Multithreading and concurrency

=====================================================================

|  |  |
| --- | --- |
| **Java Concurrency** | |
| **1** | [**Java Concurrency / Multithreading Tutorial**](http://tutorials.jenkov.com/java-concurrency/index.html)   * The single CPU was shared between the programs. The operating system would switch between the programs running, executing each of them for a little while before switching. * mulithreading is even more challenging than multitasking. The threads are executing within the same program and are hence reading and writing the same memory simultanously. * Modern computers, though, come with multi core CPUs, and even with multiple CPUs too. This means that separate threads can be executed by separate cores or CPUs simultanously. * If a thread reads a memory location while another thread writes to it, what value will the first thread end up reading? The old value? The value written by the second thread? Or a value that is a mix between the two? Or, if two threads are writing to the same memory location simultanously, what value will be left when they are done? The value written by the first thread? The value written by the second thread? Or a mix of the two values written? |
|  | C:\Users\Ceic Spiritualr\Desktop\java-concurrency-tutorial-introduction-1.png |
| 2 | [Multithreading Benefits](http://tutorials.jenkov.com/java-concurrency/benefits.html)   * Better resource utilization. * Simpler program design in some situations. * More responsive programs. |
| 3 | [Multithreading Costs](http://tutorials.jenkov.com/java-concurrency/costs.html)  **More complex design**  Code executed by multiple threads accessing shared data need special attention. Thread interaction is far from always simple. Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix. Context Switching OverheadWhen a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch"Increased Resource Consumption A thread needs some resources from the computer in order to run. Besides CPU time a thread needs some memory to keep its local stack. It may also take up some resources inside the operating system needed to manage the thread. |
| 4 | [Concurrency Models](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html)   * [Concurrency Models and Distributed System Similarities](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#concurrency-models-and-distributed-system-similarities) * [Parallel Workers](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#parallel-workers) * C:\Users\Ceic Spiritualr\Desktop\java-concurrency-tutorial-introduction-1.png * [Parallel Workers Advantages](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#parallel-workers-advantages) * [Parallel Workers Disadvantages](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#parallel-workers-disadvantages)   + [Shared State Can Get Complex](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#shared-state-can-get-complex)   C:\Users\Ceic Spiritualr\Desktop\concurrency-models-2.png   * + [Stateless Workers](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#stateless-workers)   + [Job Ordering is Nondeterministic](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#job-ordering-is-nondeterministic) * [Assembly Line](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#assembly-line)   + [Reactive, Event Driven Systems](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#reactive-event-driven-systems)   + [Actors vs. Channels](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#actors-vs-channels) * [Assembly Line Advantages](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#assembly-line-advantages)   + [No Shared State](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#no-shared-state)   + [Stateful Workers](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#stateful-workers)   + [Better Hardware Conformity](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#better-hardware-conformity)   + [Job Ordering is Possible](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#job-ordering-is-possible) * [Assembly Line Disadvantages](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#assembly-line-disadvantages) * [Functional Parallelism](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#functional-parallelism) * [Which Concurrency Model is Best?](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#which-concurrency-model-is-best) |
| 5 | [Concurrency vs. Parallelism](http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html) Concurrency Concurrency means that an application is making progress on more than one task at the same time Parallelism 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. Concurrency vs. Parallelism In Detail Parallelism on the other hand, is related to how an application handles each individual task. An application may process the task serially from start to end, or split the task up into subtasks which can be completed in parallel.  As you can see, an application can be concurrent, but not parallel. This means that it processes more than one task at the same time, but the tasks are not broken down into subtasks.  An application can also be parallel but not concurrent. This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel.  Additionally, an application can be neither concurrent nor parallel. This means that it works on only one task at a time, and the task is never broken down into subtasks for parallel execution. |
| 6 | [Creating and Starting Java Threads](http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html)  MyThread myThread = new MyThread();  myTread.start();  Thread thread = new Thread(new MyRunnable());  thread.start(); |
| 7 | [Race Conditions and Critical Sections](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html)  Race Conditions  -----------------  A race condition is a special condition that may occur inside a critical section. A critical section is a section of code that is executed  by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the  critical section.  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.  In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the  same resources, as long as the resources do not change. |
| 8 | [Thread Safety and Shared Resources](http://tutorials.jenkov.com/java-concurrency/thread-safety.html)  Local Variables  -------------------  Local variables are stored in each thread's own stack. That means that local variables are never shared between threads.  That also means that all local primitive variables are thread safe.  Local Object References  -------------------  Local references to objects are a bit different. The reference itself is not shared. The object referenced however, is not stored in  a each threads's local stack. All objects are stored in the shared heap.  If an object created locally never escapes the method it was created in, it is thread safe. In fact you can also pass it on to other methods  and objects as long as none of these methods or objects make the passed object available to other threads.  expublic void someMethod(){  LocalObject localObject = new LocalObject();  localObject.callMethod();  method2(localObject);  }  public void method2(LocalObject localObject){  localObject.setValue("value");  }  The LocalObject instance in this example is not returned from the method, nor is it passed to any other objects that are accessible from  outside the someMethod() method. Each thread executing the someMethod() method will create its own LocalObject instance and assign it to  the localObject reference. Therefore the use of the LocalObject here is thread safe.  In fact, the whole method someMethod() is thread safe. Even if the LocalObject instance is passed as parameter to other methods in the  same class, or in other classes, the use of it is thread safe.  The only exception is of course, if one of the methods called with the LocalObject as parameter, stores the LocalObject instance in a way  that allows access to it from other threads.  Object Member Variables  ----------------------  Object member variables (fields) are stored on the heap along with the object. Therefore, if two threads call a method on the same object  instance and this method updates object member variables, the method is not thread safe.  Ex-  public class NotThreadSafe{  StringBuilder builder = new StringBuilder();  public add(String text){  this.builder.append(text);  }  }  If two threads call the add() method simultaneously on the same NotThreadSafe instance then it leads to race conditions  The Thread Control Escape Rule  ------------------------------  If a resource is created, used and disposed within  the control of the same thread,  and never escapes the control of this thread,  the use of that resource is thread safe. |
| 9 | [Thread Safety and Immutability](http://tutorials.jenkov.com/java-concurrency/thread-safety-and-immutability.html)  The Reference is not Thread Safe!  We can make sure that objects shared between threads are never updated by any of the threads by making the shared objects immutable, and thereby thread safe.  Ex-  public class ImmutableValue{  private int value = 0;  public ImmutableValue(int value){  this.value = value;  }  public int getValue(){  return this.value;  }  }  public ImmutableValue add(int valueToAdd){  return new ImmutableValue(this.value + valueToAdd);  }  Notice how the add() method returns a new ImmutableValue instance with the result of the add operation,  rather than adding the value to itself.  The Reference is not Thread Safe!  -------------------------  It is important to remember, that even if an object is immutable and thereby thread safe, the reference to this object may not be thread safe  Ex-  public class Calculator{  private ImmutableValue currentValue = null;  public ImmutableValue getValue(){  return currentValue;  } |
| 10 | [Java Memory Model](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html)   * [The Internal Java Memory Model](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#javas-logic-memory-model)   Thread Stack +   * [Hardware Memory Architecture](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#hardware-memory-architecture) * [Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#bridging-the-gap-between-the-java-memory-model-and-the-hardware-memory-architecture)   + [Visibility of Shared Objects](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#visibility-of-shared-objects)   + [Race Conditions](http://tutorials.jenkov.com/java-concurrency/java-memory-model.html#race-conditions)   Java memory model from a logic perspective: |
| 11 | [Java Synchronized Blocks](http://tutorials.jenkov.com/java-concurrency/synchronized.html)   * [The Java synchronized Keyword](http://tutorials.jenkov.com/java-concurrency/synchronized.html#synchronized-keyword)   All synchronized blocks synchronized on the 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.  These blocks are synchronized on different objects. Which type of synchronized block you need depends on the concrete situation.   * [Synchronized Instance Methods](http://tutorials.jenkov.com/java-concurrency/synchronized.html#synchronized-instance-methods)   public **synchronized** void add(int value){  this.count += value;  }  One thread at a time can execute inside a synchronized instance method per instance. One thread per instance.   * [Synchronized Static Methods](http://tutorials.jenkov.com/java-concurrency/synchronized.html#synchronized-static-methods)   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 VM per class, only one thread can execute inside a static synchronized method in the same class. * One thread per class regardless of which static synchronized method it calls. * [Synchronized Blocks in Instance Methods](http://tutorials.jenkov.com/java-concurrency/synchronized.html#synchronized-blocks-instance-methods)   public void add(int value){  **synchronized(this){**  this.count += value;  **}**  }   * A synchronized instance method uses the object it belongs to as monitor object.   Only one thread can execute inside a Java code block synchronized on the same monitor object.  The following two examples are both synchronized on the instance they are called on. They are therefore equivalent with respect to synchronization:    public class MyClass {    public **synchronized** void log1(String msg1, String msg2){  log.writeln(msg1);  log.writeln(msg2);  }    public void log2(String msg1, String msg2){  **synchronized(this){**  log.writeln(msg1);  log.writeln(msg2);  **}**  }  }  Thus only a single thread can execute inside either of the two synchronized blocks in this example.   * [Synchronized Blocks in Static Methods](http://tutorials.jenkov.com/java-concurrency/synchronized.html#synchronized-blocks-static-methods)   These methods are synchronized on the class object of the class the methods belong to:  public class MyClass {  public static synchronized void log1(String msg1, String msg2){  log.writeln(msg1);  log.writeln(msg2);  }    public static void log2(String msg1, String msg2){  synchronized(MyClass.class){  log.writeln(msg1);  log.writeln(msg2);  }  }  }  Only one thread can execute inside any of these two methods at the same time.   * [Java Synchronized Example](http://tutorials.jenkov.com/java-concurrency/synchronized.html#java-synchronized-example) * [Java Concurrency Utilities](http://tutorials.jenkov.com/java-concurrency/synchronized.html#java-concurrency-utilities)   Java synchronized blocks can be used to avoid [race conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html). |
| 12 | [Java Volatile Keyword](http://tutorials.jenkov.com/java-concurrency/volatile.html) The Java volatile Visibility Guarantee  * The Java volatile keyword guarantees visibility of changes to variables across threads. * 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.   C:\Users\Ceic Spiritualr\Desktop\concurrency-models-2.png  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  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 countervariable 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   C:\Users\Ceic Spiritualr\Desktop\concurrency-models-2.png   * 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 * 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.   public class SharedObject {  public **volatile** int counter = 0;  } |
| 13 | [Java ThreadLocal](http://tutorials.jenkov.com/java-concurrency/threadlocal.html)  The ThreadLocal class in Java enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to aThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables. Creating a ThreadLocal As you can see, you instantiate a new ThreadLocal object. This only needs to be done once per thread. Even if different threads execute the same code which accesses a ThreadLococal, each thread will see only its own ThreadLocal instance. Creating a ThreadLocal private ThreadLocal myThreadLocal = new ThreadLocal();  As you can see, you instantiate a new ThreadLocal object. This only needs to be done once per thread. Even if different threads execute the same code which accesses a ThreadLococal, each thread will see only its own ThreadLocal instance. Accessing a ThreadLocal Once a ThreadLocal has been created you can set the value to be stored in it like this:  myThreadLocal.set("A thread local value");  You read the value stored in a ThreadLocal like this:  String threadLocalValue = (String) myThreadLocal.get();  The get() method returns an Object and the set() method takes an Object as parameter. Generic ThreadLocal private ThreadLocal myThreadLocal = new ThreadLocal<String>();  myThreadLocal.set("Hello ThreadLocal");  String threadLocalValue = myThreadLocal.get(); Initial ThreadLocal Value Since values set on a ThreadLocal object only are visible to the thread who set the value, no thread can set an initial value on a ThreadLocal using set() which is visible to all threads.  private ThreadLocal myThreadLocal = new ThreadLocal<String>() {  @Override protected String initialValue() {  return "This is the initial value";  }  }; InheritableThreadLocal The InheritableThreadLocal class is a subclass of ThreadLocal. Instead of each thread having its own value inside a ThreadLocal, the InheritableThreadLocal grants access to values to a thread and all child threads created by that thread. |
| 14 | [Thread Signaling](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html)  The purpose of thread signaling is to enable threads to send signals to each other. Additionally, thread signaling enables threads to wait for signals from other threads. For instance, a thread B might wait for a signal from thread A indicating that data is ready to be processed.   * [Signaling via Shared Objects](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#shared-objects) * [Busy Wait](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#busy-wait) * [wait(), notify() and notifyAll()](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#wait-notify) * [Missed Signals](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#missed-signals) * [Spurious Wakeups](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#spurious-wakeups) * [Multiple Threads Waiting for the Same Signals](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#multiple-threads) * [Don't call wait() on constant String's or global objects](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#strings) |
| 15 | [Deadlock](http://tutorials.jenkov.com/java-concurrency/deadlock.html)   Deadlock can occur when multiple threads need the same locks, at the same time, but obtain them in different order.  The situation is illustrated below:  Thread 1 locks A, waits for B  Thread 2 locks B, waits for A More Complicated Deadlocks Deadlock can also include more than two threads. This makes it harder to detect. Here is an example in which four threads have deadlocked:  Thread 1 locks A, waits for B  Thread 2 locks B, waits for C  Thread 3 locks C, waits for D  Thread 4 locks D, waits for A  Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1. Database Deadlocks A more complicated situation in which deadlocks can occur, is a database transaction. When a record is updated during a transaction, that record is locked for updates from other transactions, until the first transaction completes. Each update request within the same transaction may therefore lock some records in the database.  sFor example  Transaction 1, request 1, locks record 1 for update  Transaction 2, request 1, locks record 2 for update  Transaction 1, request 2, tries to lock record 2 for update.  Transaction 2, request 2, tries to lock record 1 for update.  it is hard to detect or prevent deadlocks in database transactions. |
| 16 | [Deadlock Prevention](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html) Lock Ordering Deadlock occurs when multiple threads need the same locks but obtain them in different order.  If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occu  Thread 1:  lock A  lock B  Thread 2:  wait for A  lock C (when A locked)  Thread 3:  wait for A  wait for B  wait for C Lock Timeout Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up  Thread 1 locks A  Thread 2 locks B  Thread 1 attempts to lock B but is blocked  Thread 2 attempts to lock A but is blocked  Thread 1's lock attempt on B times out  Thread 1 backs up and releases A as well  Thread 1 waits randomly (e.g. 257 millis) before retrying.  Thread 2's lock attempt on A times out  Thread 2 backs up and releases B as well  Thread 2 waits randomly (e.g. 43 millis) before retrying. Deadlock Detection Deadlock detection is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible.  Every time a thread **takes** a lock it is noted in a data structure (map, graph etc.) of threads and locks. Additionally, whenever a thread **requests** a lock this is also noted in this data structure.  When a thread requests a lock but the request is denied, the thread can traverse the lock graph to check for deadlocks.  C:\Users\Ceic Spiritualr\Desktop\deadlock-detection-graph.png  So what do the threads do if a deadlock is detected?  One possible action is to release all locks, backup, wait a random amount of time and then retry.  A better option is to determine or assign a priority of the threads so that only one (or a few) thread backs up. The rest of the threads continue taking the locks they need as if no deadlock had occurred |
| 17 | [Starvation and Fairness](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html)  If a thread is not granted CPU time because other threads grab it all, it is called "starvation". The thread is "starved to death" because other threads are allowed the CPU time instead of it. The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute.   * [Causes of Starvation in Java](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#starvation)   + [Threads with high priority swallow all CPU time from threads with lower priority](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#high-priority)   + [Threads are blocked indefinitely waiting to enter a synchronized block](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#synchronized)   + [Threads waiting on an object (called wait() on it) remain waiting indefinitely](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#wait) * [Implementing Fairness in Java](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#fairness)   + [Using Locks Instead of Synchronized Blocks](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#locks)   + [A Fair Lock](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#fairlock)   + [A Note on Performance](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#performance)   **Incomplete coding** |
| 18 | [Nested Monitor Lockout](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html)  **Incomplete coding and theory** |
| 19 | [Slipped Conditions](http://tutorials.jenkov.com/java-concurrency/slipped-conditions.html) What is Slipped Conditions? Slipped conditions means, that from the time a thread has checked a certain condition until it acts upon it, the condition has been changed by another thread so that it is errornous for the first thread to act |
| 20 | [Locks in Java](http://tutorials.jenkov.com/java-concurrency/locks.html)  A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks. Locks (and other more advanced synchronization mechanisms A Simple Lock public int inc(){  lock.lock();  int newCount = ++count;  lock.unlock();  return newCount;  }  }  The lock() method locks the Lock instance so that all threads calling lock() are blocked until unlock() is executed. Lock Reentrances Synchronized blocks in Java are reentrant.  If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. Calling unlock() From a finally-clause |
| 21 | [Read / Write Locks in Java](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html) |
| 22 | [Reentrance Lockout](http://tutorials.jenkov.com/java-concurrency/reentrance-lockout.html) |
| 23 | [Semaphores](http://tutorials.jenkov.com/java-concurrency/semaphores.html) |
| 24 | [Blocking Queues](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) |
| 25 | [Thread Pools](http://tutorials.jenkov.com/java-concurrency/thread-pools.html) |
| 26 | [Compare and Swap](http://tutorials.jenkov.com/java-concurrency/compare-and-swap.html) |
| 27 | [Anatomy of a Synchronizer](http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html) |
| 28 | [Non-blocking Algorithms](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html) |
| 29 | [Amdahl's Law](http://tutorials.jenkov.com/java-concurrency/amdahls-law.html) |
| 30 | [Java Concurrency References](http://tutorials.jenkov.com/java-concurrency/references.html) |