Thready Safety & Synchronized

This lesson explains thread-safety and the use of the synchronized keyword.

> With the abstract concepts discussed, we'll now turn to the concurrency constructs offered by Java and use them in later sections to solve practical coding problems.

Thread Safe

A class and its public APIs are labelled as ***thread safe*** if multiple threads can consume the exposed APIs without causing race conditions or state corruption for the class. Note that composition of two or more thread-safe classes doesn't guarantee the resulting type to be thread-safe.

Synchronized

Java's most fundamental construct for thread synchronization is the `synchronized` keyword. It can be used to restrict access to critical sections one thread at a time.

Each object in Java has an entity associated with it called the "monitor lock" or just monitor. Think of it as an exclusive lock. Once a thread gets hold of the monitor of an object, it has exclusive access to all the methods marked as synchronized. No other thread will be allowed to invoke a method on the object that is marked as synchronized and will block, till the first thread releases the monitor which is equivalent of the first thread exiting the synchronized method.

Note carefully:

Note carefully:

1. For static methods, the monitor will be the class object, which is distinct from the monitor of each instance of the same class.
2. If an uncaught exception occurs in a synchronized method, the monitor is still released.
3. Furthermore, synchronized blocks can be re-entered.

You may think of "synchronized" as the mutex portion of a monitor.

```
class Employee {

    // shared variable
    private String name;

    // method is synchronize on 'this' object
    public synchronized void setName(String name) {
        this.name = name;
    }

    // also synchronized on the same object
    public synchronized void resetName() {

        this.name = "";
    }

    // equivalent of adding synchronized in method
    // definition
    public String getName() {
        synchronized (this) {
            return this.name;
        }
    }
}
```

As an example look at the employee class above. All the three methods are synchronized on the **"this"** object. If we created an object and three different threads attempted to execute each method of the object, only

one will get access, and the other two will block. If we synchronized on a different object other than the ***this*** object, which is only possible for the `getName` method given the way we have written the code, then the 'critical sections' of the program become protected by two different locks. In that scenario, since `setName` and `resetName` would have been synchronized on the ***this*** object only one of the two methods could be executed concurrently. However `getName` would be synchronized independently of the other two methods and can be executed alongside one of them. The change would look like as follows:

```
class Employee {

    // shared variable
    private String name;
    private Object lock = new Object();

    // method is synchronize on 'this' object
    public synchronized void setName(String name) {
        this.name = name;
    }

    // also synchronized on the same object
    public synchronized void resetName() {

        this.name = "";
    }

    // equivalent of adding synchronized in method
    // definition
    public String getName() {
        // Using a different object to synchronize on
        synchronized (lock) {
            return this.name;
        }
    }
}
```

All the sections of code that you guard with synchronized blocks on the same object can have at most one thread executing inside of them at any given point in time. These sections of code may belong to different methods, classes or be spread across the code base.

Note with the use of the synchronized keyword, ***Java forces you to***

implicitly acquire and release the monitor-lock for the object within the same method! One can't explicitly acquire and release the monitor in different methods. This has an important ramification, ***the same thread will acquire and release the monitor!*** In contrast, if we used semaphore, we could acquire/release them in different methods or by different threads.

A classic newbie mistake is to synchronize on an object and then somewhere in the code reassign the object. As an example, look at the code below. We synchronize on a Boolean object in the first thread but sleep before we call `wait()` on the object. While the first thread is asleep, the second thread goes on to change the `flag`'s value. When the first thread wakes up and attempts to invoke `wait()`, it is met with a **IllegalMonitorState** exception! The object the first thread synchronized on before going to sleep has been changed, and now it is attempting to call `wait()` on an entirely different object without having synchronized on it.

```
class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        IncorrectSynchronization.runExample();
    }
}

class IncorrectSynchronization {

    Boolean flag = new Boolean(true);

    public void example() throws InterruptedException {

        Thread t1 = new Thread(new Runnable() {

            public void run() {
                synchronized (flag) {
                    try {
                        while (flag) {
                            System.out.println("First thread about to sleep");
                            Thread.sleep(5000);
                            System.out.println("Woke up and about to invoke wait()");
                            flag.wait();
                        }
                    } catch (InterruptedException ie) {

                    }
                }
            }
        });
```

```
        Thread t2 = new Thread(new Runnable() {

            public void run() {

                flag = false;
                System.out.println("Boolean assignment done.");
            }
        });

        t1.start();
        Thread.sleep(1000);
        t2.start();
        t1.join();
        t2.join();
    }

    public static void runExample() throws InterruptedException {
        IncorrectSynchronization incorrectSynchronization = new IncorrectSynchronization();
        incorrectSynchronization.example();
    }
}
```




Marking all the methods of a class `synchronized` in order to make it thread-safe may reduce throughput. As a naive example, consider a class with two completely independent properties accessed by getter methods. Both the getters synchronize on the same object, and while one is being invoked, the other would be blocked because of synchronization on the same object. The solution is to lock at a finer granularity, possibly use two different locks for each property so that both can be accessed in parallel.