Fix the Trivial Problems

In this lesson, we will begin redesigning the Random library and have a look at the performance issues.

WE'LL COVER THE FOLLOWING

In the previous lesson, we discussed the shortcomings of `System.Random`. In this lesson, we will begin redesigning the `Random` library.

First Attempt to Fixing Random

The first thing we want is two *static methods*, one that gives me an actually-random integer from `0` to `Int32.MaxValue`, and one that gives us a random double from 0.0 (inclusive) to 1.0 (exclusive). Not *pseudo-random*, but indistinguishable from a true random uniform distribution.

Here's an attempt:

```
using CRNG = System.Security.Cryptography.RandomNumberGenerator;
public static class BetterRandom
{
  private static readonly ThreadLocal<CRNG> crng = new ThreadLocal<CRNG>(C
RNG.Create);
  private static readonly ThreadLocal<byte[]> bytes =
    new ThreadLocal<byte[]>(()=>new byte[sizeof(int)]);
  public static int NextInt()
  {
    crng.Value.GetBytes(bytes.Value);
    return BitConverter.ToInt32(bytes.Value, 0) & int.MaxValue;
  }
  public static double NextDouble()
  {
```

```
  {
    while (true)
    {
      long x = NextInt() & 0x001FFFFF;
      x <<= 31;
      x |= (uint)NextInt();
      double n = x;
      const double d = 1L << 52;
      double q = n / d;
      if (q != 1.0)
        return q;
    }
  }
}
```

It's pretty straightforward, but let's take a quick look. First, all the state is thread-local, so we trade a small amount of per-thread overhead and per-call indirection for thread safety, which is probably a good tradeoff. We're going to use the same *four-byte* buffer over and over again. So it is probably worthwhile to cache it and avoid the hit to collection pressure; however, this is just a guess and we would want to verify that with empirical tests.

There are other possible performance problems here, for example:

- Is it worthwhile to generate a few thousand random bytes at once, cache them, and then gradually use them up? or,
- Is the per-call cost of `GetBytes` sufficiently low that this is not an issue?

> We will have to check this empirically, however, note that this course is not about making good performance optimizations. There are lots of places where the code could be a lot faster. We went for clarity rather than performance. It's easier to make a clear program fast than it is to make a fast program clear!

We want only positive integers; clearing the top bit by `and-ing` with `0x7FFFFFF` does that nicely without changing the distribution.

For a random double, we know that doubles have `52 bits` of precision, so we generate a random `52 bit integer` and make that the numerator of a fraction. We want to guarantee that 1.0 is never a possible output. So we reject that possibility; one in every few billion calls we'll do some additional calls to

`NextInt`.

> An alternative and possibly better solution would be to build the double from bits directly, rather than doing an expensive division.

Now that we have a crypto-strength thread-safe random number library, we can build a cheap *pseudo-random* library out of it:

```
public static class Pseudorandom
{
  private readonly static ThreadLocal<Random> prng =
    new ThreadLocal<Random>(() => new Random(BetterRandom.NextInt()));

  public static int NextInt() => prng.Value.Next();
  public static double NextDouble() =>  prng.Value.NextDouble();
}
```

We've solved two of the main problems with `Random`. It gave us highly correlated values when called repeatedly from one thread, and it is not thread-safe. We now have a single `prng` per thread, and it is seeded with a crypto-strength random seed, not the current time.

I think this is already a major improvement, in that 99% of the questions about misuse of `Random` on **StackOverflow** would not exist if this had been the received randomness library in the first place. These problems have been fixed in the core implementation.

Performance Issues

What about performance? The `crypto-RNG` is a lot slower than the `pseudo-RNG`, and there is overhead for the thread-local infrastructure. But most programs would not be observably slowed down by these improvements. For the ones that are slowed down, we can make a cheap, thread-unsafe version if we need to improve performance.

We should always make good tradeoffs; if the cost of eliminating a whole class of bugs is a tiny performance hit, we should make that tradeoff.

> The entire premise of managed languages is that we trade a small amount of performance for improved developer productivity.

That's why we have things like a garbage collector, immutable strings, checked array access, and so on; these all trade a small amount of performance for more safety, robustness, and productivity. And that's why we have unsafe pointers; if you need to abandon some safety to get performance, C# should let you do that. But we should default to safety!

Recent implementations of `System.Random` solve this problem using some similar techniques.

> Remember, that codes in this course are for pedagogy and not intended to be an example of industrial-quality code. You'll notice that for instance, they make only the rarely-used parts of the algorithm thread-safe. That seems like a reasonable tradeoff between performance and safety.

We're off to a good start, but we can do a lot better still. These are trivial problems. There are a lot more problems we could be solving. Two that come to mind are as follows:

- The received `PRNG` is simply not very good in terms of its randomness. There are far better algorithms that produce harder-to-predict outputs with similar performance and a similar amount of state.
- A common use for `PRNG`s is to generate a random-seeming but actually-deterministic sequence to randomize a game. But the received implementation has no way to say "save the current state of the `PRNG` to disk and restore it later", or any such thing.

We are not going to address either of these problems in this course. We are going to take a different approach to improve *how we deal with randomness*. We want to look at the problem at a higher level of abstraction than the low-level details of how the random numbers are generated and what the state of the generator is. To improve the state of the art of dealing with probability in C#, we'll need some more powerful types.

Implementation

The following code implements improvements in the `Random` class:

Program.cs

BetterRandom.cs

Pseudorandom.cs

```
using System;
using System.Threading;

namespace Probability
{
  // A threadsafe, all-static, crypto-randomized wrapper around Random.
  // Still not great, but a slight improvement.
  public static class Pseudorandom
  {
    private readonly static ThreadLocal<Random> prng =
      new ThreadLocal<Random>(() => new Random(BetterRandom.NextInt()));

    public static int NextInt() => prng.Value.Next();
    public static double NextDouble() => prng.Value.NextDouble();
  }
}
```

In the next lesson, we will make some useful classes for common continuous probability distributions.