**Java Concurrency Tutorial**

In simple words, [**concurrency**](https://en.wikipedia.org/wiki/Concurrency_%28computer_science%29) is the ability to run several programs or several parts of a program in parallel. Concurrency enable a program to achieve high performance and throughput by utilizing the untapped capabilities of underlying operating system and machine hardware. e.g. modern computers has several CPU’s or several cores within one CPU, program can utilize all cores for some part of processing; thus completing task much before in time in comparison to sequential processing.

The backbone of **java concurrency** are threads. A thread is a lightweight process which has its own call stack, but can access shared data of other threads in the same process. A Java application runs by default in one process. Within a Java application you can work with many threads to achieve parallel processing or concurrency.

**What makes java application concurrent?**

The very first class, you will need to make a java class concurrent, is [java.lang.Thread](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html" \t "_blank) class. This class is the basis of all concurrency concepts in java. Then you have [java.lang.Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html" \t "_blank) interface to abstract the thread behavior out of thread class.

Other classes you will need to build advance applications can be found at [java.util.concurrent](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html" \t "_blank) package added in Java 1.5.

**Read More:** [Java concurrency evolution](https://howtodoinjava.com/core-java/multi-threading/java-multi-threading-evolution-and-topics/)

**Is java concurrency really that simple?**

Above description gives impression that concurrency is indeed a good concept, and is very easy to implement. Well, it is not. It requires a good amount of understanding of the basic concepts – as well as – clear understanding of application goals.

Concurrent applications usually have more complex design in comparison to single threaded application. Code executed by multiple threads accessing shared data need special attention. Errors arising from incorrect thread synchronization are very hard to detect, reproduce and fix. They usually shows up in higher environments like production, and replicating the error is sometimes not possible in lower environments.

Apart from complex defects, concurrency requires more resources to run the application. So make sure, you have sufficient resources in your kitty.

# Java Multi-threading Evolution and Topics

October 21, 2015 by Lokesh Gupta

One of our reader, Anant, asked this extremely good question to elaborate / list down all related topics that we should know about multi-threading including changes made in java 8.( Beginner level to Advance level). All he wanted to know was evolution of Multi-threading Framework in Java from Simple Runnable interface to latest feature in Java 8. Let us solve his query.

I spent good amount of time in collecting all below information. So please feel free to suggest edit/update in below information if you think otherwise on any point.

## JDK release-wise multi-threading concepts

As per [**JDK 1.x release**](https://www.cs.princeton.edu/courses/archive/fall97/cs461/jdkdocs/), there were only few classes present in this initial release. To be very specific, there classes/interfaces were:

* java.lang.Thread
* java.lang.ThreadGroup
* java.lang.Runnable
* java.lang.Process
* java.lang.ThreadDeath
* and some exception classes

e.g.

1. java.lang.IllegalMonitorStateException
2. java.lang.IllegalStateException
3. java.lang.IllegalThreadStateException.

It also had few synchronized collections e.g. java.util.Hashtable.

**JDK 1.2** and **JDK 1.3** had no noticeable changes related to multi-threading. (Correct me if I have missed anything).

**JDK 1.4**, there were few JVM level changes to suspend/resume multiple threads with single call. But no major API changes were present.

[**JDK 1.5**](http://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html) was first big release after JDK 1.x; and it had included multiple concurrency utilities. Executor, semaphore, mutex, barrier, latches, concurrent collections and blocking queues; all were included in this release itself. The biggest change in java multi-threading applications cloud happened in this release.

Read full set of changes in this link: <http://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html>

**JDK 1.6** was more of platform fixes than API upgrades. So new change was present in JDK 1.6.

[**JDK 1.7**](http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/changes7.html) added support for ForkJoinPool which implemented **work-stealing technique** to maximize the throughput. Also Phaser class was added.

TransferQueue, ConcurrentLinkedDeque, and ThreadLocalRandom were also added.

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TransferQueue.html>  
<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadLocalRandom.html>  
<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html>

[**JDK 1.8**](http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/changes8.html) is largely known for Lambda changes, but it also had few concurrency changes as well. Two new interfaces and four new classes were added in **java.util.concurrent** package e.g. CompletableFuture and CompletionException.

The Collections Framework has undergone a major revision in Java 8 to add aggregate operations based on the newly added **streams facility** and **lambda expressions**; resulting in large number of methods added in almost all Collection classes, and thus in concurrent collections as well.

Read full set of changes in this link: <http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/changes8.html>

#### References:

* <https://www.cs.princeton.edu/courses/archive/fall97/cs461/jdkdocs/relnotes/intro.html>
* [http://programmers.stackexchange.com/questions/147205/what-were-the-core-api-packages-of-java-1-0](https://programmers.stackexchange.com/questions/147205/what-were-the-core-api-packages-of-java-1-0)
* <http://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html>
* <http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/changes7.html>
* <http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html>
* <http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/changes8.html>

# Java Concurrency – Thread Safety?

June 2, 2014 by Lokesh Gupta

Defining **thread safety** is surprisingly tricky. A quick Google search turns up numerous “definitions” like these:

1. Thread-safe code is code that will work even if many Threads are executing it simultaneously.
2. A piece of code is thread-safe if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time.

And there are more similar definitions.

Don’t you think that definitions like above actually does not communicate anything meaningful and even add some more confusion. Though these definitions can’t be ruled out just like that, because they are not wrong. But the fact is **they do not provide any practical help or perspective**. How do we make a **difference between a thread-safe class and an unsafe one**? **What do we even mean by “safe”**?

## What is Correctness in thread safety?

At the heart of any reasonable definition of thread safety is the concept of correctness. So, before understanding the thread-safety we should understand first, this “correctness“.

Correctness means that a class conforms to its specification.

You will agree that a good class specification will have all information about a class’s state at any given time and it’s post condition if some operation is performed on it. Since we often don’t write adequate specifications for our classes, how can we possibly know they are correct? We can’t, but that doesn’t stop us from using them anyway once we’ve convinced ourselves that “the code works”. This **“code confidence**” is about as close as many of us get to correctness.

Having optimistically defined “correctness” as something that can be recognized, we can now define thread safety in a somewhat less circular way: **a class is thread-safe when it continues to behave correctly when accessed from multiple threads**.

A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

If the loose use of “correctness” here bothers you, you may prefer to **think of a thread-safe class as one that is no more broken in a concurrent environment than in a single-threaded environment**. Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own.

## Example: A Stateless Servlet

A good example of thread safe class is java servlets which have no fields and references, no fields from other classes etc. They are **stateless**.

|  |
| --- |
| public class StatelessFactorizer implements Servlet  {      public void service(ServletRequest req, ServletResponse resp)      {          BigInteger i = extractFromRequest(req);          BigInteger[] factors = factor(i);          encodeIntoResponse(resp, factors);      }  } |

The transient state for a particular computation exists solely in local variables that are stored on the thread’s stack and are accessible only to the executing thread. One thread accessing a StatelessFactorizer cannot influence the result of another thread accessing the same StatelessFactorizer; because the two threads do not share state, it is as if they were accessing different instances. Since the actions of a thread accessing a stateless object cannot affect the correctness of operations in other threads, stateless objects are thread-safe.

That’s all for this small but important concept around **What is Thread Safety**?

**Happy Learning !!**

1. vikash

April 5, 2015

Hi Lokesh,

This is a excellent website. Thanks a lot.

Please help me in one query that if My Class is running and I dont want any thread to interrupt it even from another JVM. Then whats should I do ?  
Will Class level lock works from another JVM also ?

[Reply](https://howtodoinjava.com/core-java/multi-threading/what-is-thread-safety/#comment-15489)

* + [Lokesh](https://howtodoinjava.com/)

April 6, 2015

Threads can access another threads only inside same JVM.

[Reply](https://howtodoinjava.com/core-java/multi-threading/what-is-thread-safety/#comment-15493)

1. Ramakrishna

June 3, 2014

If a method parameters are primitive, each thread can maintain these values in its own stack, what about a method which has reference(s) to other objects as parameters, will each thread then also maintain the state of these objects in thread stack.. how this works.. pls explain,

[Reply](https://howtodoinjava.com/core-java/multi-threading/what-is-thread-safety/#comment-3870)

* + Rajesh

June 4, 2014

State of objects will always be on heap not stack. If you have defined the objects with in the method then in this case each thread will have its own copy of objects, hence threadsafe.

# Concurrency vs. Parallelism

**Concurrency** means multiple tasks which start, run, and complete in overlapping time periods, in no specific order. **Parallelism** is when multiple tasks OR several part of a unique task literally run at the same time, e.g. on a multi-core processor. Remember that Concurrency and parallelism are NOT the same thing.

Let’s understand more in detail that what I mean when I say **Concurrency vs. Parallelism**.

## Concurrency

Concurrency is essentially applicable when we talk about minimum two tasks or more. When an application is capable of executing two tasks virtually at same time, we call it concurrent application. Though here tasks run looks like simultaneously, but essentially they MAY not. They take advantage of **CPU time-slicing** feature of operating system where each task run part of its task and then go to waiting state. When first task is in waiting state, CPU is assigned to second task to complete it’s part of task.

Operating system based on priority of tasks, thus, assigns CPU and other computing resources e.g. memory; turn by turn to all tasks and give them chance to complete. To end user, it seems that all tasks are running in parallel. This is called concurrency.

## Parallelism

Parallelism does not require two tasks to exist. It literally physically run parts of tasks OR multiple tasks, at the same time using multi-core infrastructure of CPU, by assigning one core to each task or sub-task.

Parallelism requires hardware with multiple processing units, essentially. In single core CPU, you may get concurrency but NOT parallelism.

## Differences between concurrency vs. parallelism

Now let’s list down remarkable differences between concurrency and parallelism.

Concurrency is when two tasks can start, run, and complete in overlapping time periods. Parallelism is when tasks literally run at the same time, eg. on a multi-core processor.

Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.

Concurrency is about **dealing with lots of things** at once. Parallelism is about **doing lots of things** at once.

An application can be concurrent – but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at same time instant.

An application can be parallel – but not concurrent, which means that it processes multiple sub-tasks of a task in multi-core CPU at same time.

An application can be neither parallel – nor concurrent, which means that it processes all tasks one at a time, sequentially.

An application can be both parallel – and concurrent, which means that it processes multiple tasks concurrently in multi-core CPU at same time .

That’s all about **Concurrency vs. Parallelism**, a very important concept in [java multi-threading](https://howtodoinjava.com/java-concurrency-tutorial/) concepts.

# Concurrency vs. Parallelism

**Concurrency** means multiple tasks which start, run, and complete in overlapping time periods, in no specific order. **Parallelism** is when multiple tasks OR several part of a unique task literally run at the same time, e.g. on a multi-core processor. Remember that Concurrency and parallelism are NOT the same thing.

Let’s understand more in detail that what I mean when I say **Concurrency vs. Parallelism**.

## Concurrency

Concurrency is essentially applicable when we talk about minimum two tasks or more. When an application is capable of executing two tasks virtually at same time, we call it concurrent application. Though here tasks run looks like simultaneously, but essentially they MAY not. They take advantage of **CPU time-slicing** feature of operating system where each task run part of its task and then go to waiting state. When first task is in waiting state, CPU is assigned to second task to complete it’s part of task.

Operating system based on priority of tasks, thus, assigns CPU and other computing resources e.g. memory; turn by turn to all tasks and give them chance to complete. To end user, it seems that all tasks are running in parallel. This is called concurrency.

## Parallelism

Parallelism does not require two tasks to exist. It literally physically run parts of tasks OR multiple tasks, at the same time using multi-core infrastructure of CPU, by assigning one core to each task or sub-task.

Parallelism requires hardware with multiple processing units, essentially. In single core CPU, you may get concurrency but NOT parallelism.

## Differences between concurrency vs. parallelism

Now let’s list down remarkable differences between concurrency and parallelism.

Concurrency is when two tasks can start, run, and complete in overlapping time periods. Parallelism is when tasks literally run at the same time, eg. on a multi-core processor.

Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.

Concurrency is about **dealing with lots of things** at once. Parallelism is about **doing lots of things** at once.

An application can be concurrent – but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at same time instant.

An application can be parallel – but not concurrent, which means that it processes multiple sub-tasks of a task in multi-core CPU at same time.

An application can be neither parallel – nor concurrent, which means that it processes all tasks one at a time, sequentially.

An application can be both parallel – and concurrent, which means that it processes multiple tasks concurrently in multi-core CPU at same time .

That’s all about **Concurrency vs. Parallelism**, a very important concept in [java multi-threading](https://howtodoinjava.com/java-concurrency-tutorial/) concepts.

# Compare and Swap [CAS] Algorithm

One of the best additions in java 5 was Atomic operations supported in classes such as AtomicInteger, AtomicLong etc. These classes help you in minimizing the need of complex (un-necessary) [**multi-threading**](https://howtodoinjava.com/category/core-java/multi-threading/) code for some basic operations such as increment or decrement a value which is shared among multiple threads. These classes internally rely on an algorithm named **CAS (compare and swap)**. In this article, I am going to discuss this concept in detail.

## Optimistic and Pessimistic Locking

Traditional locking mechanisms, e.g. **using synchronized keyword in java, is said to be pessimistic technique**of locking or multi-threading. It asks you to first guarantee that no other thread will interfere in between certain operation (i.e. lock the object), and then only allow you access to any instance/method.

It’s much like saying “please close the door first; otherwise some other crook will come in and rearrange your stuff”.

Though above approach is safe and it does work, but it **put a significant penalty on your application in terms of performance**. Reason is simple that waiting threads cannot do anything unless they also get a chance and perform the guarded operation.

There exist one more approach which is more efficient in performance, and it **optimistic** in nature. In this approach, you proceed with an update, **being hopeful that you can complete it without interference**. This approach relies on collision detection to determine if there has been interference from other parties during the update, in which case the operation fails and can be retried (or not).

The optimistic approach is like the old saying, “It is easier to obtain forgiveness than permission”, where “easier” here means “more efficient”.

**Compare and Swap** is a good example of such optimistic approach, which we are going to discuss next.

## Compare and Swap Algorithm

This algorithm compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple Boolean response (this variant is often called compare-and-set), or by returning the value read from the memory location (not the value written to it).

There are 3 parameters for a CAS operation:

1. A memory location V where value has to be replaced
2. Old value A which was read by thread last time
3. New value B which should be written over V

CAS says “I think V should have the value A; if it does, put B there, otherwise don’t change it but tell me I was wrong.” CAS is an optimistic technique—it proceeds with the update in the hope of success, and can detect failure if another thread has updated the variable since it was last examined.

Let’s understand thw whole process with an ***example***. Assume V is a memory location where value “10” is stored. There are multiple threads who want to increment this value and use the incremented value for other operations, a very practical scenario. Let’s break the whole CAS operation in steps:

**1) Thread 1 and 2 want to increment it, they both read the value and increment it to 11.**

V = 10, A = 0, B = 0

**2) Now thread 1 comes first and compare V with it’s last read value:**

V = 10, A = 10, B = 11

if     A = V

   V = B

else

   operation failed

   return V

Clearly the value of V will be overwritten as 11, i.e. operation was successful.

**3) Thread 2 comes and try the same operation as thread 1**

V = 11, A = 10, B = 11

if     A = V

   V = B

else

   operation failed

   return V

**4) In this case, V is not equal to A, so value is not replaced and current value of V i.e. 11 is returned. Now thread 2, again retry this operation with values:**

V = 11, A = 11, B = 12

And this time, condition is met and incremented value 12 is returned to thread 2.

In summary, when multiple threads attempt to update the same variable simultaneously using CAS, one wins and updates the variable’s value, and the rest lose. But the losers are not punished by suspension of thread. They are free to retry the operation or simply do nothing.

Thats all for this simple but important concept related to atomic operations supported in java.

**Object level Locking vs. Class level Locking in Java**

March 8, 2013 by Lokesh Gupta

**Synchronization** refers to multi-threading. A synchronized block of code can only be executed by one thread at a time.

Java supports multiple threads to be executed. This may cause two or more threads to access the same fields or objects. Synchronization is a process which keeps all concurrent threads in execution to be in synch. Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as synchronized; the thread holds the monitor for that method’s object If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.

Synchronization in java is achieved using ***synchronized***keyword.***You can use synchronized keyword in your class on defined methods or blocks. Keyword can not be used with variables or attributes in class definition.***

**Object level locking**

Object level locking is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe. This can be done as below :

|  |
| --- |
| public class DemoClass  {      public synchronized void demoMethod(){}  }    or    public class DemoClass  {      public void demoMethod(){          synchronized (this)          {              //other thread safe code          }      }  }    or    public class DemoClass  {      private final Object lock = new Object();      public void demoMethod(){          synchronized (lock)          {              //other thread safe code          }      }  } |

**Class level locking**

Class level locking prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of  DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

|  |
| --- |
| public class DemoClass  {      public synchronized static void demoMethod(){}  }    or    public class DemoClass  {      public void demoMethod(){          synchronized (DemoClass.class)          {              //other thread safe code          }      }  }    or    public class DemoClass  {      private final static Object lock = new Object();      public void demoMethod(){          synchronized (lock)          {              //other thread safe code          }      }  } |

**Some Important notes**

1. Synchronization in java guarantees that no two threads can execute a synchronized method which requires same lock simultaneously or concurrently.
2. synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
3. When ever a thread enters into java synchronized method or block it acquires a lock and whenever it leaves java synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
4. java synchronized keyword is re-entrant in nature it means if a java synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
5. Java Synchronization will throw NullPointerException if object used in java synchronized block is null. For example, in above code sample if lock is initialized as null, the synchronized (lock) will throw NullPointerException.
6. Synchronized methods in Java put a performance cost on your application. So use synchronization when it is absolutely required. Also, consider using synchronized code blocks for synchronizing only critical section of your code.
7. It’s possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.
8. According to the Java language specification you can not use java synchronized keyword with constructor it’s illegal and result in compilation error.
9. Do not synchronize on non final field on synchronized block in Java. because reference of non final field may change any time and then different thread might synchronizing on different objects i.e. no synchronization at all. Best is to use String class, which is already immutable and declared final.

# Difference between Runnable vs Thread in Java

March 12, 2013 by Lokesh Gupta

In java language, as we all know that there are two ways to create threads. One using [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html" \t "_blank) interface and another by extending [Thread](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html) class. Let’s identify the differences between both ways i.e Runnable vs Thread.

## Java Code to Create Threads using Runnable Interface and Thread class

Let’s quickly check the java code of usage of both techniques.

|  |
| --- |
| public class DemoRunnable implements Runnable {      public void run() {          //Code      }  }  //with a "new Thread(demoRunnable).start()" call    public class DemoThread extends Thread {      public DemoThread() {          super("DemoThread");      }      public void run() {          //Code      }  }  //with a "demoThread.start()" call |

## Difference between Runnable vs Thread

There has been a good amount of debate on which is better way. Well, I also tried to find out and below is my learning:

1. Implementing Runnable is the preferred way to do it. Here, you’re not really specializing or modifying the thread’s behavior. You’re just giving the thread something to run. That means composition is the better way to go.
2. Java only supports single inheritance, so you can only extend one class.
3. Instantiating an interface gives a cleaner separation between your code and the implementation of threads.
4. Implementing Runnable makes your class more flexible. If you extend thread then the action you’re doing is always going to be in a thread. However, if you extend Runnable it doesn’t have to be. You can run it in a thread, or pass it to some kind of executor service, or just pass it around as a task within a single threaded application.
5. By extending Thread, each of your threads has a unique object associated with it, whereas implementing Runnable, many threads can share the same runnable instance.
6. If you are working on JDK 4 or lesser, then there is bug :

[http://bugs.java.com/bugdatabase/view\_bug.do;jsessionid=5869e03fee226ffffffffc40d4fa881a86e3:WuuT?bug\_id=4533087](http://bugs.java.com/bugdatabase/view_bug.do;jsessionid=5869e03fee226ffffffffc40d4fa881a86e3:WuuT?bug_id=4533087%20terget=)

It’s fixed in Java 1.5 but Sun doesn’t intend to fix it in 1.4.

The issue is that at construction time, a Thread is added to a list of references in an internal thread table. It won’t get removed from that list until its start() method has completed. As long as that reference is there, it won’t get garbage collected.

That’s all about differences between Runnable interface and Thread class in java. If you know something more, please put that in comments section and I will include in post content.