BASICS OF SYNCHRONIZATION IN JAVA

When several threads try to access a single resource, there is a chance that it might corrupt that data. **Synchronization ensures that when several threads want to access a single resource, only one thread can access it at any given time;** other threads trying to access will have to wait for its turn.
A synchronized region is guarded by a lock object. **When a thread enters a synchronized region, it acquires and holds the lock and when it leaves the synchronized region, it releases that lock.** Only one thread can acquire a lock at a time or in other words, only one thread can enter a synchronized region at a time.
Any object in java can act as a lock for synchronization. This is because every object has an intrinsic lock (or monitor) associated with it. Whenever a thread tries to access a synchronized block or method, it acquires the intrinsic lock or the monitor on the object on which it is synchronized. From Java 5, you can also use explicit locks from the java.util.concurrent.locks package, instead of intrinsic locks. *Here we will see only intrinsic locks, and will see explicit locks and their advantages in another tutorial.*
We can either synchronize code blocks or even whole methods.

We write **synchronized code blocks** as:

```
synchronized(myObject)
{

//synchronized code

}
```

Here myObject is any object.

We write **synchronized methods** as:

```
public synchronized void myMethod()
{

//synchronized code

}
```

In the case of non-static synchronized methods, the lock object is not specified explicitly and the method will be synchronized on the instance on which this method is called. Above code is same as synchronizing all contents of a method on the ‘this’ pointer from within the method as below:

```
public void myMethod()

{
```

```
synchronized(this){

//synchronized code

}

}
```

Static Synchronization

Static methods are synchronized on the class object, not on the 'this' reference. This is because the 'this' variable refers to the current instance and is available only to instance methods, but not to static methods; static methods belong to class and are not associated with any particular instance. Every class in Java has a class object corresponding to it represented by the class 'java.lang.Class', whose use comes mainly in reflection. A synchronized static method is same as synchronizing all contents of a method on a java.lang.Class instance.

```
public static void myMethod()

{

synchronized(MyClass.class){
//synchronized code

}

}
```

Which synchronization to use? Static vs. Instance vs. Block

When you use **static** synchronization, it will block **all static methods of all the objects of that class** that are synchronized, and any other code block that is synchronized on the .class object. When you use **instance**method synchronization, it will block **all methods of that particular object** that are synchronized, and any other code block that is synchronized on that object. When you use **block** synchronization, it will block **only code that are locked on the same object** that this block is synchronized on. You should always try to use block level synchronization instead of static and method, whenever possible.

Reentrancy

If a thread already holding a lock, try to enter another synchronized block, synchronized on the same lock that it holds, it will succeed acquiring that lock and enter that synchronized block. This property is called reentrancy.

Java implements reentrancy by associating an acquisition count and owning thread with each lock. Owner thread is recorded and acquisition count is incremented to one, when a thread acquires a lock for first time. If the same thread acquires that lock again, the count is incremented; when the thread exits a synchronized block, the count is decremented; and when the count becomes zero, the lock is released. Associated owner thread details of a lock is used to determine if the requesting thread already has that lock.

There is also an explicit lock called **ReentrantLock** in java.util.concurrent.locks package with the same basic reentrant behavior and semantics as the implicit monitor lock, but with extended capabilities.

Volatile

Synchronization provides two features: mutual exclusion and visibility. Mutual exclusion ensures that when several threads want to access a single resource, only one thread can access it at any given time, and we saw that already. Synchronization also provides visibility. Without synchronization (or volatile), there is no guarantee that values updated by one thread on a variable, will be visilbe to other threads. You might even see stale values in case of 64 bit primitives like long and double as the value may be written as two 32-bit values. By synchronizing the write and read to a variable, the latest value written to a variable by one thread will be visible to all other threads, and there won't be any stale value. So synchronization also provides memory visibility in addition to mutual exclusion.

If you require only visibility, but no mutual exclusion, then you can declare that variable as volatile as it provides same memory visibility advantage with synchronization, but without mutual exclusion, and also has slight performance advantage over synchronization.

Source : http://javajee.com/basics-of-synchronization-in-java