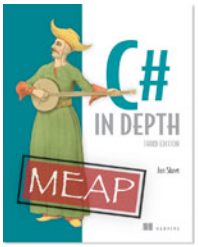

A Quick Sweep through C#

By Jon Skeet, author of *C# in Depth, Third Edition*

Each new version of C# has added significant features to reduce developer angst, but always in a carefully considered way, and with little backward incompatibility. In this greenpaper, based on C# in Depth, Third Edition, *author Jon Skeet gives an overview what C# can do.*

In this greenpaper, I'll let the C# compiler do amazing things without telling you how and barely mentioning the what or the why. I won't explain how things work or try to go one step at a time. Quite the opposite, in fact—the plan is to impress rather than educate.

The evolution of a data type

The example I'll use is contrived—it's designed to pack as many new features into as short a piece of code as possible. It's also clichéd, but at least that makes it familiar. Yes, it's a product/name/price example, the e-commerce alternative to "hello, world." We'll look at how various tasks can be achieved, and how, as we move forward in versions of C#, you can accomplish them more simply and elegantly than before. You won't see any of the benefits of C# 5 until right at the end, but don't worry—that doesn't make it any less important.

The Product type in C# 1

We'll start off with a type representing a product, and then manipulate it. You won't see anything particularly impressive yet—just the encapsulation of a couple of properties. To make life simpler for demonstration purposes, this is also where we'll create a list of predefined products.

Listing 1 shows the type as it might be written in C# 1. We'll then move on to see how the code might be rewritten for each later version. This is the pattern we'll follow for each of the other pieces of code. Given that I'm writing this in 2013, it's likely that you're already familiar with code that uses some of the features I'll introduce, but it's worth looking back so you can see how far the language has come.

Listing 1 The `Product` type (C# 1)

```
using System.Collections;
public class Product
{
    string name;
    public string Name { get { return name; } }

    decimal price;
    public decimal Price { get { return price; } }

    public Product(string name, decimal price)
    {
       this.name = name;
       this.price = price;
    }
```

```
    public static ArrayList GetSampleProducts()
    {
       ArrayList list = new ArrayList();
       list.Add(new Product("West Side Story", 9.99m));
       list.Add(new Product("Assassins", 14.99m));
       list.Add(new Product("Frogs", 13.99m));
       list.Add(new Product("Sweeney Todd", 10.99m));
       return list;
    }

    public override string ToString()
    {
       return string.Format("{0}: {1}", name, price);
    }
}
```

Nothing in listing 1 should be hard to understand—it's just C# 1 code, after all. There are three limitations that it demonstrates, though:

- An `ArrayList` has no compile-time information about what's in it. You could accidentally add a string to the list created in `GetSampleProducts`, and the compiler wouldn't bat an eyelid.
- You've provided public getter properties, which means that if you wanted matching setters, they'd have to be public, too.
- There's a lot of fluff involved in creating the properties and variables—code that complicates the simple task of encapsulating a string and a decimal.

Let's see what C# 2 can do to improve matters.

Strongly typed collections in C# 2

Our first set of changes (shown in the following listing) tackles the first two items listed previously, including the most important change in C# 2: generics. The parts that are new are in bold font.

Listing 2 Strongly typed collections and private setters (C# 2)

```
public class Product
{
   string name;
   public string Name
   {
      get { return name; }
      private set { name = value; }
   }

   decimal price;
   public decimal Price
   {
      get { return price; }
      private set { price = value; }
   }

   public Product(string name, decimal price)
   {
      Name = name; Price = price;
   }

   public static List<Product> GetSampleProducts()
   {
      List<Product> list = new List<Product>();
      list.Add(new Product("West Side Story", 9.99m));
      list.Add(new Product("Assassins", 14.99m));
      list.Add(new Product("Frogs", 13.99m));
      list.Add(new Product("Sweeney Todd", 10.99m)); return list;
   }

   public override string ToString()
   {
```

```
        return string.Format("{0}: {1}", name, price);
    }
}
```

You now have properties with private setters (which you use in the constructor), and it doesn't take a genius to guess that `List<Product>` is telling the compiler that the list contains products. Attempting to add a different type to the list would result in a compiler error, and you also don't need to cast the results when you fetch them from the list.

The changes in C# 2 leave only one of the original three difficulties unanswered, and C# 3 helps out there.

Automatically implemented properties in C# 3

We're starting off with some fairly tame features from C# 3. The automatically implemented properties and simplified initialization shown in the following listing are relatively trivial compared with lambda expressions and the like, but they can make code a lot simpler.

Listing 3 Automatically implemented properties and simpler initialization (C# 3)

```
using System.Collections.Generic;

class Product
{
    public string Name { get; private set; }
    public decimal Price { get; private set; }

    public Product(string name, decimal price)
    {
        Name = name; Price = price;
    }

    Product() {}

    public static List<Product> GetSampleProducts()
    {
        return new List<Product> {
        new Product { Name="West Side Story", Price = 9.99m },
        new Product { Name="Assassins", Price=14.99m },
        new Product { Name="Frogs", Price=13.99m },
        new Product { Name="Sweeney Todd", Price=10.99m} }
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", Name, Price);
    }
}
```

Now the properties don't have any code (or visible variables!) associated with them, and you're building the hardcoded list in a very different way. With no `name` and `price` variables to access, you're forced to use the properties everywhere in the class, improving consistency. You now have a private parameterless constructor for the sake of the new property-based initialization. (This constructor is called for each item before the properties are set.)

In this example, you could've removed the public constructor completely, but then no outside code could've created other product instances.

Named arguments in C# 4

For C# 4, we'll go back to the original code when it comes to the properties and constructor, so that it's fully immutable again. A type with only private setters can't be *publicly* mutated, but it can be clearer if it's not privately mutable either.[1] There's no shortcut for read-only properties, unfortunately, but C# 4 lets you specify argument names for the constructor call, as shown in the following listing, which gives you the clarity of C# 3 initializers without the mutability.

[1] The C# 1 code could've been immutable too—I only left it mutable to simplify the changes for C# 2 and 3.

Listing 4 Named arguments for clear initialization code (C# 4)

```
using System.Collections.Generic;
public class Product
{
   readonly string name;
   public string Name { get { return name; } }

   readonly decimal price;
   public decimal Price { get { return price; } }

   public Product(string name, decimal price)
   {
      this.name = name;
      this.price = price;
   }

   public static List<Product> GetSampleProducts()
   {
      return new List<Product>
      {
         new Product(
             name: "West Side Story", price: 9.99m),
         new Product(
             name: "Assassins", price: 14.99m),
         new Product(
             name: "Frogs", price: 13.99m),
         new Product(
             name: "Sweeney Todd", price: 10.99m)
      };
   }

   public override string ToString()
   {
      return string.Format("{0}: {1}", name, price);
   }
}
```

The benefits of specifying the argument names explicitly are relatively minimal in this particular example, but when a method or constructor has several parameters, it can make the meaning of the code much clearer—particularly if they're of the same type, or if you're passing in `null` for some arguments. You can choose when to use this feature, of course, only specifying the names for arguments when it makes the code easier to understand.

Figure 1 summarizes how the `Product` type has evolved so far. I'll include a similar diagram after each task, so you can see the pattern of how the evolution of C# improves the code.

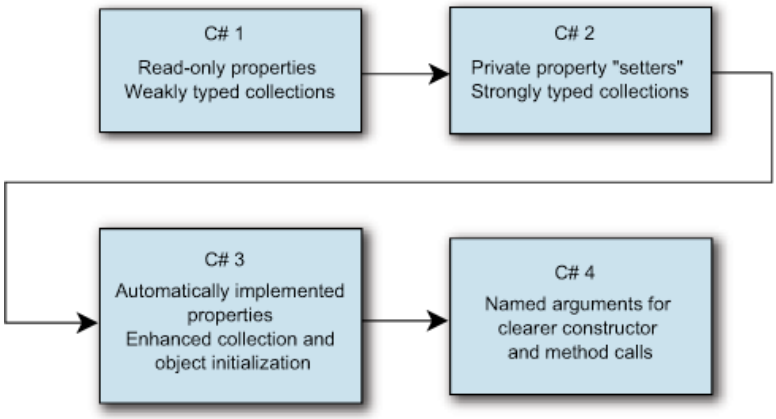

Figure 1 Evolution of the **`Product`** type, showing greater encapsulation, stronger typing, and ease of initialization over time

You'll notice that C# 5 is missing from all of the block diagrams; that's because the main feature of C# 5 (asynchronous functions) is aimed at an area that really hasn't evolved much in terms of language support.

We'll take a peek at .NET platform now.

Dissecting the .NET platform

When it was originally introduced, *.NET* was used as a catchall term for a vast range of technologies coming from Microsoft. For instance, Windows Live ID was called *.NET Passport*, despite there being no clear relationship between that and what you currently know as .NET. Fortunately, things have calmed down somewhat since then. In this section, we'll look at the various parts of .NET.

In several places in this book, I'll refer to three different kinds of features: features of C# as a *language*, features of the *runtime* that provides the "engine," if you will, and features of the .NET *framework libraries*. This book is heavily focused on the language of C#, and I'll generally only discuss runtime and framework features when they relate to features of C# itself. Often features will overlap, but it's important to understand where the boundaries lie.

C#, the language

The language of C# is defined by its specification, which describes the format of C# source code, including both syntax and behavior. It doesn't describe the platform that the compiler output will run on, beyond a few key points where the two interact. For instance, the C# language requires a type called `System.IDisposable`, which contains a method called `Dispose`. These are required in order to define the `using` statement. Likewise, the platform needs to be able to support (in one form or another) both value types and reference types, along with garbage collection.

In theory, any platform that supports the required features could have a C# compiler targeting it. For example, a C# compiler could legitimately produce output in a form other than the *Intermediate Language* (*IL*), which is the typical output at the time of this writing. A runtime could interpret the output of a C# compiler, or convert it all to native code in one step rather than JIT-compiling it. Though these options are relatively uncommon, they do exist in the wild; for example, the Micro Framework uses an interpreter, as can Mono (http://mono-project.net). At the other end of the spectrum, ahead-of-time compilation is used by NGen and by Xamarin.iOS (http://xamarin.com/ios)—a platform for building applications for the iPhone and other iOS devices.

Runtime

The runtime aspect of the .NET platform is the relatively small amount of code that's responsible for making sure that programs written in IL execute according to the *Common Language Infrastructure* (CLI) specification (ECMA-335 and ISO/IEC 23271), partitions I to III. The runtime part of the CLI is called the *Common Language Runtime* (CLR). When I refer to *the CLR* in the rest of the book, I mean Microsoft's implementation.

Some elements of the C# language never appear at the runtime level, but others cross the divide. For instance, enumerators aren't defined at a runtime level, and neither is any particular meaning attached to the `IDisposable` interface, but arrays and delegates are important to the runtime.

Framework libraries

Libraries provide code that's available to your programs. The framework libraries in .NET are largely built as IL themselves, with native code used only where necessary.

This is a mark of the strength of the runtime: your own code isn't expected to be a second-class citizen—it can provide the same kind of power and performance as the libraries it utilizes. The amount of code in the libraries is much greater than that of the runtime, in the same way that there's much more to a car than the engine.

The framework libraries are partially standardized. Partition IV of the CLI specification provides a number of different profiles (*compact* and *kernel*) and libraries. Partition IV comes in two parts—a general textual description of the libraries identifying, among other things, which libraries are required within which profiles, and another part containing the details of the libraries themselves in XML format. This is the same form of documentation produced when you use XML comments within C#.

There's much within .NET that's *not* within the base libraries. If you write a program that *only* uses libraries from the specification, and uses them correctly, you should find that your code works flawlessly on any implementation—Mono, .NET, or anything else. But in practice, almost any program of any size will use libraries that aren't standardized—Windows Forms or ASP.NET, for instance. The Mono project has its own libraries that aren't part of .NET, such as GTK#, and it implements many of the nonstandardized libraries.

The term *.NET* refers to the combination of the runtime and libraries provided by Microsoft, and it also includes compilers for C# and VB.NET. It can be seen as a whole *development platform* built on top of Windows. Each aspect of .NET is versioned separately, which can be a source of confusion. Appendix gives a quick rundown of which version of what came out when and with what features.

If that's all clear, I have one last bit of housekeeping to go through.

Summary

In this article, I've shown (but not explained) some of the features that are tackled in *C# in Depth, Third Edition*. There are plenty more that I haven't shown here, and many of the features you've seen so far have further subfeatures associated with them. Hopefully what you've seen here has whetted your appetite for the book.

Here are some other Manning titles you might be interested in:

Dependency Injection in .NET
Mark Seemann

Windows Phone 7 in Action
Timothy Binkley-Jones, Massimo Perga and Michael Sync

Real-World Functional Programming
Tomas Petricek

Last updated: July 6, 2013