**Multi-threading**

**volatile vs AtomicInteger (**<https://www.youtube.com/watch?v=WH5UvQJizH0&ab_channel=DefogTech>)

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

**Q. Difference b/w Wait() and Sleep()**

|  |  |
| --- | --- |
| ***Wait()*** | ***Sleep()*** |
| The Wait() method is related to the Object class. | The Sleep () method is related to the Thread class.  The Sleep () method does not release the lock on the object during Synchronization. |
| It is not a static method. | It is a static method. |
| At the time of the Synchronization, the Wait() method releases obj. | At the time of the Synchronization, the Sleep() method doesn't release the obj, i.e., lock. |
| We can call the Wait () method only from the Synchronized context. | We can call the Sleep () method from outside the Synchronized context. |
| The Sleep() method has two overloaded methods, which are as follows:   * sleep(long milliseconds, int nanoseconds) * sleep(long milliseconds) | The Sleep() method has three overloaded methods, which are as follows:   * Wait() * wait(long timeout, int nanoseconds) * wait(long timeout) |
| The constructor of the Wait() method is defined in the following way: public final void Wait(long timeout) | The constructor of the Sleep () method in the following way: public static void Sleep (long millis) throws Interrupted\_Execption |

**Q. What is Race Condition in Multi-threading?**

A race condition occurs **when two or more threads can access shared data and they try to** change it at the same time.

Race conditions are most commonly associated with computer science and programming. They occur when two computer program processes, or threads, attempt to access the same resource at the same time and cause problems in the system. Race conditions are considered a common issue for multithreaded applications.

**Q. What is Slipped Condition in Multi-threading?**

**Slipped Condition** is a special type of [race condition](https://www.geeksforgeeks.org/operating-system-process-synchronization/) that can occur in a multithreaded application. In this, a thread is suspended **after reading a condition** and **before performing the activities** related to it. It rarely occurs, however, one must look for it if the outcome is not as expected.

## **Q. Difference between synchronization and volatile keyword**

Volatile keyword is not a substitute of a synchronized keyword, but it can be used as an alternative in certain cases. There are the following differences are as follows:

|  |  |
| --- | --- |
| **Volatile Keyword** | **Synchronization Keyword** |
| Volatile keyword is a field modifier. | Synchronized keyword modifies code blocks and methods. |
| The thread cannot be blocked for waiting in case of volatile. | Threads can be blocked for waiting in case of synchronized. |
| It improves thread performance. | Synchronized methods degrade the thread performance. |
| It synchronizes the value of one variable at a time between thread memory and main memory. | It synchronizes the value of all variables between thread memory and main memory. |
| Volatile fields are not subject to compiler optimization. | Synchronize is subject to compiler optimization. |

**Q: The Major Difference between User and Daemon Threads:**

|  |  |
| --- | --- |
| User Thread | Daemon Thread |
| User threads are foreground threads. | Daemon threads are background threads. |
| User threads are high priority threads. | Daemon threads are low priority threads. |
| Its life independent. | Its life depends on user threads. |
| JVM wait until user threads to finish their work. It never exit until all user threads finish their work. | The JVM will’t wait for daemon threads to finish their work. The JVM will exit as soon as all user threads finish their work. |
| JVM will not force to user threads for terminating, so JVM will wait for user threads to terminate themselves. | If all user threads have finished their work JVM will force the daemon threads to terminate |
| User threads are created by the application. | Mostly Daemon threads created by the JVM. |
| Mainly user threads are designed to do some specific task. | Daemon threads are design as to support the user threads. |

Java Thread Pool

**Java Thread pool** represents a group of worker threads that are waiting for the job and reused many times.

In the case of a thread pool, a group of fixed-size threads is created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, the thread is contained in the thread pool again.

Thread Pool Methods

**newFixedThreadPool(int s):** The method creates a thread pool of the fixed size s.

**newCachedThreadPool():** The method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.

**Ques: Java join() method**

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution.

Ques: Difference between atomic and volatile?

Note: The yield() method is used to make the currently running thread pause and allow other threads of the same or higher priority to execute.

**Ques: How can we create a daemon thread in Java?**

**Answer:** A daemon thread in Java can be created by calling the setDaemon(true) method on a thread object before it is started.

Ques: Difference callable and Runnable interface?

Answer: The main difference between the Callable and Runnable interfaces in Java is that Callable can return a value, while Runnable cannot:

* **Callable**

Returns the result of executing the task to the caller. It can also throw a checked exception. Callable instances can only be executed via ExecutorService.

* **Runnable**

Does not return a result and cannot throw a checked exception. Runnable instances can be run by Thread class as well as ExecutorService.

Ques: Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

public class TestThreadTwice1 extends Thread{

public void run(){

System.out.println("running...");

}

public static void main(String args[]){

TestThreadTwice1 t1=new TestThreadTwice1();

t1.start();

t1.start();

}

}

**Ques: What if we call Java run() method directly instead start() method?**

Each thread starts in a separate call stack.

Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

class TestCallRun1 extends Thread{

public void run(){

System.out.println("running...");

}

public static void main(String args[]){

TestCallRun1 t1=new TestCallRun1();

t1.run();//fine, but does not start a separate call stack

}

}

**Java Shutdown Hook**

[**https://www.javatpoint.com/ShutdownHook-thread**](https://www.javatpoint.com/ShutdownHook-thread)

A special construct that facilitates the developers to add some code that has to be run when the Java Virtual Machine (JVM) is shutting down is known as the **Java shutdown hook**. The Java shutdown hook comes in very handy in the cases where one needs to perform some special cleanup work when the JVM is shutting down. Note that handling an operation such as invoking a special method before the JVM terminates does not work using a general construct when the JVM is shutting down due to some external factors. For example, whenever a kill request is generated by the operating system or due to resource is not allocated because of the lack of free memory, then in such a case, it is not possible to invoke the procedure. The shutdown hook solves this problem comfortably by providing an arbitrary block of code.

When does the JVM shut down?

The JVM shuts down when:

* user presses ctrl+c on the command prompt
* System.exit(int) method is invoked
* user logoff
* user shutdown etc

The addShutdownHook() method of the Runtime class is used to register the thread with the Virtual Machine.

**Syntax:**

1. **public** **void** addShutdownHook(Thread hook){}

The object of the Runtime class can be obtained by calling the static factory method getRuntime(). For example:

Runtime r = Runtime.getRuntime();

The removeShutdownHook() method of the Runtime class is invoked to remove the registration of the already registered shutdown hooks.

public **boolean** removeShutdownHook(Thread hook){ }

class MyThread extends Thread{

public void run(){

System.out.println("shut down hook task completed..");

}

}

public class TestShutdown1{

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

Runtime r=Runtime.getRuntime();

r.addShutdownHook(new MyThread());

System.out.println("Now main sleeping... press ctrl+c to exit");

try{Thread.sleep(3000);}catch (Exception e) {}

}

}