Thread synchronisation

Java provides a mechanism to control the access of multiple threads to a shared resource of any type. The mechanism is known as **thread synchronisation**.

Important terms and concepts

1) A **critical section** is a region of code that accesses shared resources and should not be executed by more than one thread at the same time. A shared resource may be a variable, file, input/output port, database or something else.

Let's consider an example. A class has a static field named counter:
public static long counter = 0;

Two threads increment the field (increase by 1) 10 000 000 times concurrently. The final value should be 20 000 000. But, as we've discussed in previous topics, the result often might turn out wrong, for example, 10 999 843.

This happens because sometimes a thread does not see changes of shared data performed by another thread, and sometimes a thread may see an intermediate value of the non-atomic operation. Those are visibility and atomicity problems we deal with while working with shared data.

The **monitor** is a special mechanism to control concurrent access to an object. In Java, each object has an associated implicit monitor. A thread can acquire a monitor, then other threads cannot acquire this monitor at the same time. They will wait until the owner (the thread that acquired the monitor) releases it.

Thus, a thread can be locked by the **monitor** of an object and wait for its release. This mechanism allows programmers to protect **critical sections** from being accessed by multiple threads concurrently.

The synchronised keyword

The "classic" and simplest way to protect code from being accessed by multiple threads concurrently is using the keyword **synchronized**.

It is used in two different forms:

- synchronized method (a static or an instance method)
- synchronized blocks or statements (inside a static or an instance method)

A synchronized method or block needs an object for locking threads. The monitor associated with this object controls concurrent access to the specified critical section. Only one thread can execute code in a synchronized block or method at the same time. Other threads are blocked until the thread inside the synchronized block or method exits it.

Static synchronized methods

When we synchronize static methods using the **synchronized** keyword the monitor is the class itself. Only one thread can execute the body of a synchronized static method at the same time. This can be summarized as *"one thread per class"*.

```
class SomeClass {
    public static synchronized void doSomething() {
        String threadName = Thread.currentThread().getName();
        System.out.println(String.format("%s entered the method", threadName));
        System.out.println(String.format("%s leaves the method", threadName));
    }
}
```

The method is synchronized on the object of SomeClass class the static method belongs to. Java creates a single special object for each class. To get it use class name plus .class prefix. In our case it is SomeClass.class.

Let's call the method from two threads concurrently. The result will always be similar to:

```
Thread-0 entered the method
Thread-0 leaves the method
Thread-1 entered the method
Thread-1 leaves the method
```

It's impossible for more than one thread to execute code inside the method at the same time.

Instance synchronized methods

Instance methods are synchronized on the instance (object). The monitor is the current **this** object that owns the method. If we have two instances of a class, each instance has its own monitor for synchronization.

Only one thread can execute code in a synchronized instance method of a particular instance. But different threads can execute methods of different objects at the same time. This can be summarized as *"one thread per instance"*.

```
class SomeClass {
   private String name;
   public SomeClass(String name) {
      this.name = name;
   }

   public synchronized void doSomething() {
      String threadName = Thread.currentThread().getName();
      System.out.println(String.format("%s entered the method of %s",
threadName, name));
      System.out.println(String.format("%s leaves the method of %s",
threadName, name));
   }
}

class MyThread extends Thread {
   private SomeClass someClass;

   public MyThread(SomeClass someClass) {
      this.someClass = someClass;
   }

   @Override
   public void run() {
      someClass.doSomething();
   }
}


SomeClass instance1 = new SomeClass("instance-1");
SomeClass instance2 = new SomeClass("instance-2");

MyThread first = new MyThread(instance1);
MyThread second = new MyThread(instance1);
MyThread third = new MyThread(instance2);

first.start();
second.start();
third.start();
```

The result will look like this:

```
Thread-0 entered the method of instance-1
Thread-2 entered the method of instance-2
Thread-0 leaves the method of instance-1
Thread-1 entered the method of instance-1
Thread-2 leaves the method of instance-2
Thread-1 leaves the method of instance-1
```

As you can see, there are no threads executing the code in doSomething of the instance-1 at the same time.

Synchronized blocks (statements)

Sometimes you need to synchronize only a part of a method. This is possible by using synchronized blocks (statements). They must specify an object for locking threads.

```
class SomeClass {
   public static void staticMethod() {
      // unsynchronized code
      ...
      synchronized (SomeClass.class) { // synchronization on object of
SomeClass class
         // synchronized code
         ...
      }
   }

   public void instanceMethod() {
      // unsynchronized code
      ...
      synchronized (this) { // synchronization on this instance
         // synchronized code
         ...
      }
   }
}
```

The block inside staticMethod is synchronized on the SomeClass.class object, which means only one thread can execute code in this block.

The block inside instanceMethod is synchronized on this instance, which means only one thread can execute the block of the instance. But some other thread is able to execute the block of different instances at the same time.

Synchronized blocks may resemble synchronized methods, but they allow programmers to synchronize only necessary parts of methods.

The main difference between static and instance methods is a synchronization monitor. Applying syncronized keyword to a static method uses the class object as a monitor. Instance method synchronization uses the instance itself as a monitor. Applying synchronization to block is the most agile way. It allows configuring bondaries of critical section as well as specifying synchronization monitor.

More concept:

https://stackoverflow.com/questions/15438727/if-i-synchronized-two-methods-on-the-same-class-can-they-run-simultaneously