**Threads**

Java Threads allow class objects, processes and methods to be run simultaneously alongside of other objects and processes and “share” processor and memory resources.

Thread is basically, a lightweight sub-process, a smallest unit of processing.

In this way, Java programs can do multitask and do several things at once. The processor will rotate its attention between Threads based on the **priority** value given to each Thread.

When multiple Threads share a single CPU resource, rapidly alternating between CPU cycles, it is known as “**Time Slicing**”.

The amount of time allocated to each Thread is referred to as a “**quantum**”.

Multithreading, as its name implies, uses multiple Threads simultaneously.

**Advantages**:

* It does not block the user because threads are independent and you can perform multiple operations at same time.
* Threads are independent so it doesn’t affect other threads if exception occur in a single thread.
* Threads share the same address space.
* Cost of communication between the thread is low.

**Multitasking and Multithreading:**

* Multitasking refers to a computer’s ability to perform multiple jobs concurrently.
* more than one program is running concurrently, e.g., Unix.
* A thread is a sequence of execution within a program.
* Multithreading refers to multiple threads of control within single program.

**Difference between Concurrency and Parallelism:**

Concurrency means that an application is making progress on more than one task at the same time (concurrently). Well, if computer has only one CPU the application may not make progress on more than one task at exactly the same time, but more than one task being processed at a time inside the application. It does not completely finish one task before it begins the next.

**In short:** Multiple tasks makes progress at the same time.

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time.

**In short:** Each task is broken into subtasks which can be processed in parallel.

**Threads:**

Java Threads are objects like any other Java objects. Threads are instances of class java.lang.Thread, or instances of subclasses of this class. In addition to being objects, java thread also execute code.

**Creating and Starting Threads:**

Creating a thread in Java is done like this:

Thread thread = new Thread();

To start the Java thread, you will call its start() method, like this:

thread.start();

There are 2 ways to specify what code the thread should execute. The first is to create a subclass of Thread and override the run() method. The second method is to pass an object that implements Runnable (java.lang.Runnable) to the Thread constructor.

**Thread Subclass:**

The first way to specify what code a thread is to run, is to create a subclass of Thread and override the run() method. The run() method is what is executed by the thread after you call start(). Here is an example of creating a Java Thread subclass.

|  |
| --- |
| public class MyThread extends Thread {  public void run(){  System.out.println("MyThread running");  }  } |

To create and start the above thread you can do like this:

|  |
| --- |
| MyThread myThread = new MyThread();  myTread.start(); |

The start() call will return as soon as the thread is started. It will not wait until the run() method is done. The run() method will execute as if executed by a different CPU.

You can also create an anonymous subclass of Thread like this:

**Runnable Interface Implementation:**

The second way to specify what code a thread should run is by creating a class that implements java.lang.Runnable. The Runnable object can be executed by a Thread.

Here is a Java Runnable Example:

|  |
| --- |
| public class MyRunnable implements Runnable {  public void run(){  System.out.println("MyRunnable running");  }  } |

To have the run() method executed by a thread, pass an instance of MyRunnable to Thread in its constructor. Here is how that is done:

|  |
| --- |
| Thread thread = new Thread(new MyRunnable());  thread.start(); |

When the thread is started it will call the run() method of the MyRunnable instance instead of executing its own run() method.

You can also create an anonymous implementation of Runnable, like this:

|  |
| --- |
| Runnable myRunnable = new Runnable(){  public void run () {  System.out.println("Runnable running");  }  }  Thread thread = new Thread(myRunnable);  thread.start(); |

**Subclass or Runnable?**

There are no rules about which of the two methods that is the best. Both methods work.

Personally though, I prefer implementing Runnable, and handing an instance of the implementation to a Thread instance. When having the Runnable executed by a **Thread Pool** it is easy to queue up the Runnable instances until a thread from the pool is idle. This is little harder to do with Thread subclasses.

**Java Thread Example:**

Note that even if the threads are started in sequence (1,2,3 etc.) they may not execute sequentially, meaning thread 1 may not be the first thread to write its name to System.Out.Println(). This is because the threads are in principle executing in parallel and not sequentially. The JVM and/or operating System determines the order in which the threads are executed. This order does not have to be same order in which they started.

**Critical Sections:**

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance, the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.

In fact, problems only arise if one or more threads write to these resources. It is safe to let multiple threads to read the same resources, as long as the resources do not change.

Here is a critical section Java code that may fail if executed by multiple threads simultaneously:

|  |
| --- |
| public class Counter {  protected long count = 0;  public void add(long value){  this.count = this.count + value;  }  } |

Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class. There is no way to know when the operating system switches between the two threads. The code in the add() method is not executed as a single atomic instruction by the JVM. Rather it is executed as a set of smaller instructions similar to this:

1. Read this.count from memory into register.
2. Add value to register.
3. Write register to memory.

Observe what happens with the following mixed execution of Threads A and B:

|  |
| --- |
| this.count = 0;  A: Reads this.count into a register (0)  B: Reads this.count into a register (0)  B: Adds value 2 to register  B: Writes register value (2) back to memory. this.count now equals 2  A: Adds value 3 to register  A: Writes register value (3) back to memory. this.count now equals 3 |

The two threads wanted to add the values 2 and 3 to the counter. Thus the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, the results ends up being different.

In the execution sequence example listed above, both threads read the value 0 from memory. Then they add their individual values 2, 3 to the value, and write the result back to memory. Instead of 5, the value left in this.count will be the value written by the last thread to write its value. In the above case it is thread A, but it could as well have been thread B.

**Race Conditions in Critical Sections:**

The code in the add() method in the example earlier contains a critical section. When multiple threads execute this critical section, race condition occur.

**In Short:** The situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section.

**Preventing Race Conditions:**

To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Race conditions can be avoided by proper thread synchronization in critical sections. Thread synchronization can be achieved using a **synchronized block of Java code.** Thread synchronization can also be achieved using other synchronization constructs like **lock** or atomic variables like **java.util.concurrent.atomic.AtomicInteger.**

**Thread Safety:** Code that is safe to call by multiple threads simultaneously is called thread safe.

**Java Synchronized Blocks:**

A Java synchronized block marks a method or block of code as synchronized. Java synchronized blocks can be used to avoid **race conditions**.

**The Java Synchronized keyword:**

Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

They synchronized keyword can be used to mark four different types of blocks:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods
4. Code blocks inside static methods

**Synchronized Instance methods:**

Here is synchronized instance method:

|  |
| --- |
| public **synchronized** void add(int value){  this.count += value;  } |

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object. Only one thread can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

**Synchronized Static Methods:**

Here is a Java synchronized static method example:

|  |
| --- |
| public **static synchronized** void add(int value){  count += value;  } |

Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java JVM per class, only one thread can execute inside a static synchronized method in same class.

**Synchronized Blocks in Instance Methods:**

You don’t have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

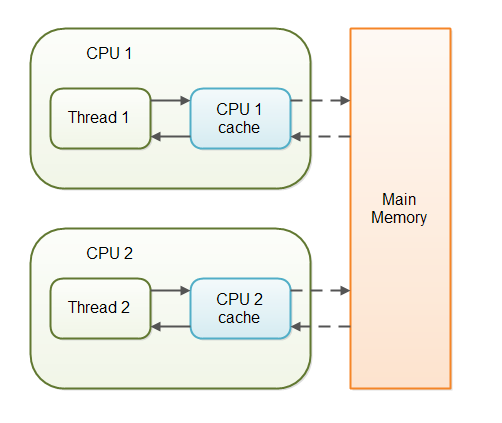
|  |
| --- |
| public void add(int value){  **synchronized(this){**  this.count += value;  **}**  } |

**Java Volatile Keyword:**

The Java volatile keyword is used to mark a Java variable as “being stored in main memory”. More precisely that means, that every read of a volatile variable will be read from the computer’s main memory, and not just from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

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 main 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. This is illustrated here:



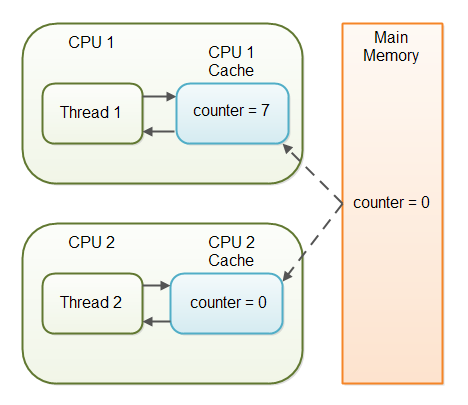
With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems which I will explain in the following sections.

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

|  |
| --- |
| public class SharedObject {  public int counter = 0;  } |

Imagine too, that only Thread 1 increments the counter variable, but both Thread 1 and Thread 2 may read the counter variable from time to time.

If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory. Here is how the volatile declaration of the counter variable looks:

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

**Deadlock:**

Deadlock describes the situation where two or more threads are blocked forever and waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order.

Example for DeadLock:

**DeadLock Solution Example:**

Change the order of the lock and run the above same program to see if still both the threads waits for each other.

Example:

**Java Interthread Communication:**

**Interrupts:**

**Producer and Consumer Example:**

[**http://www.javatpoint.com/multithreading-in-java**](http://www.javatpoint.com/multithreading-in-java)

[**http://www.tutorialspoint.com/java/java\_multithreading.htm**](http://www.tutorialspoint.com/java/java_multithreading.htm)

**Difference between preemptive scheduling and time slicing?**

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool for ready tasks.

**Java Thread Pool:**

**JTP** represents a group of worker threads that are waiting for the job and reuse many times.

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

ExecutorService executor = Executors.newFixedThreadPool(10);

executor.execute(Thread Object);

**ThreadGroup in Java:**

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Example:

|  |
| --- |
| ThreadGroup tg1 = **new** ThreadGroup("Parent ThreadGroup");        Thread t1 = **new** Thread(tg1, runnable,"one");      t1.start();      Thread t2 = **new** Thread(tg1, runnable,"two");      t2.start();      Thread t3 = **new** Thread(tg1, runnable,"three");             t3.start();  Thread.currentThread().getThreadGroup().interrupt(); |

**Shutdown Hook:**

The shutdown hook can be used to perform clean up resources 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.

**addShutdownHook(Thread hook):**

This method of Runtime class is used to register the thread with the Virtual machine.

Runtime r = Runtime.getRuntime();

r.addShutdownHook(new Thread());