Use of join() in thread in java

**Example Scenario -1**

public class Thread1 extends Thread {  
 public Thread1(String name) {  
 super(name);  
 }  
 @Override  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "<--->" + i);  
 try {  
 Thread.*sleep*(300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

public class TestTheadJoin {  
 public static void main(String[] args) throws Exception {  
 Thread th1 = new Thread1("First Thread");  
 Thread th2 = new Thread1("Second Thread");  
  
*// First th1 will run after th1 completes operation, th2 will run  
//* ***th1.start();*** *//* ***th1.join();*** *//* ***th2.start();*** *//* ***th2.join();*** *// Two threads will run alternately, each will run for 30 miliseconds* **th1.start();  
 th1.join(10);  
 th2.start();  
 th2.join(10);** }  
}

**Example Scenario – 2**

class MyThread extends Thread {  
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 System.*out*.println("Child created thread-----" + i);  
 try {  
 Thread.*sleep*(300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

**OUTPUT**

Child created thread-----0

Child created thread-----1

......

Child created thread-----4

Main thread------>0

........

Main thread------>4

class JoinDemo {  
 public static void main(String a[]) throws Exception {  
 MyThread t = new MyThread();  
 t.start();  
 t.join(); *// or t.join(3000);* for (int i = 0; i < 5; i++) {  
 System.*out*.println("Main thread------>" + i);  
 }  
 }  
}

In the above case, the child thread will run first and after that main thread will print the values.

**Example Scenario – 3 : There are two players and they are playing alternately.**

public class **Game** {  
 volatile boolean status = false;  
  
 public synchronized void play(String name) {  
 try {  
 System.*out*.println("---------------------START--------------------");  
 System.*out*.println(name + " is playing the game");  
 Thread.*sleep*(2000);  
 System.*out*.println(name + " completed the game...");  
 System.*out*.println("---------------------OVER---------------------\n");  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
}

**OUTPUT**

---------------------START--------------------

Player 1 is playing the game

Player 1 completed the game...

---------------------OVER---------------------

---------------------START--------------------

Player 2 is playing the game

Player 2 completed the game...

---------------------OVER---------------------

---------------------START--------------------

Player 1 is playing the game

Player 1 completed the game...

---------------------OVER---------------------

…….. …… ……

public class **Player1** extends Thread {  
 private Game game;  
  
 public Player1(Game game) {  
 this.game = game;  
 }  
  
 public void run() {  
 while (true) {  
 **game.play("Player 1");**  
 try {  
 **join(1000);**  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

public class **Player2** extends Thread {  
 private Game game;  
  
 public Player2(Game game) {  
 this.game = game;  
 }  
  
 public void run() {  
 while (true) {  
 **game.play("Player 2");**  
 try {  
 **join(1000);**  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

public class **TestGameJoin** {  
 public static void main(String[] args) {  
 Game game = new Game();  
 new Player1(game).start();  
 new Player2(game).start();  
 }  
  
}

**Use of Thread.yield**

**This static method is essentially used to notify the system that the current thread is willing to "give up the CPU" for a while.** We should try to avoid using Thread.yield() method.

* **Invocation of method yield() on java.lang.Thread is usually used to masquerade synchronization problems and should be avoided.**
* **The method Thread.yield() should not be used because its behavior is not consistent across all platforms.**

**What is the difference between sleep() and yield()**

**Sleep is used to delay the execution for a period of** time, and sleeping thread is guaranteed for sleep at least the specified time. But it not guaranteed that newly weak up thread will actually returning to running.

Yield is used to get the running thread into out of runnable state with the same priority.

A thread might yield, then immediately re-enter to the running state.

May be you can confused on that, read about thread priority and it'll give an idea what exactly happen.

But with the yield method the thread never goes into the sleeping state it only allows other threads of the same priority to execute.

**isAlive()**

**The isAlive method is generally used to test if a thread has started.** Right after creating a Thread object, the thread object is in the "**New Thread**" state. When a thread is a New Thread, it is merely an "empty" Thread object; no system resources have been allocated for it yet. When a thread is in this state, you can only start the thread. Calling any method besides start when a thread is in this state makes no sense and causes an IllegalThreadStateException. (Infact, the runtime system throws an IllegalThreadStateException any time a method is called on a thread and that thread's state does not allow for that method call.)

**DEFINITION OF THE isAlive() METHOD:**

The isAlive method returns true if the thread has been started and not stopped. If the isAlive method returns false, you know that the thread either is a New Thread (start has not been called yet) or is Dead. If the isAlive method returns true, you know that the thread is either Runnable or Not Runnable. You cannot differentiate between a New Thread or a Dead thread. Nor can you differentiate between a Runnable thread and a Not Runnable thread.

**There is a difference between join() and sleep()**

**join( )** method of java waits till the thread on which it is called terminates. The isAlive( ) method of java returns true only if the thread is still running. **join()** will wait until the timeout expires or the thread finishes. **sleep()** will just wait for the specified amount of time unless interrupted. So it is perfectly possible for join() to return much faster than the specified time. **join()** waits for something meaningful while **sleep()** just sits there doing nothing.

**\*\* Situation \*\***

Two threads, one by one thread, it means one operation will be performed by only one thread. Once that thread completes that operation, another thread has to start the operation. In this case, you have to use "Thread.join()" method. But there is a trick in this case, see below.

**th1.start();**

**incorrect Implementation 🡺 th2.start();**

**th1.join();**

**th2.join();**

th1.start();

th1.join(); 🡸 **correct implementation**

th2.start();

th2.join();

**Join() in Thread**

**The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing**,

**t.join(); causes the current thread to pause execution until t's thread terminates.**

**When to use join method in Java**

Important point on Thread.join method

Now we know How to use join method in Java, it’s time to see some important points about Thread.join() method.

1. Join is a final method in java.lang.Thread class and you cannot override it.

2) Join method throw IntrupptedException if another thread interrupted waiting thread as a result of join() call.

3) Join is also an overloaded method in Java, three version of join() available, check javadoc for details.

Wait() : If a thread calls wait method on an object (obliviously that thread must already have acquired the lock of the object it is calling the wait method on, otherwise a runtime exception is thrown), that thread releases the object and goes to the waiting state. So in all the wait call forces the thread to release the object it is calling wait on.

Join() : If a thread (name it T1) calls the join method on another Thread (remember the wait method is in the Object class and the join method is specific to the Thread class and is present in the Thread class) named T2, then T1 waits for T2 to complete its execution before it continues from that point. Now think what will happen. Will thread T1 leave the locks on the object it has acquired lock on or not ? If we consider the fact that the join method internally calls the wait method the answer that pops up immediately is that “obviously it should leave the lock“, which is WRONG !!!!

**Join()/wait() method in Java Threads**

The method **join** (of class Thread) wait for a thread to die.

The methods **wait, notify, notifyAll** are not related to the end of execution of one thread.

**The method wait causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.**

**Methods notify and notifyAll are used to awake a sleeping thread.** A common use of wait with notify is accessing a shared resource.

**wait(): Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object**.

**notify(): Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened**.

**notifyAll():Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.**

**The join() method waits for a thread to die. While wait() and notify() are used for inter thread communication**.

Example on join() method.

**import** java.util.concurrent.TimeUnit;  
**public class** TaskThread1 **implements** Runnable {  
 **private int sleepTime**;  
 **private** String **threadName**;  
  
 **public** TaskThread1(String threadName, **int** sleepTime) {  
 **this**.**sleepTime** = sleepTime;  
 **this**.**threadName** = threadName;  
 }  
  
 @Override  
 **public void** run() {  
 Thread.*currentThread*().setName(**threadName**);  
 **try** {  
 **for** (**int** i = 0; i < 5; i++) {  
 String thName = Thread.*currentThread*().getName();  
 System.***out***.println(thName + **" running ..."**+i);  
 TimeUnit.***SECONDS***.sleep(**sleepTime**);  
 }  
 } **catch** (Exception ex) {  
 ex.printStackTrace();  
 }  
 }  
}

Test class is given below.  
**public class** TestTaskThread1 {  
 **public static void** main(String[] args) **throws** Exception {  
 Thread th1 = **new** Thread(**new** TaskThread1(**"Alice"**,1));  
 Thread th2 = **new** Thread(**new** TaskThread1(**"Bob"**,2));  
 **th1.start();  
 th2.start();** System.***out***.println(**"All threads started ..."**);  
 **th1.join();  
 th2.join();** System.***out***.println(**"All operations completed ..."**);  
 }  
}

In the above case, join() method has no impact, simply we are using.

**If we run the above code, see below the output.**

All threads started ...

Alice running ...0

Bob running ...0

Alice running ...1

Bob running ...1

Alice running ...2

Alice running ...3

Bob running ...2

**The following statement System.*out*.println("All operations completed ..."); gets executed only after the completion of all the threads.**

**Note: It is a good way where you can spawn all the threads and wait for all the threads to finish the tasks.**

Now let us execute the following code.  
**public class** TestTaskThread2 {  
 **public static void** main(String[] args) **throws** Exception {  
 Thread th1 = **new** Thread(**new** TaskThread1(**"Alice"**,1));  
 Thread th2 = **new** Thread(**new** TaskThread1(**"Bob"**,2));  
 th1.start();  
 th1.join();  
 th2.start();  
 th2.join();  
 System.***out***.println(**"All operations completed ..."**);  
 }  
}

It means, it looks like a sequential operations even if you have created two threads. First thread 1 is started and after that we have given join() method, it means second thread will not be started until the first thread finishes the operation.

**Output**

Alice running ...0

Alice running ...1

Alice running ...2

Alice running ...3

Alice running ...4

Bob running ...0

Bob running ...1

Bob running ...2

**Join() method in Java Threads**

**Join()** method allows other thread to wait until the already running thread completes its operation or current thread dies. There are two threads, you will different types of behaviors depending upon the usage of join() method.

TaskThread1 is a thread.  
**import** java.util.concurrent.TimeUnit;  
**public class** TaskThread1 **implements** Runnable {  
 **private int sleepTime**;  
 **private** String **threadName**;  
  
 **public** TaskThread1(String threadName, **int** sleepTime) {  
 **this**.**sleepTime** = sleepTime;  
 **this**.**threadName** = threadName;  
 }  
  
 @Override  
 **public void** run() {  
 Thread.*currentThread*().setName(**threadName**);  
 **try** {  
 **for** (**int** i = 0; i < 5; i++) {  
 String thName = Thread.*currentThread*().getName();  
 System.***out***.println(thName + **" running ..."**+i);  
 TimeUnit.***SECONDS***.sleep(**sleepTime**);  
 }  
 } **catch** (Exception ex) {  
 ex.printStackTrace();  
 }  
 }  
}  
**public class** TestTaskThread1 { 🡸 Test class is given below.  
 **public static void** main(String[] args) **throws** Exception {  
 Thread th1 = **new** Thread(**new** TaskThread1(**"Alice"**,1));  
 Thread th2 = **new** Thread(**new** TaskThread1(**"Bob"**,2));  
 **th1.start();  
 th2.start();** System.***out***.println(**"All threads started ..."**);  
 **th1.join();  
 th2.join();** System.***out***.println(**"All operations completed ..."**);  
 }  
}

All threads started ...

Alice running ...0

Bob running ...0

Alice running ...1

Bob running ...1

Alice running ...2

Alice running ...3

Bob running ...2

Alice running ...4

Bob running ...3

Bob running ...4

All operations completed ...

If we run the above code, see below the output.

Here you can see both the threads have been started and running and the following statement

System.***out***.println(**"All operations completed ..."**); gets executed only after the completion of all the threads.

Note: **It is a good way where you can spawn all the threads and wait for all the threads to finish the tasks.**

Alice running ...0

Alice running ...1

Alice running ...2

Alice running ...3

Alice running ...4

Bob running ...0

Bob running ...1

Bob running ...2

Bob running ...3

Bob running ...4

All operations completed ...

Now let us execute the following code.  
**public class** TestTaskThread2 {  
 **public static void** main(String[] args) **throws** Exception {  
 Thread th1 = **new** Thread(**new** TaskThread1(**"Alice"**,1));  
 Thread th2 = **new** Thread(**new** TaskThread1(**"Bob"**,2));  
 th1.start();  
 th1.join();  
 th2.start();  
 th2.join();  
 System.***out***.println(**"All operations completed ..."**);  
 }  
}

Here the output is given below.

It means, it looks like a sequential operations even if you have created two threads. First thread 1 is started and after that we have given join() method, it means second thread will not be started until the first thread finishes the operation.

Question: **Suppose there are two threads, both are running, I want to execute a method after the completion of two threads. How will you do it ?**

There is a another way to perform third task after the completion of two threads.  
**public class** TestTaskThread3 {  
 **public static void** main(String[] args) **throws** Exception {  
 Thread th1 = **new** Thread(**new** TaskThread1(**"Alice"**, 1));  
 Thread th2 = **new** Thread(**new** TaskThread1(**"Bob"**, 2));  
 th1.start();  
 th2.start();  
 **boolean flag = true;  
 while (flag) {  
 if (!th1.isAlive() && !th2.isAlive()) {  
 flag = false;  
 System.*out*.println("All operations completed ...");  
 }  
 }** }  
}

Or

Thread th1 = **new** Thread(**new** TaskThread1(**"Alice"**, 1));  
Thread th2 = **new** Thread(**new** TaskThread1(**"Bob"**, 2));  
th1.start();  
th2.start();  
t1.join();  
t2.join();

**System.*out*.println("All operations completed ...");**

OR

**boolean** flag = **true**;  
**while** (flag) {  
 **if** (t1.isAlive() && t2.isAlive()) {  
 flag = **true**;  
 } **else** flag = **false**;  
}  
System.***out***.println(**"All operations completed ..."**);