# **Atomicity**

Source: www.internalpointers.com/post/lock-free-multithreading-atomic-operations

The Greek word "atom" (ἄτομος; atomos) means **uncuttable**. A task performed by a computer is said to be atomic when it is not divisible anymore: it **can't be broken into smaller steps.**

Atomicity is an important property of multithreaded operations: since they are indivisible, there is no way for a thread to slip through an atomic operation concurrently performed by another one. For example, when a thread atomically writes on shared data no other thread can read the modification half-complete. Conversely, when a thread atomically reads from shared data, it sees the value as it appeared at a single moment in time. In other words, there is no risk of data races.

## Aside: Race Condition vs. Data Race

Source: blog.regehr.org/archives/490

A **race condition** is a flaw that occurs when the **timing or ordering of events affects a program’s correctness**. Generally speaking, some kind of external timing or ordering non-determinism is needed to produce a race condition; typical examples are context switches, OS signals, memory operations on a multiprocessor, and hardware interrupts.

A **data race** happens when there are two memory accesses in a program where both:

* target the same location (the same memory)
* are performed concurrently by two threads
* are not reads (as opposed to a write or other modification)
* are not synchronization operations

In practice there is considerable overlap: many race conditions are due to data races, and many data races lead to race conditions.

*The web page goes on to give an example of a bad bank account with both data races and a race condition. The example of the bad bank we did is also a race condition and data race. The race condition is the error (money is lost!). The data race is that the value in memory depends on the timing of interruptions as both threads write to the same variable.*

*A data race occurs when 2 instructions from different threads access the same memory location, at least one of these accesses is a write and there is no synchronization that is mandating any particular order among these accesses.*

Source for the following: dzone.com/articles/java-concurrency-atomic-variables

Atomic operations are operations that ALWAYS execute together. Either all of them execute together, or none of them executes. If an operation is atomic, then it cannot be partially complete, either it will be complete, or not start at all, but will not be incomplete.

Source: stackoverflow.com/questions/16729364/atomic-operations-and-multithreading

Doing a = 28 (with a being an int) is an atomic operation. But doing a++ is not an atomic operation because it requires a read of the value of a, an incrementation, and a write to a of the result. As a result, if you used a++ to implement a thread-safe counter, you could have two threads reading the value concurrently, then have both increment it and writing it concurrently, resulting in 27 as a result, instead of 28.