Lab 4

ThreadLocal variables in Java

Java offers a type that is able to provide local variables to particular threads at runtime. More exactly, a thread can get its own initialized copy of a certain variable. We say that the thread owns the variable, in the sense that it will be the only thread able to modify it. The type offering this facility is `ThreadLocal` and it provides the following methods:

```
public class ThreadLocal {

 protected T initialValue( );

 public T get( );

 public void set(T value);

 public void remove( );

}
```

The class is usually used for initializing a static `ThreadLocal` object that will be used by multiple threads to obtain local instances of a variable. The `initialValue()` method is the one which is typically automatically called when the local variable is initialized for a certain thread. This happens when the thread calls `get()` for the first time in order to obtain the value. The only exception is when the thread calls `set()` to explicitly set on its own the initial value before calling `get()` for the first time. Any subsequent `get()` after the variable value is locally set will return the current value for that specific thread. The `remove()` method can be used to "reset" the variable. After this, the internal call to `initialValue()` happens again for re-initializing.

```
private static final ThreadLocal < T > uniqueNum =
   new ThreadLocal < T > () {
    // the initialValue() should be implemented
    // to get the desired value in the specific thread
    @Override protected Integer initialValue() {
     // return desired value
    }
   };
```

Immutability

The other end-run around the need to synchronize is to use *immutable* objects. Nearly all the atomicity and visibility hazards we've described so far, such as seeing stale values, losing updates, or observing an object to be in an inconsistent state, have to do with the vagaries of multiple threads trying to access the same mutable state at the same time. If an object's state cannot be modified, these risks and complexities simply go away.

An immutable object is one whose state cannot be changed after construction. Immutable objects are inherently thread-safe; their invariants are established by the constructor, and if their state cannot be changed, these invariants always hold.

Immutable objects are always thread-safe.
Immutable objects are *simple*. They can only be in one state, which is carefully controlled by the constructor. One of the most difficult elements of program design is reasoning about the possible states of complex objects. Reasoning about the state of immutable objects, on the other hand, is trivial.
Immutable objects are also *safer*. Passing a mutable object to untrusted code, or otherwise publishing it where untrusted code could find it, is dangerous — the untrusted code might modify its state, or, worse, retain a reference to it and modify its state later from another thread. On the other hand, immutable objects cannot be subverted in this manner by malicious or buggy code, so they are safe to share and publish freely without the need to make defensive copies.
Neither the Java Language Specification nor the Java Memory Model formally defines immutability, but immutability is *not* equivalent to simply declaring all fields of an object final. An object whose fields are all final may still be mutable, since final fields can hold references to mutable objects.
An object is *immutable* if:

- Its state cannot be modified after construction;
- All its fields are final[1] and It is *properly constructed* (the `this` reference does not escape during construction).

Immutable objects can still use mutable objects internally to manage their state, as illustrated by ThreeStooges in the next listing. While the `Set` that stores the names is mutable, the design of ThreeStooges makes it impossible to modify that Set after construction. The `stooges` reference is final, so all object state is reached through a final field. The last requirement, proper construction, is easily met since the constructor does nothing that would cause the `this` reference to become accessible to code other than the constructor and its caller.

```
public final class ThreeStooges {
private final Set<String> stooges = new HashSet<String>();
    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
}
public boolean isStooge(String name) { return
stooges.contains(name);
} }
```

Because program state changes all the time, you might be tempted to think that immutable objects are of limited use, but this is not the case. There is a difference between an object being immutable and the reference to it being immutable. Program state stored in immutable objects can still be updated by “replacing” immutable objects with a new instance holding new state. Many developers fear that this approach will create performance problems, but these fears are usually unwarranted. Allocation is cheaper than you might think, and immutable objects offer additional performance advantages such as reduced need for locking or defensive copies and reduced impact on garbage collection.

Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.
A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Supporting Interruption

How does a thread support its own interruption? This depends on what it's currently doing. If the thread is frequently invoking methods that throw `InterruptedException`, it simply returns from the run method after it catches that exception. For example, suppose the central message loop in the `SleepMessages` example were in the run method of a thread's `Runnable` object. Then it might be modified as follows to support interrupts:

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

Many methods that throw `InterruptedException`, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.
What if a thread goes a long time without invoking a method that throws `InterruptedException`? Then it must periodically invoke `Thread.interrupted`, which returns true if an interrupt has been received. For example:

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

In this simple example, the code simply tests for the interrupt and exits the thread if one has been received. In more complex applications, it might make more sense to throw an InterruptedException:

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

This allows interrupt handling code to be centralized in a catch clause.

The Interrupt Status Flag

The interrupt mechanism is implemented using an internal flag known as the *interrupt status*. Invoking `Thread.interrupt` sets this flag. When a thread checks for an interrupt by invoking the static method `Thread.interrupted`, interrupt status is cleared. The non-static `isInterrupted` method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an `InterruptedException` clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

Credits:
http://jcip.net
https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html