**1. Overview**

In this quick article, we’ll focus on a foundational but often misunderstood concept in the Java language – the *volatile* keyword.

In Java, each thread has a separate memory space known as working memory; this holds the values of different variables used for performing operations. After performing an operation, thread copies the updated value of the variable to the main memory, and from there other threads can read the latest value.

Simply put, the*volatile* keyword marks a variable to always go to main memory, for both reads and writes, in the case of multiple threads accessing it.

**2. When to Use *volatile***

In the situations where the next value of the variable is dependent on the previous value, there is a chance that multiple threads reading and writing the variable may go out of sync, due to a time gap between the reading and writing back to the main memory.

This can be illustrated by a simple example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class SharedObject {      private volatile int count = 0;        public void increamentCount() {          count++;      }      public int getCount() {          return count;      }  } |

**With no synchronization here, a typical race condition can occur.** Basically, with an execution gap between incrementing and writing it to main memory, other threads might see a value of 0, and try to write it to main memory.

The race condition can of course also be avoided with the use of the Java provided atomic data types like *AtomicInt* or *AtomicLong*.

**3. Volatile and Thread Synchronization**

For all the multithreaded applications, we need to ensure a couple of rules for a consistent behavior:

* Mutual Exclusion – only one thread executes a critical section at a time
* Visibility – changes made by one thread to the shared data, are visible to other threads to maintain a data consistency

*Synchronized* methods and blocks provide both of the above properties, at the cost of performance of the application.

*Volatile* is quite a useful primitive because it **can help ensure the visibility aspect of the data change, without, of course, providing the mutual exclusion**. Thus, it’s useful in the places where we are ok with multiple threads are executing a block of code in parallel but we need to ensure the visibility property.

**4. Happens-Before Guarantee**

Starting with Java 5, the *volatile* keyword also provides additional capabilities which ensure that values of all the variables including non-volatile variables are written to the main memory along with the *Volatile* write operation.

This is called Happens-Before, as it gives a visibility of all variables to another reading thread. Also, JVM doesn’t reorder the reading and writing instructions of *volatile* variables.

Let’s have a look at the example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Thread 1      object.aNonValitileVariable = 1;      object.aVolatileVariable = 100; // volatile write    Thread 2:      int aNonValitileVariable = object.aNonValitileVariable;      int aVolatileVariable =  object.aVolatileVariable; |

In this case, when *Thread 1* has written the value of *aVolatileVariable*then the value of *aNonValitileVariable*is also written to the main memory. **And even though it’s not a *volatile*variable, it is exhibiting a *volatile* behavior.**

By making use of these semantics, we can define only a few of the variables in our class as *volatile*and optimize the visibility guarantee.

**5. Conclusion**

In this tutorial, we’ve explored more about the *volatile*keyword and its capabilities, as well as the improvements made to it starting with Java 5.

As always, the code examples can be found [over on GitHub](https://github.com/eugenp/tutorials/tree/master/core-java-concurrency).