<http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#read-modify-write%20critical-sections>

# Race Conditions and Critical Sections

* [Race Conditions Tutorial Video](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#race-conditions-tutorial-video)
* [Two Types of Race Conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#two-types-of-race-conditions)
* [Read-Modify-Write Critical Sections](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#read-modify-write%20critical-sections)
  + [Race Conditions in Read-Modify-Write Critical Sections](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#race-conditions-in-read-modify-write-critical-sections)
* [Check-Then-Act Critical Sections](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#check-then-act-critical-sections)
* [Preventing Race Conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html)
* [Critical Section Throughput](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#critical-section-throughput)

A *race condition* is a concurrency problem 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.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition. The term race condition stems from the metaphor that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

This may all sound a bit complicated, so I will elaborate more on race conditions and critical sections in the following sections.

## Race Conditions Tutorial Video

If you prefer video, I have a video version of this tutorial here: [**Race Conditions in Java Multithreading**](https://www.youtube.com/watch?v=RMR75VzYoos&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=8)

**[](https://www.youtube.com/watch?v=RMR75VzYoos&list=PLL8woMHwr36EDxjUoCzboZjedsnhLP1j4&index=8)**

## Two Types of Race Conditions

Race conditions can occur when two or more threads read and write the same variable according to one of these two patterns:

* Read-modify-write
* Check-then-act

The read-modify-write pattern means, that two or more threads first read a given variable, then modify its value and write it back to the variable. For this to cause a problem, the new value must depend one way or another on the previous value. The problem that can occur is, if two threads read the value (into CPU registers) then modify the value (in the CPU registers) and then write the values back. This situation is explained in more detail later.

The check-then-act pattern means, that two or more threads check a given condition, for instance if a Map contains a given value, and then go on to act based on that information, e.g. taking the value from the Map. The problem may occur if two threads check the Map for a given value at the same time - see that the value is present - and then both threads try to take (remove) that value. However, only one of the threads can actually take the value. The other thread will get a null value back. This could also happen if a Queue was used instead of a Map.

## Read-Modify-Write Critical Sections

As mentioned above, a read-modify-write critical section can lead to race conditions. In this section I will take a closer look at why that is. Here is a read-modify-write critical section Java code example 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 Java virtual machine. 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 result 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 and 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 Read-Modify-Write Critical Sections

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

More formally, 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.

## Check-Then-Act Critical Sections

As also mentioned above, a check-then-act critical section can also lead to race conditions. If two threads check the same condition, then act upon that condition in a way that changes the condition it can lead to race conditions. If two threads both check the condition at the same time, and then one thread goes ahead and changes the condition, this can lead to the other thread acting incorrectly on that condition.

To illustrate how a check-then-act critical section can lead to race conditions, look at the following example:

public class CheckThenActExample {

public void checkThenAct(Map<String, String> sharedMap) {

if(sharedMap.containsKey("key")){

String val = sharedMap.remove("key");

if(val == null) {

System.out.println("Value for 'key' was null");

}

} else {

sharedMap.put("key", "value");

}

}

}

If two or more threads call the checkThenAct() method on the same CheckThenActExample object, then two or more threads may execute the if-statement at the same time, evaluate sharedMap.containsKey("key") to true, and thus move into the body code block of the if-statement. In there, multiple threads may then try to remove the key,value pair stored for the key "key", but only one of them will actually be able to do it. The rest will get a null value back, since another thread already removed the key,value pair.

## 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**](http://tutorials.jenkov.com/java-concurrency/synchronized.html). Thread synchronization can also be achieved using other synchronization constructs like [**locks**](http://tutorials.jenkov.com/java-concurrency/locks.html) or atomic variables like [**java.util.concurrent.atomic.AtomicInteger**](http://tutorials.jenkov.com/java-util-concurrent/atomicinteger.html).

## Critical Section Throughput

For smaller critical sections making the whole critical section a synchronized block may work. But, for larger critical sections it may be beneficial to break the critical section into smaller critical sections, to allow multiple threads to execute each a smaller critical section. This may decrease contention on the shared resource, and thus increase throughput of the total critical section.

Here is a very simplified Java code example to show what I mean:

public class TwoSums {

private int sum1 = 0;

private int sum2 = 0;

public void add(int val1, int val2){

synchronized(this){

this.sum1 += val1;

this.sum2 += val2;

}

}

}

Notice how the add() method adds values to two different sum member variables. To prevent race conditions the summing is executed inside a Java synchronized block. With this implementation only a single thread can ever execute the summing at the same time.

However, since the two sum variables are independent of each other, you could split their summing up into two separate synchronized blocks, like this:

public class TwoSums {

private int sum1 = 0;

private int sum2 = 0;

private Integer sum1Lock = new Integer(1);

private Integer sum2Lock = new Integer(2);

public void add(int val1, int val2){

synchronized(this.sum1Lock){

this.sum1 += val1;

}

synchronized(this.sum2Lock){

this.sum2 += val2;

}

}

}

Now two threads can execute the add() method at the same time. One thread inside the first synchronized block, and another thread inside the second synchronized block. The two synchronized blocks are synchronized on different objects, so two different threads can execute the two blocks independently. This way threads will have to wait less for each other to execute the add() method.

This example is very simple, of course. In a real life shared resource the breaking down of critical sections may be a whole lot more complicated, and require more analysis of execution order possibilities.

Race conditions is a situation where:

- Two or more threads access the same variable(s) (or data) in a way where the final result stored in the variable depends on how thread access to the variables is scheduled

Race conditions occur when:

- Two or more threads read and write the same variable(s) or data concurrently.

- The threads access the variable(s) using either of these patterns:

- Check then act.

- Read modify write.

- Where the modified value depends on the previously read value

- The thread access to the variable(s) or data is not atmoic.