java.lang.Thread Class API Guide

In this article, we will learn about Thread Class and it's methods with examples. Thread creates a new thread of execution. It implements the `Runnable` interface. The `Java Virtual Machine`allows an application to have multiple threads of execution running concurrently.

Thread Class Constructors

1. `Thread()` - Allocates a new `Thread` object.
2. `Thread(Runnable target)` - Allocates a new `Thread` object.
3. `Thread(Runnable target, String name)` - Allocates a new Thread object.
4. `Thread(String name)` - Allocates a new Thread object.
5. `Thread(ThreadGroup group, Runnable target)` - Allocates a new Thread object.
6. `Thread(ThreadGroup group, Runnable target, String name) -` Allocates a new Thread object so that it has targeted as its run object, has the specified name as its name, and belongs to the thread group referred to by the group.
7. `Thread(ThreadGroup group, Runnable target, String name, long stackSize)` - Allocates a new Thread object so that it has targeted as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size.
8. `Thread(ThreadGroup group, String name)` - Allocates a new Thread object.

Thread Class Methods

Let's demonstrates the usage of few important methods of `java.lang.Thread` class.

Thread.sleep() Method

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.

Thread.sleep() Method Example

In this example, we have created and started two threads `thread1` and `thread2`. Note that we have used both overloaded versions of `sleep()` methods in this example.

```
Thread.sleep(1000);

Thread.sleep(1000, 500);

/**
```

```
 * thread sleep method examples
 * @author Ramesh fadatare
 *
 */
public class ThreadSleepExample {
   public static void main(final String[] args) {
        System.out.println("Thread main started");
        final Thread thread1 = new Thread(new WorkerThread());
        thread1.setName("WorkerThread 1");
        final Thread thread2 = new Thread(new WorkerThread());
        thread1.setName("WorkerThread 2");
        thread1.start();
        thread2.start();
        System.out.println("Thread main ended");
   }
}


class WorkerThread implements Runnable {


    @Override
    public void run() {
         for (int i = 0; i < 5; i++) {
         try {
              Thread.sleep(1000);
```

```
                Thread.sleep(1000, 500);

                System.out.println("[" + Thread.currentThread().getName() + "] Message "
+ i);

            } catch (final InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}
```

Output:

```
Thread main started
Thread main ended
[WorkerThread 2] Message 0
[Thread-1] Message 0
[WorkerThread 2] Message 1
[Thread-1] Message 1
[WorkerThread 2] Message 2
[Thread-1] Message 2
[WorkerThread 2] Message 3
[Thread-1] Message 3
[WorkerThread 2] Message 4
[Thread-1] Message 4
```

Note that sleep() method throws InterruptedException exception, when another thread interrupts the current thread while sleep is active.

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.

Thread join() Method Example: We will first create a Task which will calculate the sum of 1-5 numbers(maintained in for loop). In the main thread lets create 4 tasks:

```
final Task task1 = new Task(500l);

final Task task2 = new Task(1000l);

final Task task3 = new Task(2000l);

final Task task4 = new Task(50l);
```

Now, let's create 4 threads to run above 4 tasks:

```
final Thread thread1 = new Thread(task1);

final Thread thread2 = new Thread(task2);

final Thread thread3 = new Thread(task3);

final Thread thread4 = new Thread(task4);
```

Assign name to each and start all the 4 threads:

```
thread1.setName("thread-1");

thread2.setName("thread-2");

thread3.setName("thread-3");

thread4.setName("thread-4");

thread1.start();

thread2.start();

thread3.start();

thread4.start();
```

In this example, when the target thread finishes the sum, the caller thread (main) wakes up and calls the task.getSum() method which will certainly contain the total sum as the target thread has already finished its job.

The *`task4`* has a small sleep time and therefore it finishes the sum before the others. Hence, the main thread calls the *`thread4.join()`* but immediately returns to its execution as the *`thread4`* is finished.

```java
/**
 * This class demonstrate the how join method works with an example.
 * @author Ramesh Fadatare
 *
 */
public class ThreadJoinExample {
    public static void main(final String[] args) throws InterruptedException {
         System.out.println("Thread main started");

        final Task task1 = new Task(500l);
        final Task task2 = new Task(1000l);
        final Task task3 = new Task(2000l);
        final Task task4 = new Task(50l);
        final Thread thread1 = new Thread(task1);
        final Thread thread2 = new Thread(task2);
        final Thread thread3 = new Thread(task3);
        final Thread thread4 = new Thread(task4);
        thread1.setName("thread-1");
        thread2.setName("thread-2");
        thread3.setName("thread-3");
        thread4.setName("thread-4");
        thread1.start();
        thread2.start();
        thread3.start();
```

```
        thread4.start();

        System.out.println("[" + Thread.currentThread().getName() + "] waiting for "
+ thread1.getName());
        thread1.join();
        System.out.println(thread1.getName() + " finished! Result: " +
task1.getSum());

        System.out.println("[" + Thread.currentThread().getName() + "] waiting for "
+ thread2.getName());
        thread2.join();
        System.out.println(thread2.getName() + " finished! Result: " +
task2.getSum());

       System.out.println("[" + Thread.currentThread().getName() + "] waiting for " +
thread3.getName());
       thread3.join();
       System.out.println(thread3.getName() + " finished! Result: " +
task3.getSum());

      // As thread-4 already finished (smaller sleep time), the join call only
immediately
      // returns the control to the caller thread
      System.out.println("[" + Thread.currentThread().getName() + "] waiting for " +
thread4.getName());
      thread4.join();
      System.out.println(thread4.getName() + " finished! Result: " + task4.getSum());
```

```
        System.out.println("Thread main finished");
    }
}


class Task implements Runnable {
        private long sleep;
        private int sum;

        public Task(final long sleep) {
            this.sleep = sleep;
        }

        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                System.out.println("[" + Thread.currentThread().getName() + "] Adding
" + i);
                sum += i;
                try {
                    Thread.sleep(sleep);
                } catch (final InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    public int getSum() {
        return this.sum;
    }
}
```

Output:

```
Thread main started
[thread-1] Adding 1
[thread-2] Adding 1
[thread-3] Adding 1
[main] waiting for thread-1
[thread-4] Adding 1
[thread-4] Adding 2
[thread-4] Adding 3
[thread-4] Adding 4
[thread-4] Adding 5
[thread-1] Adding 2
[thread-1] Adding 3
[thread-2] Adding 2
[thread-1] Adding 4
[thread-1] Adding 5
[thread-3] Adding 2
[thread-2] Adding 3
thread-1 finished! Result: 15
[main] waiting for thread-2
[thread-2] Adding 4
[thread-2] Adding 5
[thread-3] Adding 3
thread-2 finished! Result: 15
[main] waiting for thread-3
[thread-3] Adding 4
[thread-3] Adding 5
thread-3 finished! Result: 15
[main] waiting for thread-4
thread-4 finished! Result: 15
Thread main finished
```

Note that the output, main() thread finished it's executions last. Try to understand the example via looking into an output.

join() method throws an *InterruptedException* - if any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

Thread.interrupt() Method

An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

java.lang Thread class provides three *interrupt()* methods to work Interrupts properly.

1. void interrupt() - Interrupts this thread.
2. static boolean interrupted() - Tests whether the current thread has been interrupted.
3. boolean isInterrupted() - Tests whether this thread has been interrupted.

Thread.interrupt() method Example

```
public class TerminateTaskUsingThreadAPI {


 public static void main(final String[] args) {


  System.out.println("Thread main started");


  final Task task = new Task();
  final Thread thread = new Thread(task);
  thread.start();


  thread.interrupt();
```

```
  System.out.println("Thread main finished");
 }
}

class Task implements Runnable {
 @Override
 public void run() {
  for (int i = 0; i < 5; i++) {
   System.out.println("[" + Thread.currentThread().getName() + "] Message " + i);

   if (Thread.interrupted()) {
    System.out.println("This thread was interruped by someone calling this Thread.interrupt()");
    System.out.println("Cancelling task running in thread " + Thread.currentThread().getName());
    System.out.println("After Thread.interrupted() call, JVM reset the interrupted value to: " + Thread.interrupted());
    break;
   }
  }
 }
}
```

Output:

```
Thread main started
```

```
Thread main finished
[Thread-0] Message 0
This thread was interruped by someone calling this Thread.interrupt()
Cancelling task running in thread Thread-0
After Thread.interrupted() call, JVM reset the interrupted value to: false
```

Note that the here the task is being terminated, not the thread.

Thread is isAlive Method

java.lang.Thread class provides isAlive() method to test if this thread is alive or not. A thread is alive if it has been started and has not yet died.

Thread is isAlive Method Example

1. Let's create two threads

```
final Thread thread1 = new Thread(new MyTask());

final Thread thread2 = new Thread(new MyTask());
```

2. Before starting the threads with start() method, just print to check whether the threads are alive or not.

```
System.out.println("Thread1 is alive? " + thread1.isAlive());

System.out.println("Thread2 is alive? " + thread2.isAlive());
```

3. Start the threads and check again to check whether the threads are alive or not

```
thread1.start();

thread2.start();



while (thread1.isAlive() || thread2.isAlive()) {

 System.out.println("Thread1 is alive? " + thread1.isAlive());

 System.out.println("Thread2 is alive? " + thread2.isAlive());
```

```
 Thread.sleep(5001);

}
```

Let's put all together and the complete program:

```
public class CheckIfThreadIsAliveUsingThreadAPI {


 public static void main(final String[] args) throws InterruptedException {


   System.out.println("Thread main started");


   final Thread thread1 = new Thread(new MyTask());

   final Thread thread2 = new Thread(new MyTask());


   System.out.println("Thread1 is alive? " + thread1.isAlive());

   System.out.println("Thread2 is alive? " + thread2.isAlive());


   thread1.start();

   thread2.start();


   while (thread1.isAlive() || thread2.isAlive()) {

    System.out.println("Thread1 is alive? " + thread1.isAlive());

    System.out.println("Thread2 is alive? " + thread2.isAlive());

    Thread.sleep(5001);

   }
```

```
  System.out.println("Thread1 is alive? " + thread1.isAlive());
  System.out.println("Thread2 is alive? " + thread2.isAlive());

  System.out.println("Thread main finished");
 }
}

class MyTask implements Runnable {
 @Override
 public void run() {
  for (int i = 0; i < 5; i++) {
   System.out.println("[" + Thread.currentThread().getName() + "] Message " + i);
   try {
    Thread.sleep(200);
   } catch (final InterruptedException e) {
    e.printStackTrace();
   }
  }
 }
}
```

Output:

```
Thread main started
Thread1 is alive? false
```

```
Thread2 is alive? false
Thread1 is alive? true
Thread2 is alive? true
[Thread-0] Message 0
[Thread-1] Message 0
[Thread-1] Message 1
[Thread-0] Message 1
[Thread-1] Message 2
[Thread-0] Message 2
Thread1 is alive? true
Thread2 is alive? true
[Thread-0] Message 3
[Thread-1] Message 3
[Thread-1] Message 4
[Thread-0] Message 4
Thread1 is alive? false
Thread2 is alive? false
Thread main finished
```

Thread setPriority() Method

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

Three constants defined in Thread class:

1. public static int *MIN_PRIORITY*
2. public static int *NORM_PRIORITY*
3. public static int *MAX_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.

Set priority to a Thread Example

```
public class ThreadPriorityExample {

 public static void main(final String[] args) {

  final Runnable runnable = () -> {
```

```
    System.out.println("Running thread name : " + Thread.currentThread().getName() +
        " and it's priority : " + Thread.currentThread().getPriority());

  };

  final Thread thread1 = new Thread(runnable);
  final Thread thread2 = new Thread(runnable);
  final Thread thread3 = new Thread(runnable);
  final Thread thread4 = new Thread(runnable);
  thread1.setPriority(Thread.MIN_PRIORITY);
  thread2.setPriority(Thread.NORM_PRIORITY);
  thread3.setPriority(Thread.MAX_PRIORITY);
  thread4.setPriority(2);

  thread1.start();
  thread2.start();
  thread3.start();
  thread4.start();
 }
}
```

Output:

```
Running thread name : Thread-0 and it's priority : 1
Running thread name : Thread-1 and it's priority : 5
Running thread name : Thread-2 and it's priority : 10
```

```
Running thread name : Thread-3 and it's priority : 2
```

Thread setName() Method

The *java.lang.Thread* class provides methods to change and get the name of a thread. By default, each thread has a name i.e. *thread-0, thread-1* and so on. By we can change the name of the thread by using *setName()*method. The syntax of *setName() and getName()* methods are given below:

- *public String getName()*: is used to return the name of a thread.
- *public void setName(String name):* is used to change the name of a thread.

Thread class provides a static currentThread() - Returns a reference to the currently executing thread object.

Naming a Thread Example

```
/**
 *  Thread naming example using Thread class.
 *  @author Ramesh fadatare
 **/

public class ThreadExample {

 public static void main(final String[] args) {

  System.out.println("Thread main started");

  final Thread thread1 = new WorkerThread();
  thread1.setName("WorkerThread1");
```

```
    final Thread thread2 = new WorkerThread();
    thread2.setName("WorkerThread2");

    final Thread thread3 = new WorkerThread();
    thread3.setName("WorkerThread3");

    final Thread thread4 = new WorkerThread();
    thread4.setName("WorkerThread4");

    thread1.start();
    thread2.start();
    thread3.start();
    thread4.start();

    System.out.println("Thread main finished");
  }
}


class WorkerThread extends Thread {

  @Override
  public void run() {
    System.out.println("Thread Name :: " + Thread.currentThread().getName());
```

```
 }

}
```

Output:

```
Thread main started
Thread Name :: WorkerThread1
Thread Name :: WorkerThread2
Thread Name :: WorkerThread3
Thread main finished
Thread Name :: WorkerThread4
```

Reference

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html