@Synchronized

`synchronized` done right: Don't expose your locks.

Overview

`@Synchronized` is a safer variant of the `synchronized` method modifier. Like `synchronized`, the annotation can be used on static and instance methods only. It operates similarly to the `synchronized` keyword, but it locks on different objects. The keyword locks on `this`, but the annotation locks on a field named `$lock`, which is private.
If the field does not exist, it is created for you. If you annotate a `static` method, the annotation locks on a static field named `$LOCK` instead.

If you want, you can create these locks yourself. The `$lock` and `$LOCK` fields will of course not be generated if you already created them yourself. You can also choose to lock on another field, by specifying it as parameter to the `@Synchronized` annotation. In this usage variant, the fields will not be created automatically, and you must explicitly create them yourself, or an error will be emitted.

Locking on `this` or your own class object can have unfortunate side-effects, as other code not under your control can lock on these objects as well, which can cause race conditions and other nasty threading-related bugs.

If you would prefer `java.util.concurrent.locks` style locks (recommended if you're using virtual threads), have a look at `@Locked`.

With Lombok

```java
import lombok.Synchronized;

public class SynchronizedExample {
  private final Object readLock = new Object();

  @Synchronized
  public static void hello() {
    System.out.println("world");
  }

  @Synchronized
  public int answerToLife() {
    return 42;
  }

  @Synchronized("readLock")
  public void foo() {
    System.out.println("bar");
  }
}
```

Vanilla Java

```java
public class SynchronizedExample {
  private static final Object $LOCK = new Object[0];
  private final Object $lock = new Object[0];
  private final Object readLock = new Object();

  public static void hello() {
    synchronized($LOCK) {
      System.out.println("world");
    }
  }

  public int answerToLife() {
    synchronized($lock) {
      return 42;
    }
  }

  public void foo() {
    synchronized(readLock) {
      System.out.println("bar");
    }
  }
}
```

Supported configuration keys:

`lombok.synchronized.flagUsage` = [`warning` | `error`] **(default: not set)**
Lombok will flag any usage of `@Synchronized` as a warning or error if configured.

Small print

If `$lock` and/or `$LOCK` are auto-generated, the fields are initialized with an empty `Object[]` array, and not just a `new Object()` as most snippets showing this pattern in action use. Lombok does this because a new object is *NOT* serializable, but 0-size array is. Therefore, using `@Synchronized` will not prevent your object from being serialized.

Having at least one `@Synchronized` method in your class means there will be a lock field, but if you later remove all such methods, there will no longer be a lock field. That means your predetermined `serialVersionUID` changes. We suggest you *always* add a `serialVersionUID` to your classes if you intend to store them long-term via java's serialization mechanism. If you do so, removing all `@Synchronized` annotations from your method will not break serialization.

If you'd like to know why a field is not automatically generated when you choose your own name for the lock object: Because otherwise making a typo in the field name will result in a *very* hard to find bug!