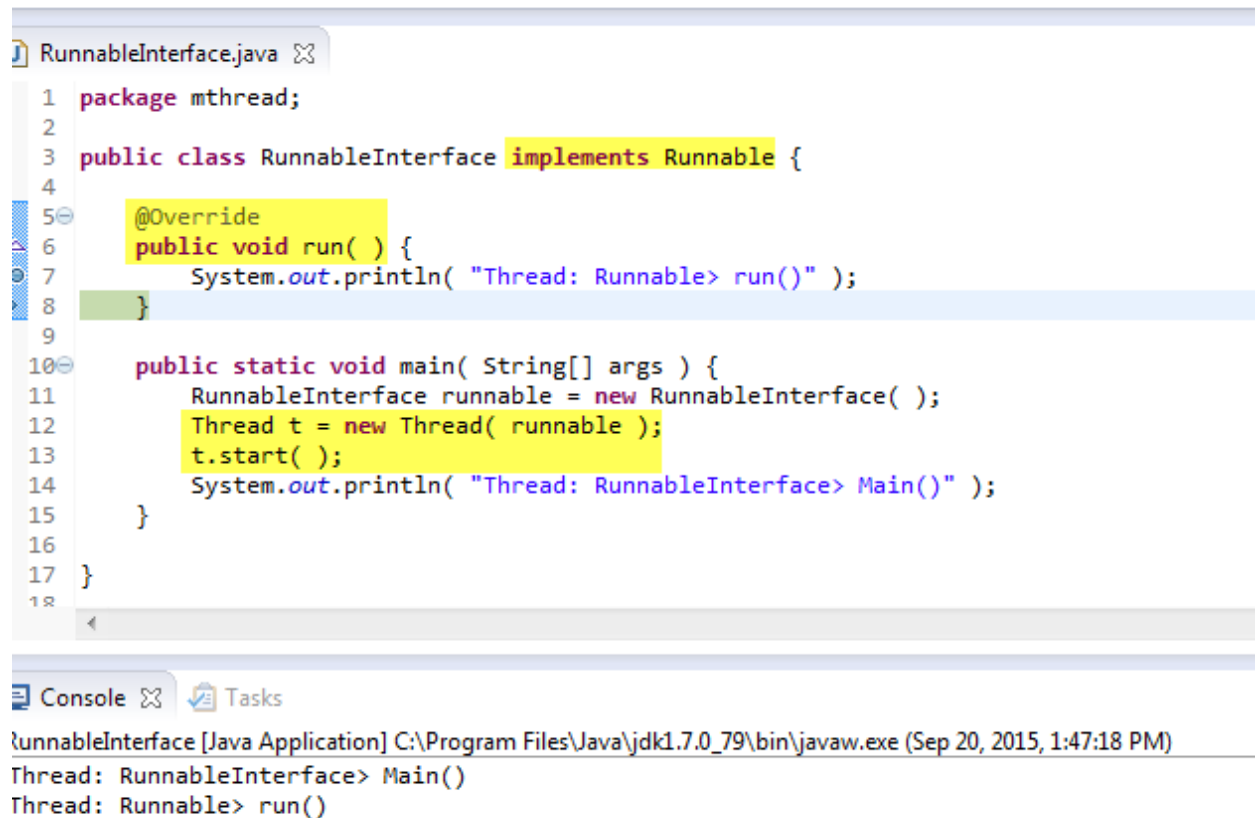
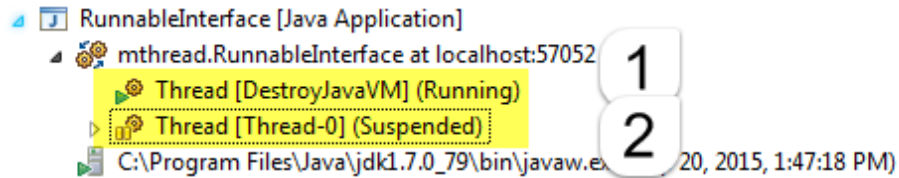
- ThreadView:** Shows the execution of `ThreadClass` [Java Application] at `localhost:57008`. It lists three threads:
 - `Thread [DestroyJavaVM] (Running)` (highlighted in yellow)
 - `Thread [Thread-0] (Suspended)` (highlighted in yellow)
 - `ThreadClass.run() line: 8` (highlighted in yellow)
 Two callout boxes labeled '1' and '2' point to the first and second threads respectively.
- Source Editor:** Shows the source code for `ThreadClass.java` and `Thread.class`. The code is as follows:


```

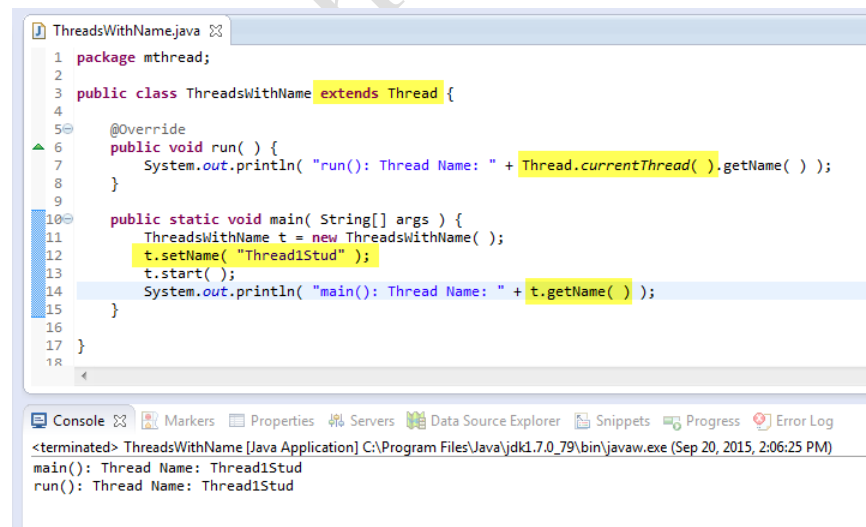
1 package mthread;
2
3 public class ThreadClass extends Thread {
4
5     @Override
6     public void run( ) {
7         System.out.println( "ThreadClass>Run> HELLO" );
8     }
9
10    public static void main( String[] args ) {
11        ThreadClass t = new ThreadClass( );
12        t.start( );
13        System.out.println( "ThreadClass: Main" );
14    }
15 }
16
```
- Console:** Shows the output of the application:


```

ThreadClass [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 1:39:27 PM)
ThreadClass: Main
ThreadClass>Run> HELLO
      
```



The currentThread() method & naming a Thread:



Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run. There is no guarantee that which runnable thread will be chosen to run by the thread scheduler. Only one thread at a time can run in a single process. The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Thread Priority

Default priority of a thread is 5 (NORM_PRIORITY).

The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

```
thread.setPriority( Thread.MAX_PRIORITY );
```

```
Thread.currentThread( ).getPriority();
```

Daemon Thread in Java

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

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

public void setDaemon(boolean status): is used to mark the current thread as daemon thread or user thread.

public boolean isDaemon(): is used to check that current is daemon.

Thread.Sleep() method in java:

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

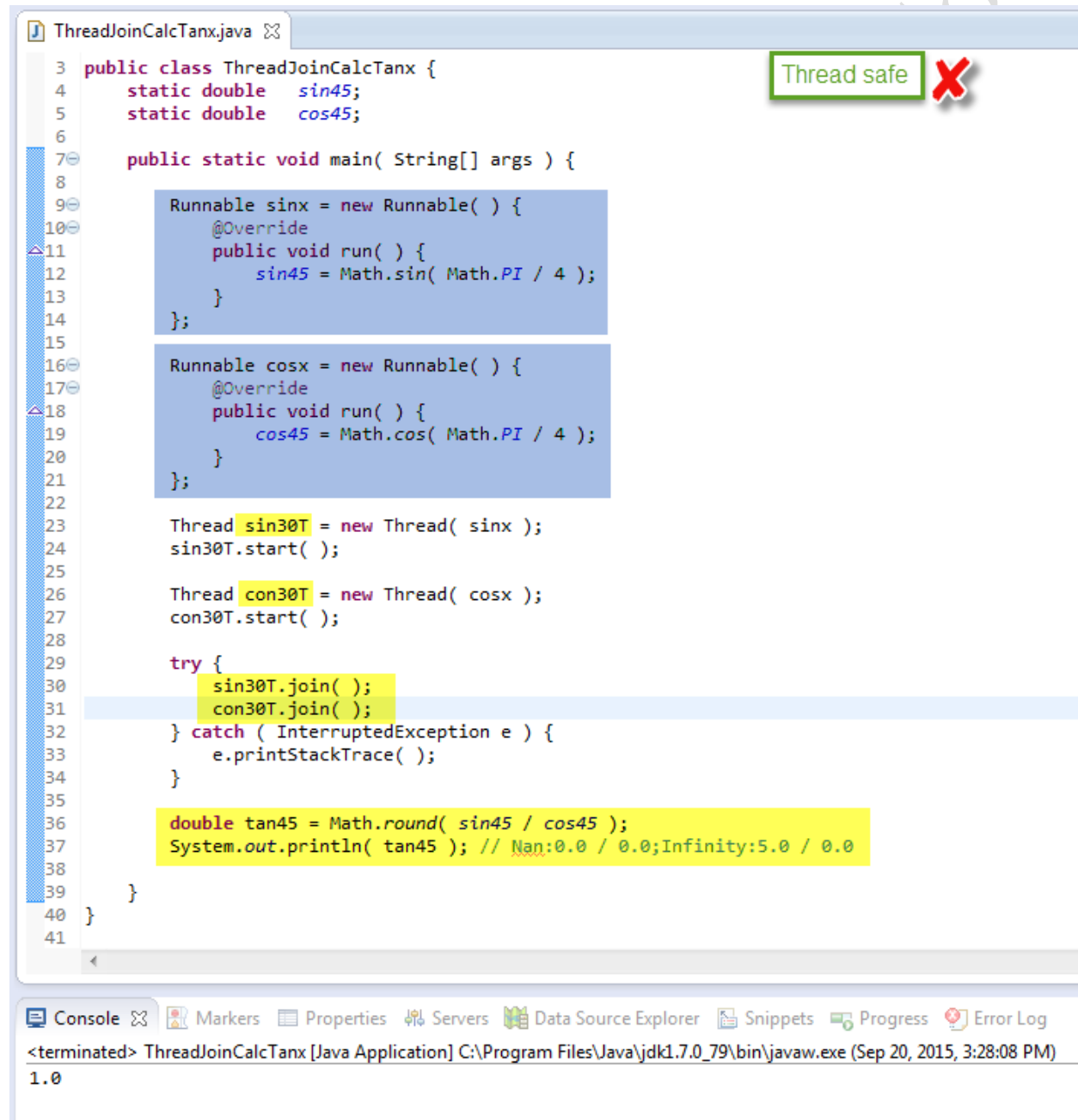
Thread.sleep(500); will halt thread for 500 milli second.

Thread can not be started twice java.lang.IllegalThreadStateException will be thrown.

Do not call `run()` method instead of `start()` method. Each thread starts in a separate call stack. Invoking the `run()` method from main thread, the `run()` method goes onto the current call stack rather than at the beginning of a new call stack.

Thread.join() method:

The `join()` method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task. Example: Calculate $\tan(x) = \sin(x)/\cos(x)$ Where `sin` and `cos` value calculation should be completed before we calculate `tan`



```

ThreadJoinCalcTanx.java
3  public class ThreadJoinCalcTanx {
4      static double  sin45;
5      static double  cos45;
6
7      public static void main( String[] args ) {
8
9          Runnable sinx = new Runnable( ) {
10             @Override
11             public void run( ) {
12                 sin45 = Math.sin( Math.PI / 4 );
13             }
14         };
15
16         Runnable cosx = new Runnable( ) {
17             @Override
18             public void run( ) {
19                 cos45 = Math.cos( Math.PI / 4 );
20             }
21         };
22
23         Thread sin30T = new Thread( sinx );
24         sin30T.start( );
25
26         Thread con30T = new Thread( cosx );
27         con30T.start( );
28
29         try {
30             sin30T.join( );
31             con30T.join( );
32         } catch ( InterruptedException e ) {
33             e.printStackTrace( );
34         }
35
36         double tan45 = Math.round( sin45 / cos45 );
37         System.out.println( tan45 ); // Nan:0.0 / 0.0;Infinity:5.0 / 0.0
38
39     }
40 }
41

```

Thread safe X

Console X Markers Properties Servers Data Source Explorer Snippets Progress Error Log

<terminated> ThreadJoinCalcTanx [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 3:28:08 PM)

1.0

```

1 package mthread;
2
3 public class ThreadJoinPrintValues extends Thread {
4     public void run() {
5         for ( int i = 1; i <= 5; i++ ) {
6
7             try {
8                 Thread.sleep( 500 );
9             } catch ( Exception e ) {
10                 e.printStackTrace();
11             }
12
13             System.out.println( i );
14         }
15     }
16
17     public static void main( String args[] ) {
18         ThreadJoinPrintValues t1 = new ThreadJoinPrintValues( );
19         ThreadJoinPrintValues t2 = new ThreadJoinPrintValues( );
20         ThreadJoinPrintValues t3 = new ThreadJoinPrintValues( );
21         t1.start();
22
23         try {
24             t1.join(); // t1.join( 1500 );start t2 after 1500 sec
25         } catch ( Exception e ) {
26             e.printStackTrace();
27         }
28
29         t2.start();
30         t3.start();
31     }
32 }
33

```

```

<terminated> ThreadJoinPrintValues [Java /
1
2
3
4
5
1
2
2
3
3
4
4
5
5

```

Shutdown Hook

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

```

1 package mthread;
2
3 public class ShutdownHookExample {
4     public static void main( String[] args ) {
5
6         Thread t1 = new Thread( ) {
7             @Override
8             public void run() {
9                 System.out.println( "Shut down hook run this method. " );
10            }
11        };
12
13        Runtime r = Runtime.getRuntime();
14        r.addShutdownHook( t1 );
15
16        try {
17            System.out.println( "Press ctrl+c in command window to exit program." );
18            Thread.sleep( 3000 );
19        } catch ( InterruptedException e ) {
20            e.printStackTrace();
21        }
22
23        System.out.println( "Calling System.exit(...)" );
24        System.exit( 0 ); // exit program in the middle
25
26        System.out.println( "After 3 second.." );
27    }
28 }
29
30

```

```

<terminated> ShutdownHookExample [Java Application] C:\Pr
Press ctrl+c in command window to exit program.
Calling System.exit()...
Shut down hook run this method.

```


Java Thread Pool

Java Thread pool represents a group of worker threads that are waiting for the job and reuse many times. Better performance It saves time because there is no need to create new thread. It is used in Servlet and JSP where container creates a thread pool to process the request. `ExecutorService executor = Executors.newFixedThreadPool(5); executor.execute(r); //Runnable r`

Thread Group in Java

Every Java thread is a member of a thread group. Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually. For example, you can start or suspend all the threads within a group with a single method call. Java thread groups are implemented by the `ThreadGroup` class in the `java.lang` package.

```
ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");
```

```
Thread myThread = new Thread(myThreadGroup, "a thread for my group");
```

Don't use `ThreadGroup` for new code. Use the `Executor` stuff in `java.util.concurrent` instead.

Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Example: When you have two threads that are reading and writing to the same 'resource', say a variable named `foo`, you need to ensure that these threads access the variable in an atomic way. Without the `synchronized` keyword, your thread 1 may not see the change thread 2 made to `foo`, or worse, it may only be half changed. This would not be what you logically expect.

```

SynchronizeFbLike.java
1 package mthread;
2
3 public class SynchronizeFbLike {
4
5     public static void main( String[] args ) {
6
7         /* Facebook Page: Everest, Current Likes: 500 */
8         final FacebookLike everestFbPagePiclike = new FacebookLike( 500 );
9
10        Thread user1 = new Thread( ) {
11            public void run( ) {
12                everestFbPagePiclike.plusOne( );
13            }
14        };
15
16        Thread user2 = new Thread( ) {
17            public void run( ) {
18                everestFbPagePiclike.plusOne( );
19            }
20        };
21
22        Thread user3 = new Thread( ) {
23            public void run( ) {
24                everestFbPagePiclike.plusOne( );
25            }
26        };
27
28        Thread user4 = new Thread( ) {
29            public void run( ) {
30                everestFbPagePiclike.plusOne( );
31            }
32        };
33
34        /* User1,2,3,4 hit like in Everest Facebook Page */
35        user1.start( );
36        user2.start( );
37        user3.start( );
38        user4.start( );
39    }
40}

FacebookLike.java
1 package mthread;
2
3 public class FacebookLike {
4
5     public Integer likes = 0;
6
7     /* set current Page likes */
8     public FacebookLike( Integer likes ) {
9         this.likes = likes;
10    }
11
12    /* Synchronized method call solve problem of Multi-thread problem */
13    public synchronized void plusOne( ) {
14        likes++;
15        System.out.println( Thread.currentThread().getName() + " Likes: " + likes );
16        try {
17            Thread.sleep( 100 );
18        } catch ( InterruptedException e ) {
19            e.printStackTrace();
20        }
21    }
22}

Console
<terminated> SynchronizeFbLike [Java Application] C:\Pr
Thread-0 Likes: 501
Thread-2 Likes: 502
Thread-3 Likes: 503
Thread-1 Likes: 504

Without synchronized keyword in above method
  
```