**https://www.javatpoint.com/thread-scheduler-in-java**

**Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared 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.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the 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 an exception occurs 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 in two ways:

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

### 1) Process-based Multitasking (Multiprocessing)

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading [registers](https://www.javatpoint.com/register-memory), memory maps, updating lists, etc.

### 2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* A thread is lightweight.
* Cost of communication between the thread is low.

Círculo

Descripción generada automáticamente con confianza baja

As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the [OS](https://www.javatpoint.com/os-tutorial), and one process can have multiple threads.

## **Java Thread class**

Java provides **Thread class** to achieve thread programming. Thread class provides [constructors](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

# **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

Diagrama

Descripción generada automáticamente

# **Java Threads | How to create a thread in Java**

There are two ways to create a thread:

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

# **Thread Scheduler in Java**

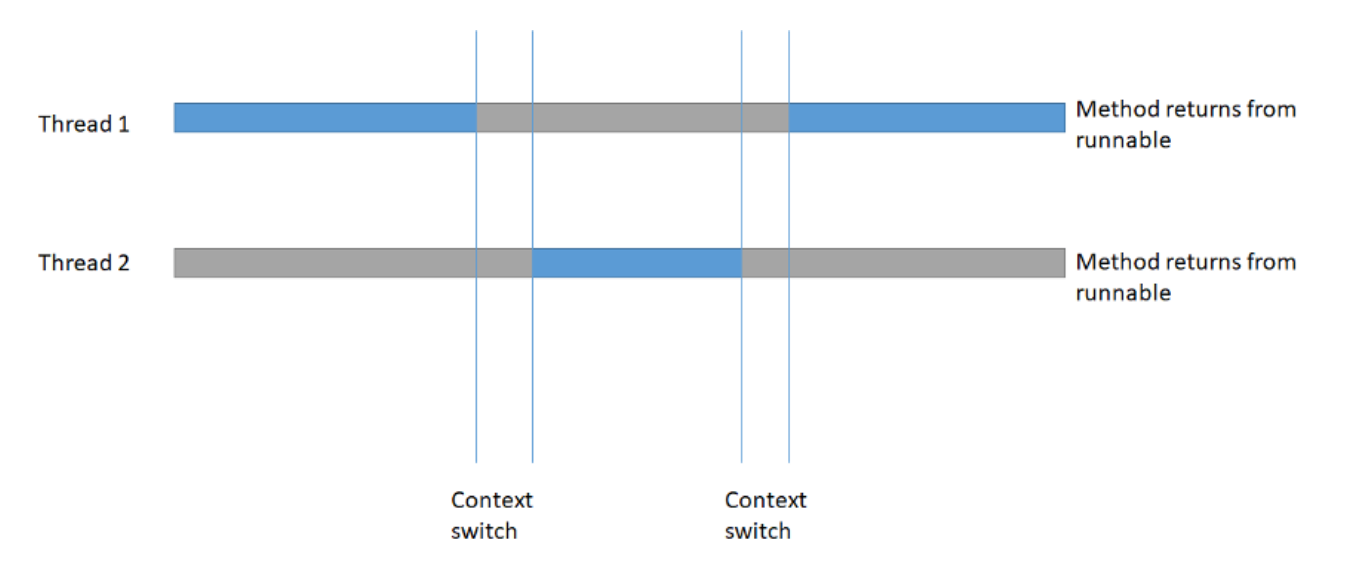
Interfaz de usuario gráfica, Diagrama

Descripción generada automáticamente

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

**Priority:** Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

**Time of Arrival:** Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.



Diagrama

Descripción generada automáticamente con confianza media

# **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.

### Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

### Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

### Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

### Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

### Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

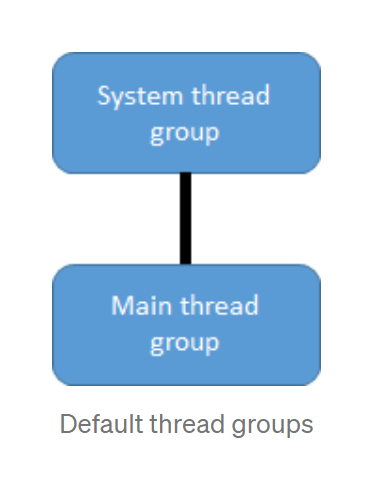
<https://medium.com/@brendoncheung/java-multi-threading-series-part-1-84bbd1227919>

THE JVM starts the main thread which executes the main() method.

Thread Groups.

Here are the key points:

1. All thread group exists in a hierarchical tree.
2. The root of the tree is known as the **root thread group** or **system thread group.**
3. The system thread group contains daemon threads that act as supporters throughout your application. It contains threads like garbage collector, signal dispatcher and more.
4. Every java program by default contains two thread groups, the system thread group and the main thread group



# Thread execution

Thread execution comes down to the start() and run() method.

## start() method

This method will be executed by the caller thread. As soon as this method returns. The thread will be in a runnable state.

## **run()** method

After the thread as entered into runnable state. It is up the the thread scheduler to decide when is the appropriate time to execute the code inside the run() method. You never call run() yourself because if you do, you are just executing code on the caller thread.

Interfaz de usuario gráfica, Texto, Sitio web

Descripción generada automáticamente

# Race condition

Race condition is a problem that naturally occurs when you introduce multi-threading capabilities into your code. If race condition left untreated, it can generate different results, or even unexpected results, that are dependent on executing order.

Simply put, a race condition will happened when you have resource shared between two threads, which give rise to data inconsistency problems.

Here are some key points:

1. if two or more threads only reads from a resource, then it is safe.
2. If two or more threads read AND write to a resource, then data inconsistency problem will occur
3. By resource, it means object’s instance variable.

# Atomic operation

Atomic operations mean code cannot be interrupted during execution, once it starts, it will end without threads interfering and you won’t find it being in a intermediate state.

In Java, reading and writing to a primitive type is considered to be a atomic operation (unless it is long or double).

An example of an non-atomic operation will be:

i++

Because this operation reads and writes to a variable, all under one operation. This means that a thread could interfere and give rise to data inconsistency problem.

ATOMIC - SYNCHRONIZED

Looking back at our TicketManager class, if we can make purchaseTicket method atomic, where no threads can interfere while purchaseTicket is executing, the data inconsistency problem will be a thing of the pass.

We can force a method to become atomic by using the synchronized keyword.

One downfall in this approach is speed, we are synchronizing the entire scope of the method. This will slow down your program because there are codes in the method that can run concurrently without causing data inconsistency problem. Therefore, they don’t need to be synchronized.

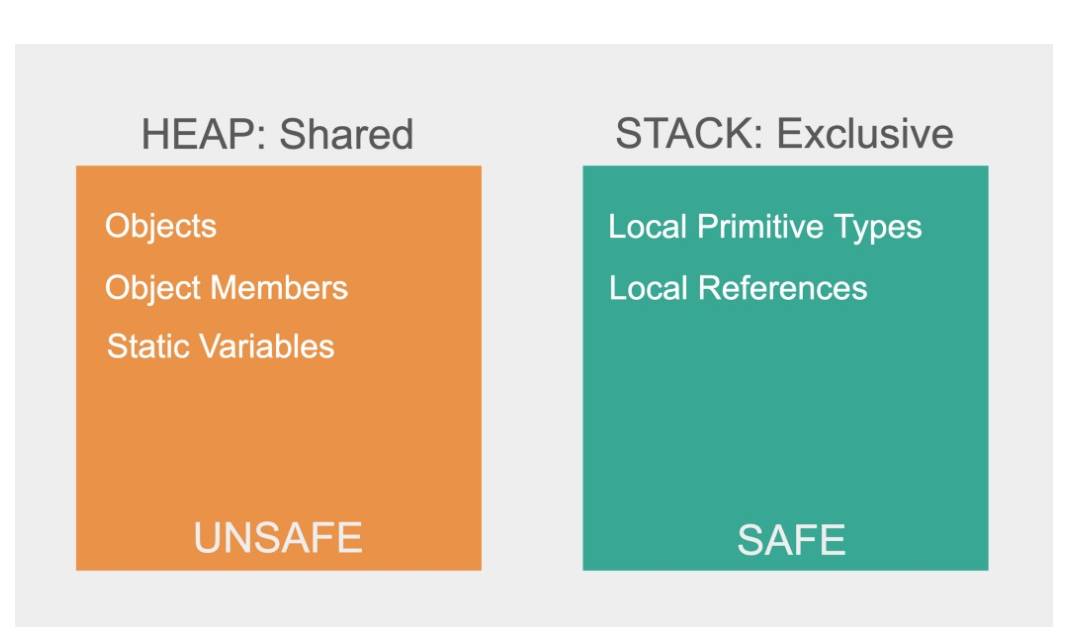
**THREAD SAFETY**

A code that is safe to call by multiple threads simultaneously is called **thread-safe.** If a piece of code is thread-safe, then it contains no **race conditions**. **Race conditions** only occur when multiple threads update shared resources. Therefore it is important to know what resources Java threads share when executing. But before we dive into the resources, we need to understand what are **race conditions and critical area**means.

# **Which Resources Are Thread Safe**

Now we know what are **race conditions** and **critical area** means. We can switch to the resources and try to understand which resources are safe and which are not.

Java Memory model includes [Heap and Stack](https://javarevisited.blogspot.com/2013/01/difference-between-stack-and-heap-java.html" \t "_blank). Data that is storing in the Heap can be shared between threads, which means it is **not a thread-safe data**. Data stored in a Stack memory is not shared between the threads, which means it is a **thread-safe data**.



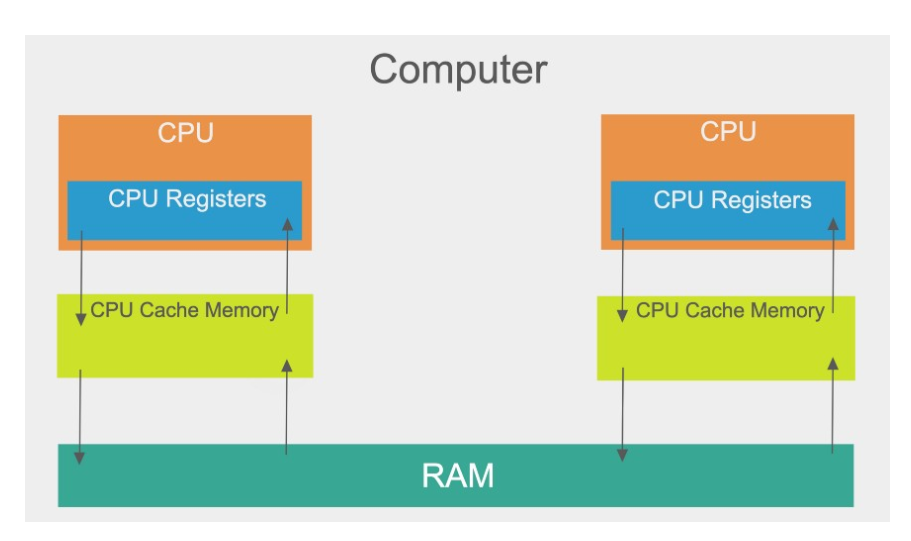
**JAVA VOLATILE**

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". That means, that every read of a volatile variable will be read from the computer's main memory (RAM), and not from the CPU cache and that each writes to a volatile variable will be written to the RAM, and not just to the CPU cache.

# Variable Visibility Problems

The Java volatile keyword guarantees visibility of changes to variables across threads.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from RAM memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs.



# The Java volatile Visibility Guarantee

The Java[volatile keyword](http://javarevisited.blogspot.sg/2011/06/volatile-keyword-java-example-tutorial.html#axzz5DmwFLA1K) is intended to address variable visibility problems. By declaring the counter variable volatile all writes to the counter variable will be written back to the RAM immediately. Also, all reads of the counter variable will be read directly from the RAM.

Here is how the volatile declaration of the counter variable looks:

public class SharedObject { public **volatile** int counter = 0;}

# Volatile Visibility Guarantee

The visibility guarantee of Java volatile goes beyond the volatile variable itself. The visibility guarantee is as follows:

* If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable will also be visible to Thread B after it has read the volatile variable.
* If Thread A reads a volatile variable, then all all variables visible to Thread A when reading the volatile variable will also be re-read from the RAM.

public class MyDate {  
 private int years;  
 private int months  
 private volatile int days;  
 public void update(int years, int months, int days){  
 this.years = years;  
 this.months = months;  
 this.days = days;  
 }  
}

The udpate() method writes three variables, of which only days is volatile.

The full volatile visibility guarantee means, that when a value is written to days, then all variables visible to the [thread](https://www.java67.com/2012/08/5-thread-interview-questions-answers-in.html" \t "_blank)are also written to the RAM. That means, that when a value is written to days, the values of years and months are also written to the RAM.

# volatile is Not Always Enough

As soon as a thread needs to first read the value of a volatile variable, and based on that value generate a new value for the shared volatile variable, a volatile variable is no longer enough to guarantee correct visibility. The short time gap in between the reading of the volatile variable and the writing of its new value creates **a [race condition](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html" \l "axzz59AbkWuk9" \t "_blank)** where multiple threads might read the same value of the volatile variable, generate a new value for the variable, and when writing the value back to the RAM - overwrite each other's values.

# When is volatile Enough?

If two threads are both reading and writing to a shared variable, then using the volatile keyword for that **is not enough**. You need to use a **[synchronized](https://medium.com/javarevisited/java-concurrency-synchronized-7828bf5f06cb?source=your_stories_page-------------------------------------)** in that case to guarantee that the reading and writing of the variable are atomic. Reading or writing a volatile variable does not block threads reading or writing. For this to happen you must use the synchronized keyword around critical sections.

As an alternative to a synchronized block you could also use one of the many atomic data types found in the **java.util.concurrent package**. For instance, the **AtomicLong** or **AtomicReference** or one of the others.

In case only one thread reads and writes the value of a volatile variable and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable. Without making the variable volatile, this would not be guaranteed.

The volatile keyword is guaranteed to work on 32 bit and 64 variables.

**ATOMIC REFERENCE**

# Overview

A reference type is a data type that represents an instance of a Java class, i.e. it is a type other than one of the Java’s primitive types. For instance:

Long myLong = new Long(5);

In the above snippet the variable myLong represents a reference type. When we create the above object myLong, Java allocates appropriate memory for the object to be stored. The variable myLong points to this address in the memory, which stores the object. The address of the object in the memory is called the reference.

Long myLong = new Long(5);  
Long otherLong = myLong;

The two variables myLong and otherLong both point to the same memory location where the Long object is stored or we can say both variables are assigned the same reference (address) to the object but not the object itself.

Given the above context, it becomes easier to understand the AtomicReference class. According to the official documentation, AtomicReference allows us to atomically update the reference to an object. Before we further delve into the topic, we’ll first clarify the distinction between AtomicReference and assignment of references atomically in Java.

https://medium.com/double-pointer/guide-to-atomicreference-in-java-132eaa1c9670