Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

C# 4, ASP.NET 4, & WPF with Visual Studio 2010

Jump Start

Christian Nagel, Bill Evjen, Rod Stephens

Table of Contents

C# 4, ASP.NET 4, & WPF with Visual Studio 2010 Jump Start

Christian Nagel
Bill Evjen
Jay Glynn
Karli Watson
Morgan Skinner
Scott Hanselman
Devin Rader
Rod Stephens

Wiley Publishing, Inc.

Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional C# 4 and .NET 4

Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner

Covariance and Contra-variance

Previous to .NET 4, generic interfaces were invariant. .NET 4 adds an important extension for generic interfaces and generic delegates with covariance and contra-variance. Covariance and contra-variance are about the conversion of types with argument and return types. For example, can you pass a `Rectangle` to a method that requests a `Shape`? Let's get into examples to see the advantages of these extensions.

With .NET, parameter types are covariant. Assume you have the classes `Shape` and `Rectangle`, and `Rectangle` derives from the `Shape` base class. The `Display()` method is declared to accept an object of the `Shape` type as its parameter:

```
public void Display(Shape o) { }
```

Now you can pass any object that derives from the `Shape` base class. Because `Rectangle` derives from `Shape`, a `Rectangle` fulfills all the requirements of a `Shape` and the compiler accepts this method call:

```
Rectangle r = new Rectangle { Width= 5, Height=2.5};
Display(r);
```

Return types of methods are contra-variant. When a method returns a `Shape` it is not possible to assign it to a `Rectangle` because a `Shape` is not necessarily always a `Rectangle`. The opposite is possible. If a method returns a `Rectangle` as the `GetRectangle()` method,

```
public Rectangle GetRectangle();
```

the result can be assigned to a `Shape`.

```
Shape s = GetRectangle();
```

Before version 4 of the .NET Framework, this behavior was not possible with generics. With C# 4, the language is extended to support covariance and contra-variance with generic interfaces and generic delegates. Let's start by defining a `Shape` base class and a `Rectangle` class:

Available for download on Wrox.com

```
public class Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public override string ToString()
    {
        return String.Format("Width: {0}, Height: {1}", Width, Height);
    }
}
```

Pro C# 4 9780470502259 code snippet Variance/Shape.cs

```
public class Rectangle: Shape
{
}
```

Pro C# 4 9780470502259 code snippet Variance/Rectangle.cs

Covariance with Generic Interfaces

A generic interface is covariant if the generic type is annotated with the `out` keyword. This also means that type `T` is allowed only with return types. The interface `IIndex` is covariant with type `T` and returns this type from a read-only indexer:

Available for download on Wrox.com

```
public interface IIndex<out T>
{
    T this[int index] { get; }
    int Count { get; }
}
```

Pro C# 4 9780470502259 code snippet Variance/IIndex.cs

If a read-write indexer is used with the IIndex interface, the generic type T is passed to the method and also retrieved from the method. This is not possible with covariance — the generic type must be defined as invariant. Defining the type as invariant is done without `out` *and* `in` *annotations.*

The `IIndex<T>` interface is implemented with the `RectangleCollection` class. `RectangleCollection` defines `Rectangle` for generic type `T`:

Available for download on Wrox.com

```
public class RectangleCollection: IIndex<Rectangle>
{
    private Rectangle[] data = new Rectangle[3]
    {
        new Rectangle { Height=2, Width=5},
        new Rectangle { Height=3, Width=7},
        new Rectangle { Height=4.5, Width=2.9}
    };

    public static RectangleCollection GetRectangles()
    {
        return new RectangleCollection();
    }

    public Rectangle this[int index]
    {
        get
        {
            if (index < 0 || index > data.Length)
                throw new ArgumentOutOfRangeException("index");
            return data[index];
        }
    }
    public int Count
    {
        get
        {
            return data.Length;
        }
    }
}
```

Pro C# 4 9780470502259 code snippet Variance/RectangleCollection.cs

The `RectangleCollection.GetRectangles()` method returns a `RectangleCollection` that implements the `IIndex<Rectangle>` interface, so you can assign the return value to a variable *rectangle* of the `IIndex<Rectangle>` type. Because the interface is covariant, it is also possible to assign the returned value to a variable of `IIndex<Shape>`. `Shape` does not need anything more than a `Rectangle` has to offer. Using the `shapes` variable, the indexer from the interface and the `Count` property are used within the `for` loop:

Available for download on Wrox.com

```
        static void Main()
        {
            IIndex<Rectangle> rectangles = RectangleCollection.GetRectangles(
);
            IIndex<Shape> shapes = rectangles;

            for (int i = 0; i < shapes.Count; i++)
            {
                Console.WriteLine(shapes[i]);
            }
        }
```

Pro C# 4 9780470502259 code snippet Variance/Program.cs

Contra-Variance with Generic Interfaces

A generic interface is contra-variant if the generic type is annotated with the `in` keyword. This way the interface is only allowed to use generic type `T` as input to its methods:

Available for download on Wrox.com

```
public interface IDisplay<in T>
{
    void Show(T item);
}
```

Pro C# 4 9780470502259 code snippet Variance/IDisplay.cs

YOU CAN DOWNLOAD THE CODE FOUND IN THIS SECTION. VISIT **WROX.COM** AND SEARCH FOR ISBN 9780470502259

The `ShapeDisplay` class implements `IDisplay<Shape>` and uses a `Shape` object as an input parameter:

```
public class ShapeDisplay: IDisplay<Shape>
{
    public void Show(Shape s)
    {
                                                        Console.WriteLine("
{0} Width: {1}, Height: {2}", s.GetType().Name,
                              s.Width, s.Height);
    }
}
```

Pro C# 4 9780470502259 code snippet Variance/ShapeDisplay.cs

Creating a new instance of `ShapeDisplay` returns `IDisplay<Shape>`, which is assigned to the `shapeDisplay` variable. Because `IDisplay<T>` is contra-variant, it is possible to assign the result to `IDisplay<Rectangle>` where `Rectangle` derives from `Shape`. This time the methods of the interface only define the generic type as input, and `Rectangle` fulfills all the requirements of a `Shape`:

Available for download on Wrox.com

```
static void Main()
{
    //...

    IDisplay<Shape> shapeDisplay = new ShapeDisplay();
    IDisplay<Rectangle> rectangleDisplay = shapeDisplay;
    rectangleDisplay.Show(rectangles[0]);

}
```

Pro C# 4 9780470502259 code snippet Variance/Program.cs

Tuples

Arrays combine objects of the same type; tuples can combine objects of different types. Tuples have the origin in functional programming languages such as F# where they are used often. With .NET 4, tuples are available with the .NET Framework for all .NET languages.

.NET 4 defines eight generic `Tuple` classes and one static `Tuple` class that act as a factory of tuples. The different generic `Tuple` classes are here for supporting a different number of elements; e.g., `Tuple<T1>` contains one element, `Tuple<T1, T2>` contains two elements, and so on.

The method `Divide()` demonstrates returning a tuple with two members — `Tuple<int, int>`. The parameters of the generic class define the types of the members, which are both integers. The tuple is created with the static `Create()` method of the static `Tuple` class. Again, the generic parameters of the `Create()` method define the type of tuple that is instantiated. The newly created tuple is initialized with the `result` and `reminder` variables to return the result of the division:

Available for download on Wrox.com

```
public static Tuple<int, int> Divide(int dividend, int divisor)
{
    int result = dividend / divisor;
    int reminder = dividend % divisor;

    return Tuple.Create<int, int>(result, reminder);
}
```

Pro C# 4 9780470502259 code snippet TuplesSample/Program.cs

The following code shows invoking the `Divide()` method. The items of the tuple can be accessed with the properties `Item1` and `Item2`:

```
var result = Divide(5, 2);
Console.WriteLine("result of division: {0}, reminder: {1}",
```

```
                result.Item1, result.Item2);
```

In case you have more than eight items that should be included in a tuple, you can use the `Tuple` class definition with eight parameters. The last template parameter is named `TRest` to indicate that you must pass a tuple itself. That way you can create tuples with any number of parameters.

To demonstrate this functionality:

```
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
```

Here, the last template parameter is a tuple type itself, so you can create a tuple with any number of items:

```
            var tuple = Tuple.Create<string, string, string, int, int, int, d
ouble,
                Tuple<int, int>>(
                    "Stephanie", "Alina", "Nagel", 2009, 6, 2, 1.37,
                    Tuple.Create<int, int>(52, 3490));
```

The Dynamic Type

The `dynamic` type allows you to write code that will bypass compile time type checking. The compiler will assume that whatever operation is defined for an object of type `dynamic` is valid. If that operation isn’t valid, the error won’t be detected until runtime. This is shown in the following example:

```
class Program
{
    static void Main(string[] args)
    {
        var staticPerson = new Person();
        dynamic dynamicPerson = new Person();
        staticPerson.GetFullName("John", "Smith");
        dynamicPerson.GetFullName("John", "Smith");
    }
}

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string GetFullName()
    {
```

```
        return string.Concat(FirstName, " ", LastName);
    }
}
```

This example will not compile because of the call to `staticPerson.GetFullName()`. There isn't a method on the `Person` object that takes two parameters, so the compiler raises the error. If that line of code were to be commented out, the example would compile. If executed, a runtime error would occur. The exception that is raised is `RuntimeBinderException`. The `RuntimeBinder` is the object in the runtime that evaluates the call to see if `Person` really does support the method that was called.

Unlike the `var` keyword, an object that is defined as *dynamic* can change type during runtime. Remember, when the `var` keyword is used, the determination of the object's type is delayed. Once the type is defined, it can't be changed. Not only can you change the type of a dynamic object, you can change it many times. This differs from casting an object from one type to another. When you cast an object you are creating a new object with a different but compatible type. For example, you cannot cast an `int` to a `Person` object. In the following example, you can see that if the object is a dynamic object, you can change it from `int` to `Person`:

Available for download on Wrox.com

```
dynamic dyn;

dyn = 100;
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);

dyn = "This is a string";
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);

dyn = new Person() { FirstName = "Bugs", LastName = "Bunny" };
Console.WriteLine(dyn.GetType());
Console.WriteLine("{0} {1}", dyn.FirstName, dyn.LastName);
```

Pro C# 4 9780470502259 code snippet Dynamic\Program.cs

Executing this code would show that the `dyn` object actually changes type from `System.Int32` to `System.String` to `Person`. If `dyn` had been declared as an `int` or `string`, the code would not have compiled.

There are a couple of limitations to the `dynamic` type. A dynamic object does not support extension methods. Anonymous functions (Lambda expressions) also cannot be used as parameters to a dynamic method call, thus LINQ does not work well with dynamic objects. Most LINQ calls are extension methods and Lambda expressions are used as arguments to those extension methods.

Dynamic Behind the Scenes

So what's going on behind the scenes to make this happen? C# is still a statically typed language. That hasn't changed. Take a look at the IL (Intermediate Language) that's generated when the `dynamic` type is used.

First, this is the example C# code that you're looking at:

```
using System;

namespace DeCompile
{
    class Program
    {
        static void Main(string[] args)
        {
            StaticClass staticObject = new StaticClass();
            DynamicClass dynamicObject = new DynamicClass();
            Console.WriteLine(staticObject.IntValue);
            Console.WriteLine(dynamicObject.DynValue);
            Console.ReadLine();
        }
    }

    class StaticClass
    {
        public int IntValue = 100;
    }
```

```
    class DynamicClass
    {
        public dynamic DynValue = 100;
    }
}
```

You have two classes, `StaticClass` and `DynamicClass`. `StaticClass` has a single field that returns an `int`. `DynamicClass` has a single field that returns a `dynamic` object. The `Main` method just creates these objects and prints out the value that the methods return. Simple enough.

Now comment out the references to the `DynamicClass` in `Main` like this:

```
static void Main(string[] args)
{
    StaticClass staticObject = new StaticClass();
    //DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

Using the `ildasm` tool, you can look at the IL that is generated for the `Main` method:

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       26 (0x1a)
  .maxstack  1
  .locals init ([0] class DeCompile.StaticClass staticObject)
  IL_0000:  nop
  IL_0001:  newobj     instance void DeCompile.StaticClass::.ctor()
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  ldfld      int32 DeCompile.StaticClass::IntValue
  IL_000d:  call       void [mscorlib]System.Console::WriteLine(int32)
  IL_0012:  nop
  IL_0013:  call       string [mscorlib]System.Console::ReadLine()
  IL_0018:  pop
  IL_0019:  ret
} // end of method Program::Main
```

Without going into the details of IL but just looking at this section of code, you can still pretty much tell what's going on. Line 0001, the `StaticClass` constructor, is called. Line 0008 calls the `IntValue` field of `StaticClass`. The next line writes out the value.

Now comment out the `StaticClass` references and uncomment the `DynamicClass` references:

```
static void Main(string[] args)
{
    //StaticClass staticObject = new StaticClass();
    DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

Compile the application again and this is what gets generated:

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       121 (0x79)
  .maxstack  9
  .locals init ([0] class DeCompile.DynamicClass dynamicObject,
           [1] class [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpA
rgumentInfo[]
                  CS$0$0000)
  IL_0000:  nop
  IL_0001:  newobj     instance void DeCompile.DynamicClass::.ctor()
  IL_0006:  stloc.0
  IL_0007:  ldsfld     class [System.Core]System.Runtime.CompilerServices.Cal
lSite'1
                       <class [mscorlib]
System.Action'3<class
[System.Core]System.Runtime.CompilerServices.CallSite,class [mscorlib]
System.Type,object>> DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1
'
  IL_000c:  brtrue.s   IL_004d
  IL_000e:  ldc.i4.0
  IL_000f:  ldstr "WriteLine"
  IL_0014:  ldtoken    DeCompile.Program
  IL_0019:  call       class [mscorlib]System.Type [mscorlib]System.Type::Get
TypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_001e:  ldnull
  IL_001f:  ldc.i4.2
  IL_0020:  newarr     [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CShar
pArgumentInfo
  IL_0025:  stloc.1
  IL_0026:  ldloc.1
  IL_0027:  ldc.i4.0
  IL_0028:  ldc.i4.s   33
  IL_002a:  ldnull
  IL_002b:  newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.Runti
meBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.Runti
meBinder
```

```
.CSharpArgumentInfoFlags,

string)
  IL_0030:  stelem.ref
  IL_0031:  ldloc.1
  IL_0032:  ldc.i4.1
  IL_0033:  ldc.i4.0
  IL_0034:  ldnull
  IL_0035:  newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.Runti
meBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.Runti
meBinder
.CSharpArgumentInfoFlags,

string)
  IL_003a:  stelem.ref
  IL_003b:  ldloc.1
  IL_003c:  newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.Runti
meBinder
.CSharpInvokeMemberBinder::.ctor(valuetype Microsoft.CSharp]Microsoft.CSharp
.RuntimeBinder.CSharpCallFlags,

string,

class [mscorlib]System.Type,

class [mscorlib]System.Collections.Generic.IEnumerable'1
<class [mscorlib]System.Type>,

class [mscorlib]System.Collections.Generic.IEnumerable'1
<class [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo>)
  IL_0041:  call       class [System.Core]System.Runtime.CompilerServices.Cal
lSite'1
<!0> class [System.Core]System.Runtime.CompilerServices.CallSite'1
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Create(class [System.Core]System.Runtim
e.CompilerServices
                          .CallSiteBinder)
  IL_0046:  stsfld     class [System.Core]System.Runtime.CompilerServices.Cal
lSite'1
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>> DeCompile.Program/'<Main>o__SiteContaine
r0'::'<>p__Site1'
  IL_004b:  br.s       IL_004d
  IL_004d:  ldsfld     class [System.Core]System.Runtime.CompilerServices.Cal
lSite'1
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>> DeCompile.Program/'<Main>o__SiteContaine
r0'::'<>p__Site1'
  IL_0052:  ldfld      !0 class [System.Core]System.Runtime.CompilerServices.
CallSite'1
<class [mscorlib]System.Action'3
```

```
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Target
  IL_0057:  ldsfld     class [System.Core]System.Runtime.CompilerServices.Cal
lSite'1
<class [mscorlib]System.Action'3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>> DeCompile.Program/'<Main>o__SiteContaine
r0'::'<>p__Site1'
  IL_005c:  ldtoken    [mscorlib]System.Console
  IL_0061:  call       class [mscorlib]System.Type [mscorlib]System.Type::Get
TypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_0066:  ldloc.0
  IL_0067:  ldfld      object DeCompile.DynamicClass::DynValue
  IL_006c:  callvirt   instance void class [mscorlib]System.Action'3
             <class [System.Core]System.Runtime.CompilerServices.CallSite, cl
ass
             [mscorlib]System.Type,object>::Invoke(!0,!1,!2)
  IL_0071:  nop
  IL_0072:  call       string [mscorlib]System.Console::ReadLine()
  IL_0077:  pop
  IL_0078:  ret
} // end of method Program::Main
```

So it's safe to say that the C# compiler is doing a little extra work to support the dynamic type. Looking at the generated code, you can see references to `System.Runtime.CompilerServices.CallSite` and `System.Runtime.CompilerServices.CallSiteBinder`.

The `CallSite` is a type that handles the lookup at runtime. When a call is made on a dynamic object at runtime, something has to go and look at that object to see if the member really exists. The call site caches this information so the lookup doesn't have to be performed repeatedly. Without this process, performance in looping structures would be questionable.

After the `CallSite` does the member lookup, the `CallSiteBinder` is invoked. It takes the information from the call site and generates an expression tree representing the operation the binder is bound to.

There is obviously a lot going on here. Great care has been taken to optimize what would appear to be a very complex operation. It should be obvious that while using the `dynamic` type can be useful, it does come with a price.

Code Contracts

Design-by-contracts is an idea from the Eiffel programming language. Now .NET 4 includes classes for static and runtime checks of code within the namespace `System.Diagnostics.Contracts` that can be used by all .NET languages.

With this functionality you can define preconditions, postconditions, and invariants within a method. The preconditions lists what requirements the parameters must fulfill, the postconditions define the requirements on returned data, and the invariants define the requirements of variables within the method itself.

Contract information can be compiled both into the debug and the release code. It is also possible to define a separate contract assembly, and many checks can also be made statically without running the application. You can also define contracts on interfaces that cause the implementations of the interface to fulfill the contracts. Contract tools can rewrite the assembly to inject contract checks within the code for runtime checks, check the contracts during compile time, and add contract information to the generated XML documentation.

The following figure shows the project properties for the code contracts in Visual Studio 2010. Here, you can define what level of runtime checking should be done, indicate if assert dialogs should be opened on contract failures, and configure static checking. Setting the Perform Runtime Contract Checking to Full defines the symbol `CONTRACTS_FULL`. Because many of the contract methods are annotated with the attribute `[Conditional("CONTRACTS_FULL")]`, all runtime checks are only done with this setting.