# ThreadLocalRandom

Guide to using ThreadLocalRandom with examples.

**We'll cover the following**





# Overview#

The class `java.util.concurrent.ThreadLocalRandom` is derived from `java.util.Random` and generates random numbers much more efficiently than `java.util.Random` in multithreaded scenarios. Interestingly, `Random` *is thread-safe* and can be used by multiple threads without malfunction, just not efficiently.

To understand why an instance of the `Random` class experiences overhead and contention in concurrent programs, we'll delve into the code for one of the most commonly used methods `nextInt()` of the `Random` class. The code is reproduced verbatim from the Java source code below:

```
    /**
     * Generates the next pseudorandom number. Subclasses should
     * override this, as this is used by all other methods.
     *
     * <p>The general contract of {@code next} is that it return
s an
     * {@code int} value and if the argument {@code bits} is betwe
en
     * {@code 1} and {@code 32} (inclusive), then that many low-or
der
     * bits of the returned value will be (approximately) independ
ently
     * chosen bit values, each of which is (approximately) equally
     * likely to be {@code 0} or {@code 1}. The method {@code nex
t} is
     * implemented by class {@code Random} by atomically updatin
g the seed to
     *  <pre>{@code (seed * 0x5DEECE66DL + 0xBL) & ((1L << 4
8) - 1)}</pre>
     * and returning
     *  <pre>{@code (int)(seed >>> (48 - bits))}.</pre>
     *
     * This is a linear congruential pseudorandom number generato
r, as
     * defined by D. H. Lehmer and described by Donald E. Knuth in
     * <i>The Art of Computer Programming,</i> Volume 2:
     * <i>Seminumerical Algorithms</i>, section 3.2.1.
     *
     * @param  bits random bits
     * @return the next pseudorandom value from this random number
     *         generator's sequence
     * @since  1.1
     */
    protected int next(int bits) {
        long oldseed, nextseed;
        AtomicLong seed = this.seed;
        do {
            oldseed = seed.get();
```

```
            nextseed = (oldseed * multiplier + addend) & mask;
        } while (!seed.compareAndSet(oldseed, nextseed));
        return (int)(nextseed >>> (48 - bits));
    }
```

Examine the code above and realize the `do-while` loop uses the `compareAndSet()` method to atomically set the `seed` variable to a new value in its predicate. Imagine several threads invoking the `next()` method on a shared instance of `Random`, only one thread will successfully exit the loop and the rest will re-execute the loop, until all of them exit one by one. This mechanism to update the `seed` variable is precisely what makes the `Random` class inefficient for highly concurrent programs, when several threads want to generate random numbers in parallel.

The performance issues faced by `Random` are addressed by the `ThreadLocalRandom` class which is isolated in its effects to a single thread. A random number generated by one thread using `ThreadLocalRandom` has no bearing on random numbers generated by other threads, unlike an instance of `Random` that generates random numbers globally. Furthermore, `ThreadLocalRandom` differs from `Random` in that the former doesn't allow setting a seed value unlike the latter. In summary, `ThreadLocalRandom` is more performant than `Random` as it eliminates concurrent access to shared state.

The astute reader would question if maintaining a distinct `Random` object per thread is equivalent to using the `ThreadLocalRandom` class? The `ThreadLocalRandom` class is singleton and uses state held by the `Thread` class to generate random numbers. In particular the `Thread` class houses the following fields for `ThreadLocalRandom` to use for generating random numbers and related book-keeping.

```
class Thread implements Runnable {

    // The following three initially uninitialized fields are excl
usively
    // managed by class java.util.concurrent.ThreadLocalRandom. Th
ese
    // fields are used to build the high-performance PRNGs in the
    // concurrent code, and we can not risk accidental false shari
ng.
    // Hence, the fields are isolated with @Contended.

    /** The current seed for a ThreadLocalRandom */
    @jdk.internal.vm.annotation.Contended("tlr")
    long threadLocalRandomSeed;

    /** Probe hash value; nonzero if threadLocalRandomSeed initial
ized */
    @jdk.internal.vm.annotation.Contended("tlr")
    int threadLocalRandomProbe;

    /** Secondary seed isolated from public ThreadLocalRandom sequ
ence */
    @jdk.internal.vm.annotation.Contended("tlr")
    int threadLocalRandomSecondarySeed;

}
```

Each thread stores the seed itself in the field `threadLocalRandomSeed` . As the seed is not shared among threads anymore, performance improves.

# Usage#

The idiomatic usage for generating random numbers takes the form of `ThreadLocalRandom.current().nextInt()` and is demonstrated in the widget below:

```
import java.util.concurrent.ThreadLocalRandom;

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

        // generate a random boolean value
        System.out.println(ThreadLocalRandom.current().nextBoolean());

        // generate a random int value
        System.out.println(ThreadLocalRandom.current().nextInt());

        // generate a random int between 0 (inclusive) and 500 (exclusive)
        System.out.println(ThreadLocalRandom.current().nextInt(500));

        // generate a random int between 700 (inclusive) and 1900 (exclusive)
        System.out.println(ThreadLocalRandom.current().nextInt(700,1900));

        // generate a random double between 0 (inclusive) and 1 (exclusive)
        System.out.println(ThreadLocalRandom.current().nextDouble());

        // generate a random float value between 0 (inclusive) and 1 (exclusive)
        System.out.println(ThreadLocalRandom.current().nextFloat());

        // generate a Gaussian ("normally") distributed double value with mean 0.0
        // standard deviation 1.0 from this random number generator's sequence
        System.out.println(ThreadLocalRandom.current().nextGaussian());
    }
}
```

# Difference between **`Random`** and **`ThreadLocalRandom`**#

Consider the scenario of a single instance of `Random` class being shared among 5 threads. Our program has each thread generate a random integer ten thousand times. We repeat the same test using the `ThreadLocalRandom` class and time the execution for both scenarios in milliseconds. As expected, the test using the `ThreadLocalRandom` class performs better than the one using the `Random` class instance. Though our test is crude but it still gives us

using the `Random` class instance. Though our test is crude but it still gives us a sense of difference in performance of the two classes.

> Some runs of the program may exhibit a longer execution time for `ThreadLocalRandom` class than the `Random` class. This may occur due to the widget code executing in a shared cloud environment beyond our control. However, if the reader executed the code with all aspects as constants `ThreadLocalRandom` would outperform `Random` when generating random numbers in our text code.

```
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadLocalRandom;

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

        performanceUsingRandom();
        performanceUsingThreadLocalRandom();
    }


    static void performanceUsingThreadLocalRandom() throws Exception {

        ExecutorService es = Executors.newFixedThreadPool(15);

        Runnable task = new Runnable() {
            @Override
            public void run() {

                for (int i = 0; i < 50000; i++) {
                    ThreadLocalRandom.current().nextInt();
                }

            }
        };

        int numThreads = 4;
        Future[] futures = new Future[numThreads];
        long start = System.currentTimeMillis();

        try {
            for (int i = 0; i < numThreads; i++)
                futures[i] = es.submit(task);

            for (int i = 0; i < numThreads; i++)
                futures[i].get();

            long executionTime = System.currentTimeMillis() - start;
            System.out.println("Execution time using ThreadLocalRandom : " + execut

        } finally {
            es.shutdown();
        }
    }

    static void performanceUsingRandom() throws Exception {

        Random random = new Random();
```

```
        ExecutorService es = Executors.newFixedThreadPool(15);


        Runnable task = new Runnable() {
            @Override
            public void run() {

                for (int i = 0; i < 50000; i++){
                    random.nextInt();
                }
            }
        };

        int numThreads = 4;
        Future[] futures = new Future[numThreads];
        long start = System.currentTimeMillis();

        try {
            for (int i = 0; i < numThreads; i++)
                futures[i] = es.submit(task);

            for (int i = 0; i < numThreads; i++)
                futures[i].get();

            long executionTime = System.currentTimeMillis() - start;
            System.out.println("Execution time using Random : " + executionTime + "

        } finally {
            es.shutdown();
        }
    }

}
```




Interviewing soon? We've partnered with Hired so that companies apply to yo
utm_source=educative&utm_medium=lesson&utm_location=CA&utm_campa



Back

ThreadLocal

Next

CountDownLatch