https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

An interrupt is an indication to a thread that it should stop what it is doing and do something else.

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.

Two errors when sharing objects between threads:
1**. Thread interference (Lost Updates)**
Interference happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:

Thread A: Retrieve c.
Thread B: Retrieve c.
Thread A: Increment retrieved value; result is 1.
Thread B: Decrement retrieved value; result is -1.
Thread A: Store result in c; c is now 1.
Thread B: Store result in c; c is now -1.
Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

2. Memory Consistency (No happens before)
Memory consistency errors occur when different threads have inconsistent views of what should be the same data. The causes of memory consistency errors are complex and beyond the scope of this tutorial. Fortunately, the programmer does not need a detailed understanding of these causes. All that is needed is a strategy for avoiding them.

The key to avoiding memory consistency errors is understanding the happens-before relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement. To see this, consider the following example. Suppose a simple int field is defined and initialized:
h
int counter = 0;
The counter field is shared between two threads, A and B. Suppose thread A increments counter:

counter++;
Then, shortly afterwards, thread B prints out counter:

System.out.println(counter);
If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

However, synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Starvation and livelock are forms of thread contention.

Happens-before relationship

Intrinsic Locks and Synchronization
Synchronization is built around an internal entity known as the intrinsic lock or monitor lock. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them. A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

Synchronized methods

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html
First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.
Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock.

- more granular concurrency -- means less time wasted on blocking

(Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.)

Atomic Access
In programming, an atomic action is one that effectively happens all at once. An atomic action
cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as c++, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
Reads and writes are atomic for all variables declared volatile (including long and double variables).

Atomic writes make interleaving impossible but they do not guarantee any ordering to the operations -- so memory consistency issues still exist

Liveness
A concurrent application's ability to execute in a timely manner is known as its liveness. This section describes the most common kind of liveness problem, deadlock, and goes on to

briefly describe two other liveness problems, starvation and livelock.

Deadlock
Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Here's an example.
t1 needs a resource that t2 has acquired a lock for
t2 needs a resource that t1 has acquired a lock for

Starvation and Livelock
Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

Starvation
Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelock
A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

How to enforce thread ordering
- join a thread before continuing
- encapsulate shared state in an object with synchronized methods