ThreadLocal

Consider the following instance method of a class

```
void add(int val) {

    int count = 5;
    count += val;
    System.out.println(val);

}
```

Do you think the above method is thread-safe? If multiple threads call this method, then each executing thread will create a copy of the local variables on its own thread stack. There would be no shared variables amongst the threads and the instance method by itself would be thread-safe.

However, if we moved the `count` variable out of the method and declared it as an instance variable then the same code will not be thread-safe.

We can have a copy of an instance (or a class) variable for each thread that accesses it by declaring the instance variable *ThreadLocal*. Look at the thread unsafe code below. If you run it multiple times, you'll see different results. The count variable is incremented 100 times by 100 threads so in a thread-safe world the final value of the variable should come out to be 10,000.

```
import java.util.concurrent.Executors;

class Demonstration {
    public static void main( String args[] ) throws Exception{

        UnsafeCounter usc = new UnsafeCounter();
        Thread[] tasks = new Thread[100];

        for (int i = 0; i < 100; i++) {
            Thread t = new Thread(() -> {
                for (int j = 0; j < 100; j++)
                    usc.increment();
            });
            tasks[i] = t;
            t.start();
        }

        for (int i = 0; i < 100; i++) {
            tasks[i].join();
        }

        System.out.println(usc.count);
    }
}

class UnsafeCounter {

    // Instance variable
```

Run Save Reset

Now we'll change the code to make the instance variable threadlocal. The change is:

```
ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 0);
```

The above code creates a separate and completely independent copy of the variable `counter` for every thread that calls the `increment()` method. Conceptually, you can think of a **ThreadLocal<T>** variable as a map that contains mapping for each thread and its copy of the threadlocal variable or equivalently a **Map<Thread, T>**. Though this is not how it is actually implemented. Furthermore, the thread specific values are stored in the thread object itself and are eligible for garbage collection once a thread terminates (if no other references exist to the threadlocal value).

The code below is a fixed version of the unsafe counter. Note that each thread has its own copy of the `counter` variable, which is incremented a 100 times. Therefore, each thread increments its own copy of counter a 100 times and that value gets printed for each thread.

ThreadLocal variables get tricky when used with the executor service (threadpools) since threads that don't terminate aren't returned to the threadpool. So any threadlocal variables for such threads aren't garbage collected. For interesting scenarios, please see Quiz#8.

```
class Demonstration {
    public static void main( String args[] ) throws Exception{
        UnsafeCounter usc = new UnsafeCounter();
        Thread[] tasks = new Thread[100];

        for (int i = 0; i < 100; i++) {
            Thread t = new Thread(() -> {
                for (int j = 0; j < 100; j++)
                    usc.increment();

                System.out.println(usc.counter.get());
            });
            tasks[i] = t;
            t.start();
        }

        for (int i = 0; i < 100; i++) {
            tasks[i].join();
        }

        System.out.println(usc.counter.get());
    }
}

class UnsafeCounter {

    ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 0);
```

Run Save Reset