Disadvantages of System.Random in C#

In this lesson, we introduce the System.Random in C# and highlight some of the problems with it.

Introducing `System.Random`

The C# design team tries hard to make the language a *pit of success*, where the natural way to write programs is also the correct, elegant and performant way. And then `System.Random` comes along; it is almost always wrong, and it is seldom easy to make it right.

Let's start with the obvious problem: the natural way to use it is also the wrong way.

The naive developer thinks "I need a new random dice roll"
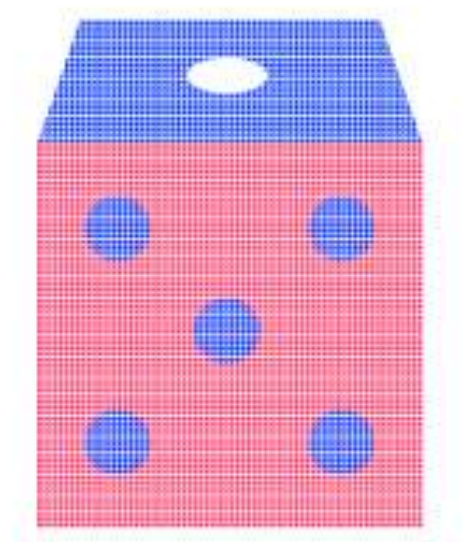
```
void M()
{
   Random r = new Random();
   int x = r.Next(1, 6);
   ...
```

Let's highlight some of the issues with this code.

Problem with `System.Random`

Firstly, every time `M()` is called, we create a `new Random`, but in most of the implementations of `Random` available for the last couple decades, by default, it seeds itself with the current time, and **the granularity of the timer is only a few milliseconds**. Computers run on the nanosecond scale, and so the likelihood that we'll get two `Random`s with the same seed is very high, and therefore it is very likely that successive calls to `M()` produce runs of the same number. You never want to make a new `Random` if you can avoid it; you want to make one once, and then re-use it. But it's a bad idea to stash it away in a static because it's not threading safe!

This problem has been fixed in some versions of the *CLR*. In those versions, a new `Random()` now seeds itself randomly, rather than based on the current time.

Second, the arguments to `Next` are the minimum value produced, inclusive, and the maximum value produced, exclusive! This program produces random numbers drawn from $1, 2, 3, 4, 5$. The correct way to get numbers from 1 to 6 is, of course, `Next(1, 7)`.

The *fundamental* problem here is that *we're working at too low a level of abstraction*. It is not the 1970s anymore when `rand()` was good enough. We have sophisticated problems in statistical modeling and the attendant probabilistic reasoning to solve in modern programming languages. We need to up our game by writing code in the "*business domain*" of probabilities, not the "*mechanism domain*" of calls to methods that return random numbers.

Implementation

Let's have a look at the basic `Random` class in C#:

Program.cs

RandomIsAwful.cs

```csharp
using System;
using System.Threading;

namespace Probability
{
    static class RandomIsAwful
    {
        static Random shared = new Random();
        static string s = "";
        public static void DoIt()
        {
          Console.WriteLine("If this code runs in older versions of C#, the output will be

          for (int i = 0; i < 100; ++i)
          {
              Random random = new Random();
              Console.Write(random.Next(1, 6) + " ");
          }
          Console.WriteLine();

          Console.WriteLine("Similarly, in earlier days this would eventually print all zer

          for (int i = 0; i < 100; ++i)
          {
              new Thread(() => s += shared.Next(1, 6) + " ").Start();
          }

          // Yeah we should wait for those to finish.
          Console.WriteLine(s);

          Console.WriteLine("The real problem though is that this interface is not strong en
        }
    }
}
```

In the next chapter, we will start by simply improving the existing implementation of `Random`, but from that humble beginning, we'll develop a new class library that makes programming with probabilities much more readable, powerful and efficient in C#

readable, powerful and efficient in C#.