ThreadLocal

**private** ThreadLocal threadLocal = **new** ThreadLocal();

Similar to temporary table in oracle

Applicable only to respective sessions.

Other users cannot see your session value.

ThreadLocal in Java is another way to achieve thread-safety apart from writing immutable classes. Thread local can be considered as a scope of access like session scope or request scope. In thread local, you can set any object and this object will be local and global to the specific thread which is accessing this object.

Java ThreadLocal class provides thread-local variables. It enables you to create variables that can only be read and write by the same thread. **If two threads are executing the same code and that code has a reference to a ThreadLocal variable then the two threads can't see the local variable of each other.**

THREADLOCAL

The purpose of ThreadLocal is that the fields do *not* have to be atomic -- there is no need to synchronize their values with centralized memory. They are thread-local and only exist in the threads local memory storage.

They serve **completely** different purposes. ThreadLocal means you don't have to worry about synchronization by granting each thread its own copy of an object. Therefore, an object stored in a ThreadLocal cannot possibly be accessed by more than one thread.

Atomic\* also mean that you don't have to worry about synchronization, but they are specifically meant to be shared across threads.

<https://raw.githubusercontent.com/vsaravanan/java22/master/src/main/java/com/saravanjs/java22/console/multithreading/ThreadLocalExample/ThreadLocalExample.java>

*private final* ThreadLocal<DateFormat> threadLocal =
 *new* ThreadLocal<DateFormat>() {
 @Override
 *protected* DateFormat initialValue() {
 *return new* SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSSSSS");
 }
 };

DateFormat dateFormat = threadLocal.get();

*// Another Example*

*public class* MyRunnable *implements Runnable* {

 *private* ThreadLocal<Integer> threadLocal = *new* ThreadLocal<Integer>();
 Integer myLocal = 0;
 @Override
 *public void* run() {
 System.*out*.println(Thread.*currentThread*().getName() + " threadLocal : " + threadLocal.get() + " myLocal " + myLocal );
 threadLocal.set( (*int*) (Math.*random*() \* 100D) );
 myLocal = (*int*) (Math.*random*() \* 100D) ;

 *try* {
 Thread.*sleep*(1000);
 } *catch* (InterruptedException e) {
 }

 System.*out*.println(Thread.*currentThread*().getName() + " threadLocal : " + threadLocal.get() + " myLocal " + myLocal );
 }
}

*public class* ThreadLocalExample {

 *public static void* main(String[] args) *throws* InterruptedException {
 MyRunnable sharedRunnableInstance = *new* MyRunnable();

 Thread thread1 = *new* Thread(sharedRunnableInstance);
 Thread thread2 = *new* Thread(sharedRunnableInstance);

 thread1.start();
 thread1.join(); *//wait for thread 1 to terminate* Thread.*sleep*(2000);
 thread2.start();
 thread2.join(); *//wait for thread 2 to terminate* }

}

Thread-0 threadLocal : null myLocal 0

Thread-0 threadLocal : 77 myLocal 25

Thread-1 threadLocal : null myLocal 25

Thread-1 threadLocal : 69 myLocal 10

This myLocal preserved the changes
but threadlocal of Thread-1 cannot see any changes in Thread-0
in Thread-1 threadLocal is initialized with null though it is set to 77 in Thread-0