# Monitors, Locks, and Synchronization

Source: <https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html>

With some helpful summarization from here: <https://stackoverflow.com/questions/49610644/in-java-what-is-the-difference-between-a-monitor-and-a-lock>

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.

When a thread releases an intrinsic lock, a **happens-before** relationship is established between that action and any subsequent acquisition of the same lock.

## Locks In Synchronized Methods

When a thread invokes a **synchronized** method, the thread automatically **acquires the intrinsic lock for that method's object** (*the whole object!*) and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus **access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class**.

# Synchronized Statements

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

public void addName(String name) {

synchronized(this) {

lastName = name;

nameCount++;

}

nameList.add(name);

}

In this example, the addName method needs to synchronize changes to lastName and nameCount, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking nameList.add.

Synchronized statements are also useful for improving concurrency with **fine-grained synchronization**. Suppose, for example, class MsLunch has two instance fields, c1 and c2, that are never used together. All updates of these fields must be synchronized, but **there's no reason to prevent an update of c1 from being interleaved with an update of c2** — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with this, we **create two objects solely to provide locks**.

public class MsLunch {

private long c1 = 0;

private long c2 = 0;

private Object lock1 = new Object();

private Object lock2 = new Object();

public void inc1() {

synchronized(lock1) {

c1++;

}

}

public void inc2() {

synchronized(lock2) {

c2++;

}

}

}

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

# Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But **a thread can acquire a lock that it already owns.** Allowing a thread to acquire the same lock more than once enables reentrant synchronization. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

Locks and monitors are not different things. Instead, they are complementary. **Every object in Java is associated with a monitor which a thread can lock or unlock.**

# **Another perspective on Locks and Monitors**

Source: howtodoinjava.com/java/multi-threading/multithreading-difference-between-lock-and-monitor/

## Locks

**Locks provide necessary support for implementing monitors.**

A lock is kind of data which is logically part of an object’s header on the **heap memory**. Each object in a JVM (Java Virtual Machine) has this lock (or **mutex**) that any program can use to coordinate multi-threaded access to the object. If any thread want to access instance variables of that object; then thread must “own” the object’s lock (set some flag in lock memory area). All other threads that attempt to access the object’s variables have to wait until the owning thread releases the object’s lock (unset the flag).

Once a thread owns a lock, it can request the same lock again multiple times, but then has to release the lock the same number of times before it is made available to other threads. **If a thread requests a lock three times, for example, that thread will continue to own the lock until it has “released” it three times.**

Please note that lock is acquired by a thread, when it explicitly asks for it. In Java, this is done with the synchronized keyword, or with wait and notify.

## Aside: Heap Memory

Source: stackoverflow.com/questions/2308751/what-is-a-memory-heap

A **memory heap** is a location in memory where memory may be allocated as random access.

Unlike the **stack** where memory is allocated and released in a very defined order, individual data elements allocated on the heap are typically released in ways which are **asynchronous** from one another. Any **such data element is freed when the program explicitly releases the corresponding pointer**, and this may result in a fragmented heap. In opposition only data at the top (or the bottom, depending on the way the stack works) may be released, resulting in data element being freed in the reverse order they were allocated.

## Monitors

Monitor is a synchronization construct that allows threads to have both **mutual exclusion** (using locks) and **cooperation** i.e. the ability to make threads wait for certain condition to be true (using wait-set).

## Mutual exclusion *(mutex)*

A monitor is like a building that contains one special room (object instance) that can be occupied by only one thread at a time. The room usually contains some data which needs to be protected from concurrent access. From the time a thread enters this room to the time it leaves, it has exclusive access to any data in the room. Entering the monitor building is called “entering the monitor.” Entering the special room inside the building is called “acquiring the monitor.” Occupying the room is called “owning the monitor,” and leaving the room is called “releasing the monitor.” Leaving the entire building is called “exiting the monitor.”

Think of that scene in the first Mission Impossible movie. Only one person is supposed to be in the room at a time.



When a thread arrives to access protected data (enter the special room), it is first put in queue in building reception (entry-set). If no other thread is waiting (own the monitor), the thread acquires the lock and continues executing the protected code. When the thread finishes execution, it releases the lock and exits the building (exiting the monitor).

If when a thread arrives, another thread already owns the monitor, it must wait in the reception queue (entry-set). When the current owner exits the monitor, the newly arrived thread must compete with any other threads also waiting in the entry-set. Only one thread will win the competition and own the lock.

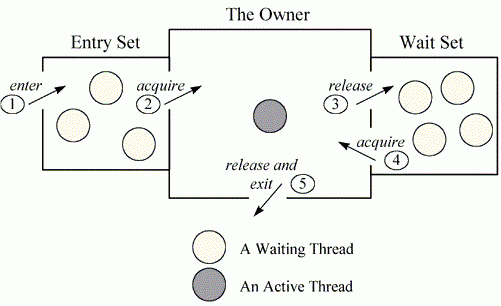
**There is no role for the wait-set feature in this example.**

## Cooperation

In general, mutual exclusion is important only when multiple threads are sharing data or some other resource. If two threads are not working with any common data or resource, they usually can’t interfere with each other and needn’t execute in a mutually exclusive way. **Whereas mutual exclusion helps keep threads from interfering with one another while sharing data, cooperation helps threads to work together towards some common goal.**

**Cooperation is important when one thread needs some data to be in a particular state and another thread is responsible for getting the data into that state e.g. producer/consumer problem** where **read thread needs the buffer to be in a “not empty” state** before it can read any data out of the buffer. If the read thread discovers that the buffer is empty, it must wait. The write thread is responsible for filling the buffer with data. Once the write thread has done some more writing, the read thread can do some more reading. It is also sometimes called a “**Wait and Notify**” OR “**Signal and Continue**” monitor because it retains ownership of the monitor and continues executing the monitor region (the continue) if needed. At some later time, the notifying thread releases the monitor and a waiting thread is resurrected to own the lock.

**This cooperation requires both i.e. entry-set (what we will call notify) and wait-set.** Consider the diagram below.



The above figure shows the monitor as three rectangles. In the center, a large rectangle contains a single thread, the monitor’s owner. On the left, a small rectangle contains the entry set. On the right, another small rectangle contains the wait set.