ThreadLocal

The possibility of creating several threads comes with the problem of needing individual data for each of them.

Getting started with ThreadLocal

The first thing you need to remember is that ThreadLocal allows the creation of variables that will only be accessible by a specific thread.

So, every thread will have its own, independently initialized copy of the variable.

ThreadLocal<Integer> threadLocalValue = new ThreadLocal<>();

There are two main methods that define the logic of ThreadLocal : get and set.

threadLocalValue.set(123); // every thread sets its value, isolated from other threads
int result = threadLocalValue.get(); // every thread gets its value, isolated from other threads

Before you set the value, it equals null.

How this thread-association thing works:

Inside the ThreadLocal class, there is a ThreadLocalMap with the key threadLocalHashCode and the value that you set.

ThreadLocal in action

```
class MyRunnable implements Runnable {
   private static final ThreadLocal<Integer> threadLocalCounter = new
ThreadLocal<>();

   public void run() {

      if (threadLocalCounter.get() != null) {
         threadLocalCounter.set(threadLocalCounter.get() + 1);
      } else {
         threadLocalCounter.set(0);
      }

      System.out.println("threadLocalCounter of " +
Thread.currentThread().getName() + ": " + threadLocalCounter.get());
   }
}
```

ThreadLocal instances are typically private static fields because it allows classes to associate their states with a thread.

```
MyRunnable commonRunnable = new MyRunnable();

Thread t1 = new Thread(commonRunnable, "first thread");
Thread t2 = new Thread(commonRunnable, "second thread");
Thread t3 = new Thread(commonRunnable, "third thread");

// start all threads
t1.start();
t2.start();
t3.start();

// wait for threads to end
t1.join();
t2.join();
t3.join();
```

The output will look like this:

```
threadLocalCounter of second thread: 0
threadLocalCounter of third thread: 0
threadLocalCounter of first thread: 0
```

In every thread, threadLocalCounter is equal to zero because the value of ThreadLocal is accessible only by a specific thread, so incrementing by one never happens.

As you already know we can't predict the order of the output.

More details

You can think of the ThreadLocal class as a grandpa — it was created in Java 2. But, since time leaves its mark, the appearance of functional programming in Java 8 brought slight changes to our class: the new method withInitial was added.

It creates a new instance of ThreadLocal and sets its initial value by taking Supplier as a parameter:

```
ThreadLocal<Integer> threadLocalA = ThreadLocal.withInitial(() -> 1);
ThreadLocal<List<String>> threadLocalB =
ThreadLocal.withInitial(ArrayList::new);
```

Another commonly used method is a simple remove:

```
threadLocalValue.remove(); // removes value of ThreadLocal for the current
```

thread

To use… or not to use?

There are two main cases:

- storing thread-specific data (something like UserInfo, TransactionInfo)

Imagine you have a web server that receives requests and has many threads. Every thread makes transactions to a database.

It would be convenient to store transaction data in ThreadLocal. Every thread would have its own copy of ThreadLocal with transaction data received from the request.

While processing the request, the thread will get transaction data or modify it if necessary. Transaction data will be stored until your thread receives a new request and updates it.

- working with non-thread-safe classes (creating an instance of the class for each thread)

```
public class MyRunnable implements Runnable {

    // SimpleDateFormat is not thread-safe, so we have separate format instance for each thread
    ThreadLocal<SimpleDateFormat> threadLocal = ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyy/MM/dd HH:mm"));

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " default format = " + threadLocal.get().toPattern());

        // change for specific thread
        threadLocal.set(new SimpleDateFormat());

        System.out.println(Thread.currentThread().getName() + " format = " + threadLocal.get().toPattern());
    }
}
```

Now, let's run it:

```
MyRunnable commonRunnable = new MyRunnable();

Thread t1 = new Thread(commonRunnable, "first thread");
Thread t2 = new Thread(commonRunnable, "second thread");
Thread t3 = new Thread(commonRunnable, "third thread");

// start all threads

// wait for threads to end
```

Our output will be:

```
second thread default format = yyyy/MM/dd HH:mm
third thread default format = yyyy/MM/dd HH:mm
first thread default format = yyyy/MM/dd HH:mm
first thread format = M/d/yy, h:mm a
third thread format = M/d/yy, h:mm a
second thread format = M/d/yy, h:mm a
```

As you can see, every thread's format has changed its value from default to the standard one.

- ThreadLocal value is accessible only by a specific thread.
- Usually, ThreadLocal instances are private static fields (the class associates its state with a thread).
- Use it to store thread-specific data and to work with non-thread-safe classes.

The right values

What values will threadLocal1 and threadLocal2 have after this code is executed?

```
ThreadLocal<Integer> threadLocal1 = ThreadLocal.withInitial(() -> 0);
ThreadLocal<Integer> threadLocal2 = new ThreadLocal<>();

// set
threadLocal1.set(1);
threadLocal2.set(1);

// remove
threadLocal1.remove();
threadLocal2.remove();

//threadLocal1 -> 0
//threadLocal2 -> null
```