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

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
Wiley Publishing, Inc.

CONTENTS

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

PROFESSIONAL

C# 4 and .NET 4

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

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

```
        public class ShapeDisplay: IDisplay<Shape>
```

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

```
{
    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, double,
    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.
CSharpArgumentInfo[]
                  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.CallSite`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::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_001e:  ldnull
  IL_001f:  ldc.i4.2
  IL_0020:  newarr     [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.
CSharpArgumentInfo
  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.
RuntimeBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.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.
RuntimeBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfoFlags,

string)
  IL_003a:  stelem.ref
  IL_003b:  ldloc.1
  IL_003c:  newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.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.CallSite`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.Runtime.
CompilerServices
                           .CallSiteBinder)
  IL_0046:  stsfld     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>> DeCompile.Program/'<Main>o__
SiteContainer0'::'<>p__Site1'
  IL_004b:  br.s       IL_004d
  IL_004d:  ldsfld     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>> DeCompile.Program/'<Main>o__
SiteContainer0'::'<>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.CallSite`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_005c:  ldtoken    [mscorlib]System.Console
  IL_0061:  call       class [mscorlib]System.Type [mscorlib]System.
Type::GetTypeFromHandle
(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, class
            [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.

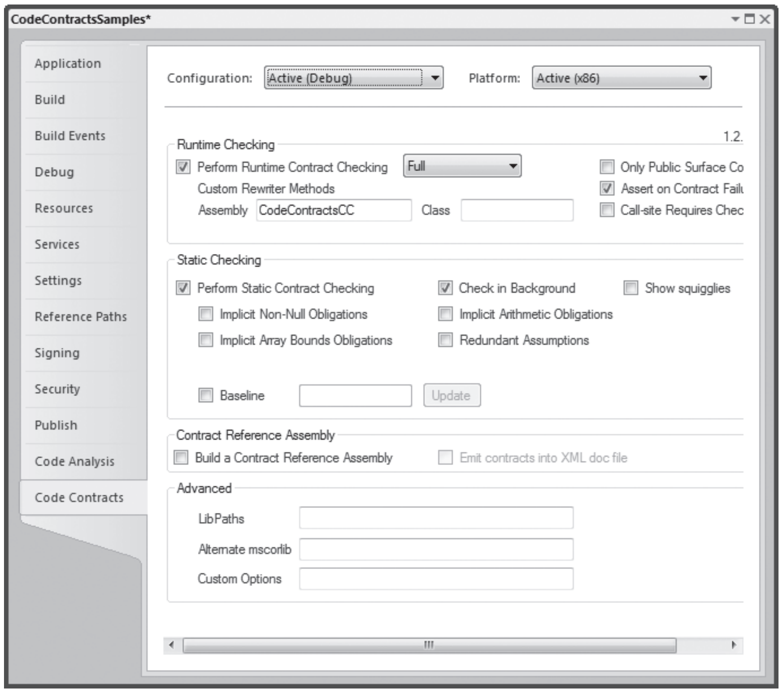
CODE CONTRACTS

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.


CodeContractsSamples*

Application
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing
Security
Publish
Code Analysis
Code Contracts

Configuration: Active (Debug)   Platform: Active (x86)

Runtime Checking   1.2.
[✓] Perform Runtime Contract Checking   Full
[ ] Only Public Surface Co
[✓] Assert on Contract Failu
[ ] Call-site Requires Chec
Custom Rewriter Methods
Assembly CodeContractsCC   Class

Static Checking
[✓] Perform Static Contract Checking
[✓] Check in Background
[ ] Show squigglies
[ ] Implicit Non-Null Obligations
[ ] Implicit Arithmetic Obligations
[ ] Implicit Array Bounds Obligations
[ ] Redundant Assumptions
[ ] Baseline   Update

Contract Reference Assembly
[ ] Build a Contract Reference Assembly
[ ] Emit contracts into XML doc file

Advanced
LibPaths
Alternate mscorlib
Custom Options

To work with code contracts you can use classes that are available with .NET 4 in the namespace `System.Diagnostics.Contracts`. *However, there's no tool included with Visual Studio 2010. You need to download an extension to Visual Studio from Microsoft DevLabs:* `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`. *For static analysis with this tool, the Visual Studio Team System is required; for runtime analysis, Visual Studio Standard edition is enough.*

Code contracts are defined with the `Contract` class. All contract requirements that you define in a method, no matter if they are preconditions or postconditions, must be placed at the beginning of the method. You can also assign a global event handler to the event `ContractFailed` that is invoked for every failed contract during runtime. Invoking `SetHandled()` with the

`ContractFailedEventArgs` parameter `e` stops the standard behavior of failures that would throw an exception.

Available for download on Wrox.com

```
        Contract.ContractFailed += (sender, e) =>
        {
            Console.WriteLine(e.Message);
            e.SetHandled();
        };
```

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

Preconditions

Preconditions check the parameters that are passed to a method. `Requires()` and `Requires<TException>()` are preconditions that can be defined with the `Contract` class. With the `Requires()` method, a Boolean value must be passed, and an optional message string with the second parameter that is shown when the condition does not succeed. The following sample requires that the argument *`min`* be lower than or equal to the argument *`max`*.

Available for download on Wrox.com

```
static void MinMax(int min, int max)
{
     Contract.Requires(min <= max);
    //...
}
```

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

The following contract throws an `ArgumentNullException` if the argument `o` is null. The exception is not thrown if an event handler that sets the `ContractFailed` event to `handled`. Also, if the Assert on Contract Failure is configured, `Trace.Assert()` is done to stop the program instead of throwing the exception defined.

```
static void Preconditions(object o)
{
    Contract.Requires<ArgumentNullException>(o != null,
        "Preconditions, o may not be null");
   //...
```

`Requires<TException>()` is not annotated with the attribute `[Conditional("CONTRACTS_FULL")]`, and it also doesn't have a condition on the DEBUG symbol, so this runtime check is done in any case. `Requires<TException>()` throws the defined exception if the condition is not fulfilled.

With a lot of legacy code, arguments are often checked with `if` statements and throw an exception if a condition is not fulfilled. With code contracts, it is not necessary to rewrite the verification; just add one line of code:

```
static void PrecondtionsWithLegacyCode(object o)
{
    if (o == null) throw new ArgumentNullException("o");
   Contract.EndContractBlock();
```

The `EndContractBlock()` defines that the preceding code should be handled as a contract. If other contract statements are used as well, the `EndContractBlock()` is not necessary.

For checking collections that are used as arguments, the `Contract` class offers `Exists()` and `ForAll()` methods. `ForAll()` checks every item in the collection if the condition succeeds. In the example, it is checked if every item in the collection has a value smaller than 12. With the `Exists()` method, it is checked if any one element in the collection succeeds the condition.

```
static void ArrayTest(int[] data)
{
   Contract.Requires(Contract.ForAll(data, i => i < 12));
```

Both the methods `Exists()` and `ForAll()` have an overload where you can pass two integers, *`fromInclusive`* and *`toExclusive`*, instead of `IEnumerable<T>`. A range from the numbers (excluding *`toExclusive`*) is passed to the delegate `Predicate<int>` defined with the third parameter. `Exists()` and `ForAll()` can be used with preconditions, postconditions, and also invariants.

Postconditions

Postconditions define guarantees about shared data and return values after the method has completed. Although they define some guarantees on return values, they must be written at the beginning of a method; all contract requirements must be at the beginning of the method.

`Ensures()` and `EnsuresOnThrow<TException>()` are postconditions. The following contract ensures that the variable `sharedState` is lower than 6 at the end of the method. The value can change in between.

Available for download on Wrox.com

```
private static int sharedState = 5;
static void Postcondition()
{
    Contract.Ensures(sharedState < 6);
    sharedState = 9;
    Console.WriteLine("change sharedState invariant {0}", sharedState);
    sharedState = 3;
    Console.WriteLine("before returning change it to a valid value {0}",
                      sharedState);
}
```

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

With `EnsuresOnThrow<TException>()`, it is guaranteed that a shared state succeeds a condition if a specified exception is thrown.

To guarantee a return value, the special value `Result<T>` can be used with an `Ensures()` contract. Here, the result is of type `int` as is also defined with the generic type `T` for the `Result()` method. The `Ensures()` contract guarantees that the *`return`* value is lower than 6.

```
static int ReturnValue()
{
    Contract.Ensures(Contract.Result<int>() < 6);
    return 3;
}
```

You can also compare a value to an old value. This is done with the `OldValue<T>()` method that returns the original value on method entry for the variable passed. The following ensures that the contract defines that the result returned (`Contract.Result<int>()`) is larger than the old value from the argument *x* (`Contract.OldValue<int>(x)`).

```
static int ReturnLargerThanInput(int x)
{
    Contract.Ensures(Contract.Result<int>() > Contract.OldValue<int>(x));
    return x + 3;
}
```

If a method returns values with the `out` modifier instead of just with the `return` statement, conditions can be defined with `ValueAtReturn()`. The following contract defines that the *x* variable must be larger than 5 and smaller than 20 on return, and with the *y* variable modulo 5 must equal 0 on return.

```
static void OutParameters(out int x, out int y)
{
    Contract.Ensures(Contract.ValueAtReturn<int>(out x) > 5 &&
                     Contract.ValueAtReturn<int>(out x) < 20);
    Contract.Ensures(Contract.ValueAtReturn<int>(out y) % 5 == 0);
    x = 8;
    y = 10;
}
```

Invariants

Invariants define contracts for variables during the method lifetime. `Contract.Requires()` defines input requirements, `Contract.Ensures()` defines requirements on method end. `Contract.Invariant()` defines conditions that must succeed during the whole lifetime of the method.

Available for download on Wrox.com

```
static void Invariant(ref int x)
{
    Contract.Invariant(x > 5);
    x = 3;
    Console.WriteLine("invariant value: {0}", x);
    x = 9;
}
```

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

Contracts for Interfaces

With interfaces you can define methods, properties, and events that a class that derives from the interface must implement. With the interface declaration you cannot define how the interface must be implemented. Now this is possible using code contracts.

Take a look at the following interface. The interface `IPerson` defines `FirstName`, `LastName`, and `Age` properties, and the method `ChangeName()`. What's special with this interface is just the attribute `ContractClass`. This attribute is applied to the interface `IPerson` and defines that the `PersonContract` class is used as the code contract for this interface.

Available for download on Wrox.com

```
[ContractClass(typeof(PersonContract))]
public interface IPerson
{
    string FirstName { get; set; }
```

```
        string LastName { get; set; }
        int Age { get; set; }
        void ChangeName(string firstName, string lastName);
    }
```

Pro C# 4 9780470502259 code snippet CodeContractsSamples/IPerson.cs

The class `PersonContract` implements the interface `IPerson` and defines code contracts for all the members. The attribute `PureAttribute` means that the method or property may not change state of a class instance. This is defined with the `get` accessors of the properties but can also be defined with all methods that are not allowed to change state. The `FirstName` and `LastName` `get` accessors also define that the result must be a string with `Contract.Result()`. The `get` accessor of the `Age` property defines a postcondition and ensures that the returned value is between 0 and 120. The `set` accessor of the `FirstName` and `LastName` properties requires that the value passed is not null. The `set` accessor of the `Age` property defines a precondition that the passed value is between 0 and 120.

Available for download on Wrox.com

```
    [ContractClassFor(typeof(IPerson))]
    public sealed class PersonContract : IPerson
    {
        string IPerson.FirstName
        {
            [Pure] get { return Contract.Result<String>(); }
            set { Contract.Requires(value != null); }
        }
        string IPerson.LastName
        {
            [Pure] get { return Contract.Result<String>(); }
            set { Contract.Requires(value != null); }
        }
        int IPerson.Age
        {
            [Pure]
            get
            {
                Contract.Ensures(Contract.Result<int>() >= 0 &&
                                 Contract.Result<int>() < 121);
                return Contract.Result<int>();
            }
            set
            {
                Contract.Requires(value >= 0 && value < 121);
            }
        }
        void IPerson.ChangeName(string firstName, string lastName)
        {
            Contract.Requires(firstName != null);
            Contract.Requires(lastName != null);
        }
    }
```

Pro C# 4 9780470502259 code snippet CodeContractsSamples/PersonContract.cs

Now a class implementing the `IPerson` interface must fulfill all the contract requirements. The class `Person` is a simple implementation of the interface that fulfills the contract.

Available for download on Wrox.com

```
public class Person : IPerson
{
    public Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public int Age { get; set; }


    public void ChangeName(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

Pro C# 4 9780470502259 code snippet CodeContractsSamples/Person.cs

When using the class `Person`, the contract must also be fulfilled. For example, it's not allowed to assign null to a property:

Available for download on Wrox.com

```
var p = new Person { FirstName = "Tom", LastName = null }; // contract error
```

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

It's also not allowed to assign an invalid value to the `Age` property:

```
 var p = new Person { FirstName = "Tom", LastName = "Turbo" };
p.Age = 133;  // contract error
```

TASKS

.NET 4 includes the new namespace `System.Threading.Tasks`, which contains classes to abstract threading functionality. Behind the scenes, `ThreadPool` is used. A *task* represents some unit of work that should be done. This unit of work can run in a separate thread; and it is also possible to start a task in a synchronized manner, which results in a wait for the calling thread. With tasks, you have an abstraction layer but also a lot of control over the underlying threads.

Tasks allow much more flexibility in organizing the work you need to do. For example, you can define continuation work — what should be done after a task is complete. This can be differentiated whether the task was successful or not. Also, you can organize tasks in a hierarchy. For example, a parent task can create new children tasks. This can create a dependency, so that canceling a parent task also cancels its child tasks.

Starting Tasks

To start a task, you can use either the `TaskFactory` or the constructor of the `Task` and the `Start()` method. The `Task` constructor just gives you more flexibility in creating the task.

When starting a task, an instance of the `Task` class can be created and the code that should run can be assigned, with an `Action` or `Action<object>` delegate, with either no parameters or one object parameter. This is similar to what you saw with the `Thread` class. Here, a method is defined without a parameter. In the implementation, the ID of the task is written to the console.

Available for download on Wrox.com

```
static void TaskMethod()
{
    Console.WriteLine("running in a task");
    Console.WriteLine("Task id: {0}", Task.CurrentId);
}
```

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

In the previous code, you can see different ways to start a new task. The first way is with an instantiated `TaskFactory`, where the method `TaskMethod` is passed to the `StartNew()` method, and the task is immediately started. The second approach uses the constructor of the `Task` class. When the `Task` object is instantiated, the task does not run immediately. Instead, it is given the status `Created`. The task is then started by calling the `Start()` method of the `Task` class. With the `Task` class, instead of invoking the `Start()` method, you can invoke the `RunSynchronously()` method. This way, the task is started as well, but it is running in the current thread of the caller, and the caller needs to wait until the task finishes. By default, the task runs asynchronously.

```
    // using task factory
    TaskFactory tf = new TaskFactory();
    Task t1 = tf.StartNew(TaskMethod);

    // using the task factory via a task
    Task t2 = Task.Factory.StartNew(TaskMethod);

    // using Task constructor
    Task t3 = new Task(TaskMethod);
  t3.Start();
```

With both the `Task` constructor and the `StartNew()` method of the `TaskFactory`, you can pass values from the enumeration `TaskCreationOptions`. Setting the option `LongRunning`, you can inform the task scheduler that the task takes a long time, so the scheduler will more likely use a new thread. If the task should be attached to the parent task and, thus, should be canceled if the parent were canceled, set the option `AttachToParent`. The value `PreferFairness` means that the scheduler should take first tasks that are already waiting. That's not the default case if a task is created within another task. If tasks create additional work using child tasks, child tasks are preferred to other tasks. They are not waiting last in the thread pool queue for the work to be done. If these tasks should be handled in a fair manner to all other tasks, set the option to `PreferFairness`.

```
    Task t4 = new Task(TaskMethod, TaskCreationOptions.PreferFairness);
  t4.Start();
```

Continuation Tasks

With tasks, you can specify that after a task is finished another specific task should start to run, for example, a new task that uses a result from the previous one or that should do some cleanup if the previous task failed.

Whereas the task handler has either no parameter or one object parameter, the continuation handler has a parameter of type `Task`. Here, you can access information about the originating task.

Available for download on Wrox.com

```
static void DoOnFirst()
{
    Console.WriteLine("doing some task {0}", Task.CurrentId);
    Thread.Sleep(3000);
}

static void DoOnSecond(Task t)
{
    Console.WriteLine("task {0} finished", t.Id);
    Console.WriteLine("this task id {0}", Task.CurrentId);
    Console.WriteLine("do some cleanup");
    Thread.Sleep(3000);
}
```

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

A continuation task is defined by invoking the `ContinueWith` method on a task. You could also use the `TaskFactory` for this. `t1.OnContinueWith(DoOnSecond)` means that a new task invoking the method `DoOnSecond()` should be started as soon as the task *t1* is finished. You can start multiple tasks when one task is finished, and a continuation task can also have another continuation task, as this example demonstrates.

```
    Task t1 = new Task(DoOnFirst);
    Task t2 = t1.ContinueWith(DoOnSecond);
    Task t3 = t1.ContinueWith(DoOnSecond);
   Task t4 = t2.ContinueWith(DoOnSecond);
```

The continuation tasks so far were always started when the previous task was finished, no matter how the previous task was finished. With values from `TaskContinuationOptions`, you can define that a continuation task should only start if the originating task was successful (or faulted). Some of the possible values are `OnlyOnFaulted`, `NotOnFaulted`, `OnlyOnCanceled`, `NotOnCanceled`, and `OnlyOnRanToCompletion`.

```
    Task t5 = t1.ContinueWith(DoOnError,
                              TaskContinuationOptions.OnlyOnFaulted);
```

Task Hierarchies

With task continuations, one task is started after another. Tasks can also form a hierarchy. When a task itself starts a new task, a parent/child hierarchy is started.

In the code snippet that follows, within the task of the parent, a new task is created. The code to create a child task is the same as creating a parent task. The only difference is that the task is created from within another task.

Available for download on Wrox.com

```
    static void ParentAndChild()
    {
        var parent = new Task(ParentTask);
        parent.Start();
        Thread.Sleep(2000);
        Console.WriteLine(parent.Status);
        Thread.Sleep(4000);
        Console.WriteLine(parent.Status);

    }
    static void ParentTask()
    {
        Console.WriteLine("task id {0}", Task.CurrentId);
        var child = new Task(ChildTask);
        child.Start();
        Thread.Sleep(1000);
        Console.WriteLine("parent started child");
    }
    static void ChildTask()
    {
        Console.WriteLine("child");
        Thread.Sleep(5000);
        Console.WriteLine("child finished");
   }
```

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

If the parent task is finished before the child task, the status of the parent task is shown as `WaitingForChildrenToComplete`. The parent task is completed with the status `RanToCompletion` as soon as all children are completed as well. Of course, this is not valid if the parent creates a task with the `TaskCreationOption DetachedFromParent`.

Canceling a parent task also cancels the children. The cancellation framework is discussed later.

Results from Tasks

When a task is finished, it can write some stateful information to a shared object. Such a shared object must be thread-safe. Another option is to use a task that returns a result. With the generic version of the `Task` class, it is possible to define the type that is returned with a task that returns a result.

A method that is invoked by a task to return a result can be declared with any return type. The example method `TaskWithResult` returns two `int` values with the help of a `Tuple`. The input of the method can be void or of type `object`, as shown here.

Available for download on Wrox.com

```
    static Tuple<int, int> TaskWithResult(object division)
    {
        Tuple<int, int> div = (Tuple<int, int>)division;
        int result = div.Item1 / div.Item2;
        int reminder = div.Item1 % div.Item2;
        Console.WriteLine("task creates a result...");

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

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

Tuples are explained earlier in the "Tuples" section.

When defining a task to invoke the method `TaskWithResult`, the generic class `Task<TResult>` is used. The generic parameter defines the return type. With the constructor, the method is passed to the `Func` delegate, and the second parameter defines the input value. Because this task needs two input values in the `object` parameter, a tuple is created as well. Next, the task is started. The `Result` property of the `Task` instance `t1` blocks and waits until the task is completed. Upon task completion, the `Result` property contains the result from the task.

```
var t1 = new Task<Tuple<int,int>>(TaskWithResult,
                                  Tuple.Create<int, int>(8, 3));
t1.Start();
Console.WriteLine(t1.Result);
t1.Wait();
Console.WriteLine("result from task: {0} {1}", t1.Result.Item1,
                  t1.Result.Item2);
```

PARALLEL CLASS

Another abstraction of threads that is new with .NET 4 is the `Parallel` class. This class defines static methods for a parallel `for` and `foreach`. With the language defined for `for` and `foreach`, the loop is run from one thread. The `Parallel` class uses multiple tasks and, thus, multiple threads for this job.

While the `Parallel.For()` and `Parallel.ForEach()` methods invoke the same method several times, `Parallel.Invoke()` allows invoking different methods concurrently.

Looping with the Parallel.For Method

The `Parallel.For()` method is similar to the C# `for` loop statement to do a task a number of times. With the `Parallel.For()`, the iterations run in parallel. The order of iteration is not defined.

With the `For()` method, the first two parameters define the start and end of the loop. The sample has the iterations from 0 to 9. The third parameter is an `Action<int>` delegate. The integer parameter is the iteration of the loop that is passed to the method referenced by the delegate. The return type of `Parallel.For()` is the struct `ParallelLoopResult`, which provides information if the loop is completed.

Available for download on Wrox.com

```
 ParallelLoopResult result =
     Parallel.For(0, 10, i =>
         {
             Console.WriteLine("{0}, task: {1}, thread: {2}", i,
                Task.CurrentId, Thread.CurrentThread.ManagedThreadId);
             Thread.Sleep(10);
         });
Console.WriteLine(result.IsCompleted);
```

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

In the body of the `Parallel.For()`, the index, task identifier, and thread identifier are written to the console. As you can see from this output, the order is not guaranteed. This run of the program had the order 0-5-1-6-2… with three tasks and three threads.

```
0, task: 1, thread: 1
5, task: 2, thread: 3
1, task: 3, thread: 4
6, task: 2, thread: 3
2, task: 1, thread: 1
4, task: 3, thread: 4
7, task: 2, thread: 3
3, task: 1, thread: 1
8, task: 3, thread: 4
9, task: 3, thread: 4
True
```

You can also break the `Parallel.For()` early. A method overload of the `For()` method accepts a third parameter of type `Action<int, ParallelLoopState>`. By defining a method with these parameters, you can influence the outcome of the loop by invoking the `Break()` or `Stop()` methods of the `ParallelLoopState`.

Remember, the order of iterations is not defined.

```
            ParallelLoopResult result =
                Parallel.For(10, 40, (int i, ParallelLoopState pls) =>
                    {
                        Console.WriteLine("i: {0} task {1}", i, Task.CurrentId);
                        Thread.Sleep(10);
                        if (i > 15)
                            pls.Break();
                    });
            Console.WriteLine(result.IsCompleted);
            Console.WriteLine("lowest break iteration: {0}",
                          result.LowestBreakIteration);
```

This run of the application demonstrates that the iteration breaks up with a value higher than 15, but other tasks can simultaneously run and tasks with other values can run. With the help of the `LowestBreakIteration` property, you can decide to ignore results from other tasks.

```
10 task 1
25 task 2
11 task 1
13 task 3
12 task 1
14 task 3
16 task 1
15 task 3
False
lowest break iteration: 16
```

`Parallel.For()` might use several threads to do the loops. If you need an initialization that should be done with every thread, you can use the `Parallel.For<TLocal>()` method. The generic version of the `For` method accepts — besides the `from` and `to` values — three delegate parameters. The first parameter is of type `Func<TLocal>`. Because the example here uses a string for `TLocal`, the method needs to be defined as `Func<string>`, a method returning a `string`. This method is invoked only once for each thread that is used to do the iterations.

The second delegate parameter defines the delegate for the body. In the example, the parameter is of type `Func<int, ParallelLoopState, string, string>`. The first parameter is the loop iteration; the second parameter, `ParallelLoopState`, allows for stopping the loop, as you saw earlier. With the third parameter, the body method receives the value that is returned from the `init` method. The body method also needs to return a value of the type that was defined with the generic `For` parameter.

The last parameter of the `For()` method specifies a delegate, `Action<TLocal>`; in the example, a string is received. This method again is called only once for each thread; this is a thread exit method.

```
               Parallel.For<string>(0, 20,
                   () =>
                   {
                       // invoked once for each thread
                       Console.WriteLine("init thread {0}, task {1}",
                           Thread.CurrentThread.ManagedThreadId, Task.CurrentId);
                       return String.Format("t{0}",
                           Thread.CurrentThread.ManagedThreadId);
                   },
                   (i, pls, str1) =>
                   {
                       // invoked for each member
                       Console.WriteLine("body i {0} str1 {1} thread {2} task {3}", i,
str1,
                           Thread.CurrentThread.ManagedThreadId,
                           Task.CurrentId);
                       Thread.Sleep(10);
                       return String.Format("i {0}", i);

                   },
                   (str1) =>
                   {
                       // final action on each thread
                       Console.WriteLine("finally {0}", str1);
                 });
```

The result of one time running this program is shown here:

```
init thread 1, task 1
body i 0 str1 t1 thread 1 task 1
body i 1 str1 i 0 thread 1 task 1
init thread 3, task 2
body i 10 str1 t3 thread 3 task 2
init thread 4, task 3
body i 3 str1 t4 thread 4 task 3
body i 2 str1 i 1 thread 1 task 1
body i 11 str1 i 10 thread 3 task 2
body i 4 str1 i 3 thread 4 task 3
body i 6 str1 i 2 thread 1 task 1
body i 12 str1 i 11 thread 3 task 2
body i 5 str1 i 4 thread 4 task 3
body i 7 str1 i 6 thread 1 task 1
body i 13 str1 i 12 thread 3 task 2
body i 17 str1 i 5 thread 4 task 3
body i 8 str1 i 7 thread 1 task 1
body i 14 str1 i 13 thread 3 task 2
body i 9 str1 i 8 thread 1 task 1
body i 18 str1 i 17 thread 4 task 3
body i 15 str1 i 14 thread 3 task 2
body i 19 str1 i 18 thread 4 task 3
finally i 9
body i 16 str1 i 15 thread 3 task 2
finally i 19
finally i 16
```

Looping with the Parallel.ForEach Method

`Parallel.ForEach` iterates through a collection implementing `IEnumerable` in a way similar to the `foreach` statement, but in an asynchronous manner. Again, the order is not guaranteed.

Available for download on Wrox.com

```
string[] data = {"zero", "one", "two", "three", "four",
                 "five", "six", "seven", "eight", "nine",
                 "ten", "eleven", "twelve"};

ParallelLoopResult result =
    Parallel.ForEach<string>(data, s =>
        {
            Console.WriteLine(s);
        });
```

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

If you need to break up the loop, you can use an overload of the `ForEach()` method with a `ParallelLoopState` parameter. You can do this in the same way as with the `For()` method you saw earlier. An overload of the `ForEach()` method can also be used to access an indexer to get the iteration number as shown.

```
Parallel.ForEach<string>(data,
    (s, pls, l) =>
    {
        Console.WriteLine("{0} {1}", s, l);

    });
```

Invoking Multiple Methods with the Parallel.Invoke Method

If multiple tasks should run in parallel, you can use the `Parallel.Invoke()` method. `Parallel.Invoke()` allows the passing of an array of `Action` delegates, where you can assign methods that should run. The sample code passes the `Foo` and `Bar` methods to be invoked in parallel.

Available for download on Wrox.com

```
static void ParallelInvoke()
{
    Parallel.Invoke(Foo, Bar);
}

static void Foo()
```

```
{
    Console.WriteLine("foo");
}

static void Bar()
{
    Console.WriteLine("bar");
}
```

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

CANCELLATION FRAMEWORK

.NET 4 includes a new cancellation framework to allow the canceling of long-running tasks in a standard manner. Every blocking call should support this mechanism. As of today, of course, not every blocking call implements this new technology, but more and more are doing so. Among the technologies that offer this mechanism already are tasks, concurrent collection classes, and Parallel LINQ, as well as several synchronization mechanisms.

The cancellation framework is based on cooperative behavior; it is not forceful. A long-running task checks if it is canceled and returns control.

A method that supports cancellation accepts a `CancellationToken` parameter. This class defines the property `IsCancellationRequested`, where a long operation can check if it should abort. Other ways for a long operation to check for cancellation are to use a `WaitHandle` property that is signaled when the token is canceled, or to use the `Register()` method. The `Register()` method accepts parameters of type `Action` and `ICancelableOperation`. The method that is referenced by the `Action` delegate is invoked when the token is canceled. This is similar to the `ICancelableOperation`, where the `Cancel()` method of an object implementing this interface is invoked when the cancellation is done.

Cancellation of Parallel.For

Let's start with a simple example using the `Parallel.For()` method. The `Parallel` class provides overloads for the `For()` method, where you can pass parameter of type `ParallelOptions`. With the `ParallelOptions`, you can pass a `CancellationToken`. The `CancellationToken` is generated by creating a `CancellationTokenSource`. `CancellationTokenSource` implements the interface `ICancelableOperation` and, thus, can be registered with the `CancellationToken` and allows cancellation with the `Cancel()` method. To cancel the parallel loop, a new task is created to invoke the `Cancel()` method of the `CancellationTokenSource` after 500 milliseconds.

Within the implementation of the `For()` loop, the `Parallel` class verifies the outcome of the `CancellationToken` and cancels the operation. Upon cancellation, the `For()` method throws an exception of type `OperationCanceledException`, which is caught in the example. With the `CancellationToken`, it is possible to register for information when the cancellation is done. This is accomplished by calling the `Register()` method and passing a delegate that is invoked on cancellation.

Available for download on Wrox.com

```
var cts = new CancellationTokenSource();
cts.Token.Register(() =>
    Console.WriteLine("*** token canceled"));

// start a task that sends a cancel after 500 ms
new Task(() =>
   {
       Thread.Sleep(500);
       cts.Cancel(false);
   }).Start();

try
{
    ParallelLoopResult result =
       Parallel.For(0, 100,
           new ParallelOptions()
           {
               CancellationToken = cts.Token,
           },
           x =>
           {
               Console.WriteLine("loop {0} started", x);
               int sum = 0;
               for (int i = 0; i < 100; i++)
               {
                   Thread.Sleep(2);
                   sum += i;
               }
               Console.WriteLine("loop {0} finished", x);
           });
}
catch (OperationCanceledException ex)
{
    Console.WriteLine(ex.Message);
}
```

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

When running the application, you will get output similar to the following. Iteration 0, 1, 25, 26, 50, 51, 75, and 76 were all started. This is on a system with a quad-core CPU. With the cancellation, all other iterations were canceled before starting. The iterations that were started are allowed to finish because cancellation is always done in a cooperative way so as to avoid the risk of resource leaks when iterations are canceled somewhere in between.

```
loop 0 started
loop 50 started
loop 25 started
loop 75 started
loop 50 finished
loop 25 finished
loop 0 finished
loop 1 started
loop 26 started
loop 51 started
loop 75 finished
loop 76 started
** token canceled
loop 1 finished
loop 51 finished
loop 26 finished
loop 76 finished
The operation was canceled.
```

Cancellation of Tasks

The same cancellation pattern is used with tasks. First, a new `CancellationTokenSource` is created. If you need just one cancellation token, you can use a default one by accessing `Task.Factory.CancellationToken`. Then, similar to the previous code, a new task is created that sends a cancel request to this `cancellationSource` by invoking the `Cancel()` method after 500 milliseconds. The task doing the major work within a loop receives the cancellation token via the `TaskFactory` object. The cancellation token is assigned to the `TaskFactory` by setting it in the constructor. This cancellation token is used by the task to check if cancellation is requested by checking the `IsCancellationRequested` property of the `CancellationToken`.

Available for download on Wrox.com

```
var cts = new CancellationTokenSource();
cts.Token.Register(() =>
    Console.WriteLine("*** task canceled"));


// start a task that sends a cancel to the
// cts after 500 ms
Task.Factory.StartNew(() =>
{
    Thread.Sleep(500);
    cts.Cancel();
});

var factory = new TaskFactory(cancellationSource.Token);
Task t1 = factory.StartNew(new Action<object>(f =>
    {
        Console.WriteLine("in task");
        for (int i = 0; i < 20; i++)
        {
            Thread.Sleep(100);
            CancellationToken ct = (f as TaskFactory).CancellationToken;
            if (ct.IsCancellationRequested)
            {
                Console.WriteLine("canceling was requested, " +
                    "canceling from within the task");
                ct.ThrowIfCancellationRequested();
                break;
            }
            Console.WriteLine("in loop");
        }
```

```
            Console.WriteLine("task finished without cancellation");
        }), factory, cts.Token);

    try
    {
        t1.Wait();
    }
    catch (Exception ex)
    {
        Console.WriteLine("exception: {0}, {1}", ex.GetType().Name,
            ex.Message);
        if (ex.InnerException != null)
            Console.WriteLine("inner exception: {0}, {1}",
                              ex.InnerException.GetType().Name,
                              ex.InnerException.Message);
    }
  Console.WriteLine("status of the task: {0}", t1.Status);
```

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

When running the application, you can see that the task starts, runs for a few loops, and gets the cancellation request. The task is canceled and throws a `TaskCanceledException`, which is initiated from the method call `ThrowIfCancellationRequested()`. With the caller waiting for the task, you can see that the exception `AggregateException` is caught and contains the inner exception `TaskCanceledException`. This is used for a hierarchy of cancellations, for example, if you run a `Parallel.For` within a task that is canceled as well. The final status of the task is `Canceled`.

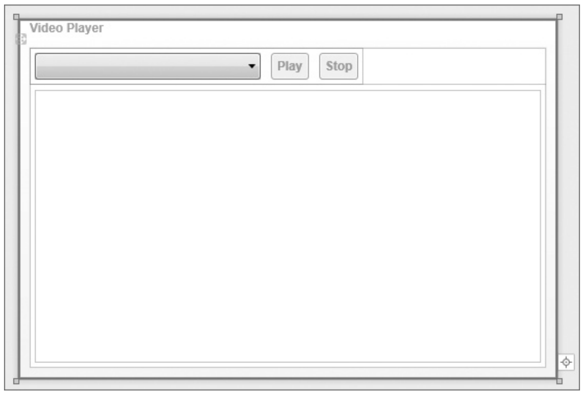
```
in task
in loop
in loop
in loop
in loop
*** task canceled
canceling was requested, canceling from within the task
exception AggregateException, One or more errors occurred.
inner exception TaskCanceledException, A task was canceled.
status of the task: Canceled
```

TASKBAR AND JUMP LIST

Windows 7 has a new taskbar. In the taskbar, you see not only the running applications, but also fast access icons. The user can pin most-often-used applications for fast access. When you hover over an item of the taskbar, you can see a preview of the currently running application. The item can also have visual state so the user can receive feedback from the items on the taskbar. Did you copy some large files using Explorer? You can see progress information on the Explorer item. Progress information is shown with the visual state. Another feature of the taskbar is the Jump List. Clicking the right mouse key on a taskbar button opens the Jump List. This list can be customized per application. With Microsoft Outlook, you can go directly to the Inbox, Calendar, Contacts, and Tasks or create a new e-mail. With Microsoft Word, you can open the recent documents.

The capability to add all these features to WPF applications is included with the .NET Framework 4 in the namespace `System.Windows.Shell`.

The sample application shown in this section lets you plays videos, gives you visual feedback on the taskbar button as to whether the video is running or stopped, and allows you to start and stop a video directly from the taskbar. The main window of the application consists of a grid with two rows, as shown here. The first row contains a `ComboBox` to display available videos, and two buttons to start and stop the player. The second row contains a `MediaElement` for playing the videos.



The XAML code of the user interface is shown below. The buttons are associated with the commands `MediaCommands.Play` and `MediaCommands.Stop`, which map to the handler methods `OnPlay()` and `OnStop()`. With the `MediaElement` the property `LoadedBehavior` is set to `Manual` so that the player doesn't start immediately on loading the video.

Available for download on Wrox.com

```
<Window.CommandBindings>
    <CommandBinding Command="MediaCommands.Play" Executed="OnPlay" />
    <CommandBinding Command="MediaCommands.Stop" Executed="OnStop" />
</Window.CommandBindings>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <StackPanel Grid.Row="0" Orientation="Horizontal"
                HorizontalAlignment="Left">
        <ComboBox x:Name="comboVideos" ItemsSource="{Binding}" Width="220"
                  Margin="5" IsSynchronizedWithCurrentItem="True" />
        <Button x:Name="buttonPlay" Content="Play" Margin="5" Padding="4"
                Command="MediaCommands.Play" />
        <Button x:Name="buttonStop" Content="Stop" Margin="5" Padding="4"
                Command="MediaCommands.Stop" />
    </StackPanel>
    <MediaElement x:Name="player" Grid.Row="1" LoadedBehavior="Manual"
            Margin="5"
            Source="{Binding ElementName=comboVideos, Path=SelectedValue}" />
</Grid>
```

Pro C# 4 9780470502259 code snippet **`TaskbarDemo/MainWindow.xaml`**

To configure the taskbar item, the namespace `System.Windows.Shell` contains a dependency property for the `Window` class to add taskbar information. The `TaskbarItemInfo` property can contain a `TaskbarItemInfo` element. With `TaskbarItemInfo`, you can set the `Description` property, which is shown as tooltip information. With the properties `ProgressState` and `ProgressValue`, feedback can be given on a current state of the application. `ProgressState` is of type `TaskbarItemProgressState`, which defines the enumeration values `None`, `Intermediate`, `Normal`, `Error`, and `Paused`. Depending on the value of this setting, progress indicators are shown on the taskbar item. The `Overlay` property allows you to define an image that is displayed over the icon in the taskbar. This property will be set from code-behind.

The taskbar item can contain a `ThumbButtonInfoCollection`, which is assigned to the `TumbButtonInfos` property of the `TaskbarItemInfo`. Here, you can add buttons of type `ThumbButtonInfo`, which are displayed in the preview of the application. The sample contains two `ThumbButtonInfo` elements, which have `Command` set to the same commands as the play and stop `Button` elements created previously. With these buttons, you can control the application in the same way as with the other `Button` elements in the application. The image for the buttons in the taskbar is taken from resources with the keys `StartImage` and `StopImage`.

```
<Window.TaskbarItemInfo>
    <TaskbarItemInfo x:Name="taskBarItem" Description="Sample Application">
        <TaskbarItemInfo.ThumbButtonInfos>
            <ThumbButtonInfo IsEnabled="True" Command="MediaCommands.Play"
                             CommandTarget="{Binding ElementName=buttonPlay}"
                             Description="Play"
                             ImageSource="{StaticResource StartImage}" />
            <ThumbButtonInfo IsEnabled="True" Command="MediaCommands.Stop"
                             CommandTarget="{Binding ElementName=buttonStop}"
                             Description="Stop"
                             ImageSource="{StaticResource StopImage}" />
        </TaskbarItemInfo.ThumbButtonInfos>
    </TaskbarItemInfo>
</Window.TaskbarItemInfo>
```

The images referenced from the `ThumbButtonInfo` elements are defined within the `Resources` section of the `Window`. One image is made up from a light green ellipse, the other image from two orange-red lines.

```
<Window.Resources>
    <DrawingImage x:Key="StopImage">
        <DrawingImage.Drawing>
            <DrawingGroup>
                <DrawingGroup.Children>
                    <GeometryDrawing>
                        <GeometryDrawing.Pen>
                            <Pen Thickness="5" Brush="OrangeRed" />
                        </GeometryDrawing.Pen>
                        <GeometryDrawing.Geometry>
                            <GeometryGroup>
                                <LineGeometry StartPoint="0,0"
                                              EndPoint="20,20" />
                                <LineGeometry StartPoint="0,20"
                                              EndPoint="20,0" />
                            </GeometryGroup>
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                </DrawingGroup.Children>
            </DrawingGroup>
        </DrawingImage.Drawing>
    </DrawingImage>
    <DrawingImage x:Key="StartImage">
        <DrawingImage.Drawing>
            <DrawingGroup>
                <DrawingGroup.Children>
                    <GeometryDrawing Brush="LightGreen">
                        <GeometryDrawing.Geometry>
                            <EllipseGeometry RadiusX="20" RadiusY="20"
                                             Center="20,20" />
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                </DrawingGroup.Children>
            </DrawingGroup>
        </DrawingImage.Drawing>
    </DrawingImage>
</Window.Resources>
```

In the code-behind, all videos from the special My Videos folder are assigned to the `DataContext` to the `Window` object, and thus, because the `ComboBox` is data-bound, listed in the `ComboBox`. In the `OnPlay()` and `OnStop()` event handlers, the `Play()` and `Stop()` methods of the `MediaElement` are invoked. To give visual feedback to the taskbar item, as well on playing and stopping a video, image resources are accessed and assigned to the `Overlay` property of the `TaskbarItemInfo` element.

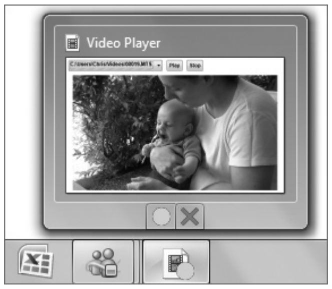
Available for download on Wrox.com

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shell;

namespace Wrox.ProCSharp.Windows7
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            string videos = Environment.GetFolderPath(
                                        Environment.SpecialFolder.MyVideos);

            this.DataContext = Directory.EnumerateFiles(videos);
        }

        private void OnPlay(object sender, ExecutedRoutedEventArgs e)
        {
            object image = TryFindResource("StartImage");
            if (image is ImageSource)
```

```
                    taskBarItem.Overlay = image as ImageSource;
                player.Play();
            }

            private void OnStop(object sender, ExecutedRoutedEventArgs e)
            {
                object image = TryFindResource("StopImage");
                if (image is ImageSource)
                    taskBarItem.Overlay = image as ImageSource;
                player.Stop();
            }
        }
    }
```

Pro C# 4 9780470502259 code snippet **`TaskbarDemo/MainWindow.xaml.cs`**

Now you can run the application, start and stop the video from the taskbar, and see visual feedback, as shown here.

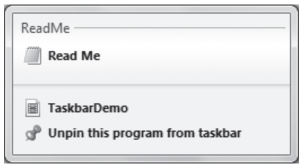


Customizing the Jump List is done by adding a `JumpList` to the application class. This can either be done in code by invoking the static method `JumpList.SetJumpList()` or by adding `JumpList` elements as a child of the `Application` element.

The sample code creates the Jump List in the code-behind. `JumpList.SetJumpList()` requires as the first argument the object of the application, which is returned from the `Application.Current` property. The second argument requires an object of type `JumpList`. The *`jumpList`* is created with a `JumpList` constructor passing `JumpItem` elements and two Boolean values. By default, a Jump List contains frequent and recent items. Setting the Boolean values, you can influence this default behavior. The items you can add to a `JumpList` derive from the baseclass `JumpItem`. The classes available are `JumpTask` and `JumpPath`. With `JumpTask`, you can define a program that should be started when the user selects this item from the Jump List. `JumpTask` defines the properties that are needed to start the application and give information to the user, which are `CustomCategory`, `Title`, `Description`, `ApplicationPath`, `IconResourcePath`, `WorkingDirectory`, and `Arguments`. With the `JumpTask` that is defined in the code snippet, `notepad.exe` is started and initialized with the `readme.txt` document. With a `JumpPath` item, you can define a file that should be listed in the Jump List to start the application from the file. `JumpPath` defines a `Path` property for assigning the filename. Using a `JumpPath` item requires that the file extension be registered with the application. Otherwise, the registration of the items is rejected. To find the reasons for items being rejected, you can register a handler to the `JumpItemsRejected` event. Here, you can get a list of the rejected items (`RejectedItems`) and the reasons for them (`RejectedReasons`).

```
var jumpItems = new List<JumpTask>();
var workingDirectory = Environment.CurrentDirectory;
var windowsDirectory = Environment.GetFolderPath(
                               Environment.SpecialFolder.Windows);
var notepadPath = System.IO.Path.Combine(windowsDirectory,
                                         "Notepad.exe");
jumpItems.Add(new JumpTask
{
    CustomCategory="Read Me",
    Title="Read Me",
    Description="Open read me in Notepad.",
    ApplicationPath=notepadPath,
    IconResourcePath=notepadPath,
    WorkingDirectory=workingDirectory,
    Arguments="readme.txt"
});

var jumpList = new JumpList(jumpItems, true, true);
jumpList.JumpItemsRejected += (sender1, e1) =>
    {
        var sb = new StringBuilder();
        for (int i = 0; i < e1.RejectedItems.Count; i++)
        {
            if (e1.RejectedItems[i] is JumpPath)
                sb.Append((e1.RejectedItems[i] as JumpPath).Path);
            if (e1.RejectedItems[i] is JumpTask)
                sb.Append((e1.RejectedItems[i] as JumpTask).
                          ApplicationPath);
            sb.Append(e1.RejectionReasons[i]);
            sb.AppendLine();
        }
        MessageBox.Show(sb.ToString());
    };

JumpList.SetJumpList(Application.Current, jumpList);
```

When you run the application and click the right mouse button, you see the Jump List, as shown here.



To use the taskbar and Jump List from Windows Forms applications, you can use extensions defined in the Windows API Code Pack.

Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional

ASP.NET 4 in C# and VB

Bill Evjen, Scott Hanselman, Devin Rader

CHART SERVER CONTROL

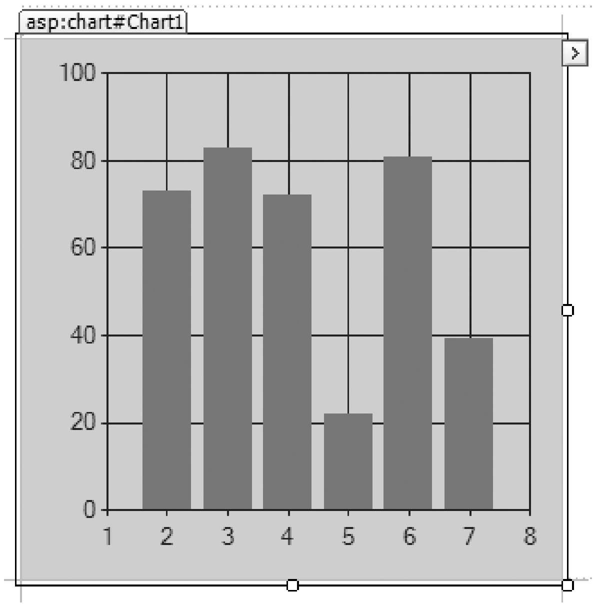
One of the newest controls available to you now with ASP.NET 4 is the Chart server control. This control made its way into the core part of ASP.NET through a previous acquisition of the Dundas charting company and is a great control for getting you up and running with some good-looking charts.

The new Chart server control supports many chart types including:

- Point
- FastPoint
- Bubble
- Line
- Spline
- StepLine
- FastLine
- Bar
- StackedBar
- StackedBar100
- Column
- StackedColumn
- StackedColumn100
- Area
- SplineArea
- StackedArea
- StackedArea100
- Pie
- Doughnut
- Stock
- CandleStick
- Range
- SplineRange
- RangeBar
- RangeColumn
- Radar
- Polar
- ErrorBar
- BoxPlot
- Renko
- ThreeLineBreak
- Kagi
- PointAndFigure
- Funnel
- Pyramid

Those are a lot of different chart styles! You can find the Chart server control in the toolbox of Visual Studio 2010 underneath the Data tab. It is part of the `System.Web.DataVisualization` namespace.

When you drag it from the toolbox and place it on the design surface of your page, you are presented with a visual representation of the chart type that are you going to use. The next figure shows an example.



Open up the smart tag for the control, and you find that you can assign a data provider for the chart as well as select the chart type you are interested in using. Changing the chart type gives you a sample of what that chart looks like (even if you are not yet working with any underlying data) in the design view of the IDE. The smart tag is shown in the next figure.

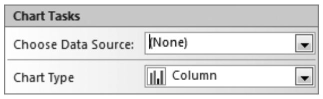


Chart Tasks

Choose Data Source: (None)

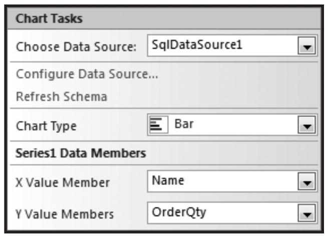
Chart Type: Column

There is a lot to this control, probably more than all the others and this single control could almost warrant a book on its own. To get you up and running with this chart server control, follow this simple example.

Create a new web application and add the AdventureWorks database to your App_Data folder within the application. After that is accomplished, drag and drop the Chart server control onto the design surface of your page. From the smart tag of the control, select <New Data Source> from the drop-down menu when choosing your data source. Work your way through this wizard making sure that you are choosing a SQL data source as your option. As you work through the wizard, you are going to want to choose the option that allows you to choose a custom SQL statement and use the following SQL for this operation:

```
SELECT TOP (5) Production.Product.Name, Sales.SalesOrderDetail.OrderQty
FROM Sales.SalesOrderDetail INNER JOIN Production.Product
ON Sales.SalesOrderDetail.ProductID = Production.Product.ProductID
ORDER BY Sales.SalesOrderDetail.OrderQty DESC
```

With that in place and the new chart server control bound to this data source control, you now find that you have more options in the smart tag of the chart server control, as shown in this figure.



Choose Data Source: SqlDataSource1
Chart Type: Bar
X Value Member: Name
Y Value Members: OrderQty

Now you can select the series data members and choose what is on the x-axis and what is on the y-axis. You can see that I have assigned the Name of the product to be on the x-axis and the quantity ordered to be on the y-axis. After widening the chart's width a bit, you end up with code similar to the following in Listing 3-52.

Available for download on Wrox.com

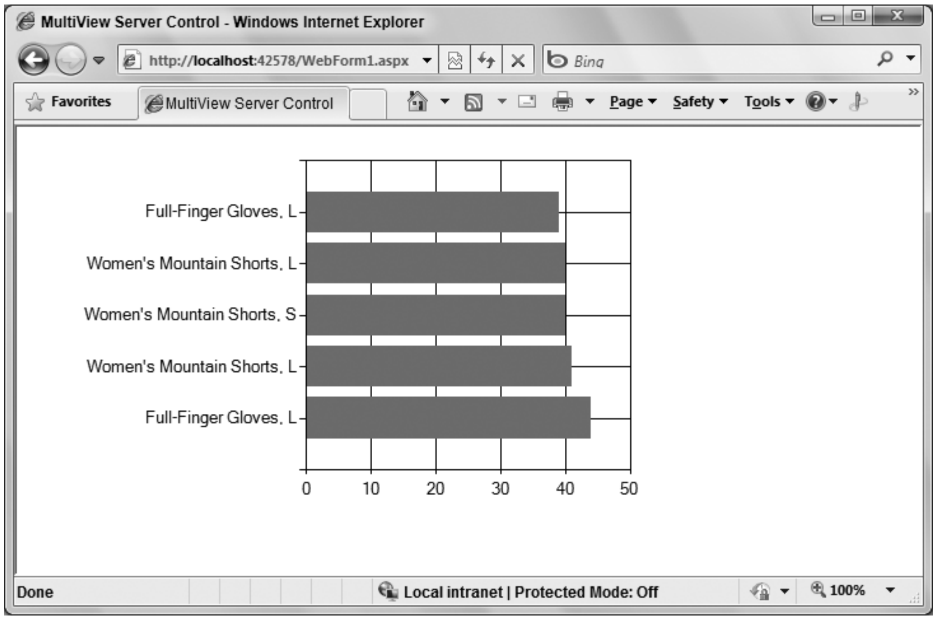
Pro ASP.NET 9780470502204 Listing 3-52: Charting with the new Chart server control

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeBehind="WebForm1.aspx.cs" Inherits="WebServerControls.WebForm1" %>
<%@ Register Assembly="System.Web.DataVisualization, Version=4.0.0.0,
      Culture=neutral, PublicKeyToken=31bf3856ad364e35"
    Namespace="System.Web.UI.DataVisualization.Charting" TagPrefix="asp" %>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>MultiView Server Control</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Chart ID="Chart1" runat="server"
         DataSourceID="SqlDataSource1"
         Width="500px">
            <Series>
                <asp:Series ChartType="Bar" Name="Series1"
                 XValueMember="Name"
                 YValueMembers="OrderQty" YValuesPerPoint="2">
                </asp:Series>
            </Series>
            <ChartAreas>
                <asp:ChartArea Name="ChartArea1">
                </asp:ChartArea>
            </ChartAreas>
        </asp:Chart>
        <asp:SqlDataSource ID="SqlDataSource1" runat="server"
         ConnectionString="<%$ ConnectionStrings:ConnectionString %>"
         SelectCommand="SELECT TOP (5) Production.Product.Name,
                Sales.SalesOrderDetail.OrderQty FROM Sales.SalesOrderDetail
                INNER JOIN Production.Product ON
                Sales.SalesOrderDetail.ProductID =
                Production.Product.ProductID ORDER BY
                Sales.SalesOrderDetail.OrderQty DESC">
             </asp:SqlDataSource>
         </div>
         </form>
     </body>
     </html>
```

From this, you can see that there isn't much code needed to wire everything up. Most notably, you can see that putting this Chart server control on your page actually added a `@Register` directive to the top of the page. This is unlike most of the ASP.NET server controls.

Within the `<Series>` element of this control, you can have as many series as you want, and this is something that is quite common when charting multiple items side by side (such as a time series of prices for two or more stocks).

Running this code, you get results similar to what is presented here.

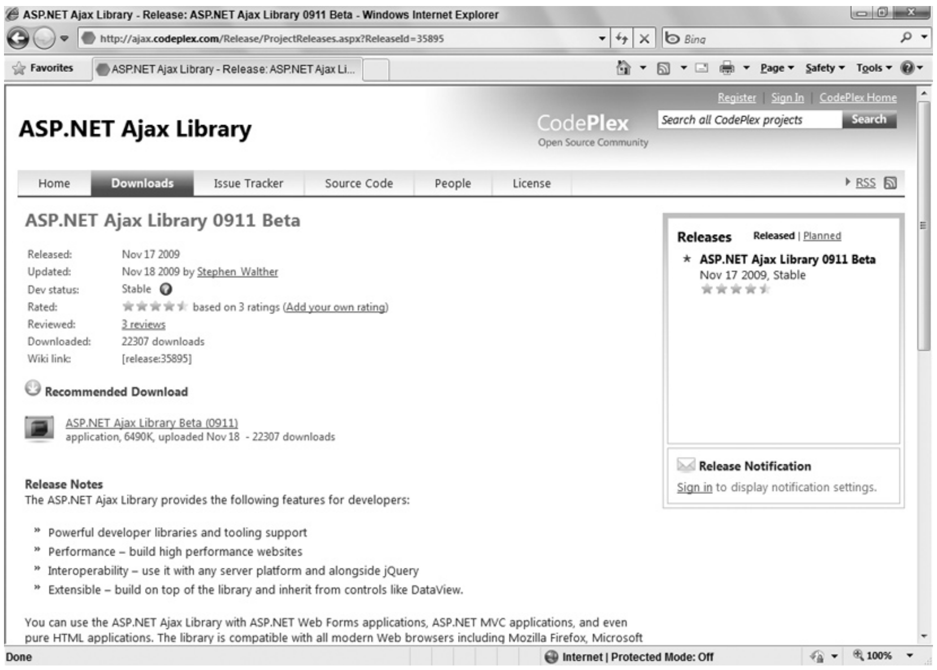

MultiView Server Control - Windows Internet Explorer
http://localhost:42578/WebForm1.aspx
Bing
Favorites
MultiView Server Control
Page
Safety
Tools
Full-Finger Gloves, L
Women's Mountain Shorts, L
Women's Mountain Shorts, S
Women's Mountain Shorts, L
Full-Finger Gloves, L
0
10
20
30
40
50
Done
Local intranet | Protected Mode: Off
100%

ASP.NET AJAX CONTROL TOOLKIT

With the install of the .NET Framework 4 and through using Visual Studio 2010, you will find a few controls available that allow you to build ASP.NET applications with server-side AJAX capabilities. This is the framework to take your applications to the next level because you can accomplish so much with it, including adding specific AJAX capabilities to your user and custom server controls. Every AJAX enhancement added to your application will make your application seem more fluid and responsive to the end user.

You might be wondering where the big AJAX-enabled server controls are for this edition of Visual Studio 2010 if this is indeed a new world for building Web applications. The reason you do not see a new section of AJAX server controls is that Microsoft has treated them as an open source project instead of just blending them into Visual Studio 2010.

Developers at Microsoft and in the community have been working on a series of AJAX-capable server controls that you can use in your ASP.NET applications. These controls are collectively called the ASP.NET AJAX Control Toolkit. Recently, Microsoft has made the ASP.NET AJAX Control Toolkit a part of the Microsoft AJAX Library found at `www.asp.net/ajaxlibrary`. Downloading the library, you will find that it now also includes the complete AJAX Control Toolkit. The following figure shows the download page for the AJAX Library.


ASP.NET Ajax Library - Release: ASP.NET Ajax Library 0911 Beta - Windows Internet Explorer
http://ajax.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=35895
Bing
Favorites
ASP.NET Ajax Library - Release: ASP.NET Ajax Li...
Page
Safety
Tools
Register
Sign In
CodePlex Home
CodePlex
Open Source Community
Search all CodePlex projects
Search
Home
Downloads
Issue Tracker
Source Code
People
License
RSS
Done
Internet | Protected Mode: Off
100%

ASP.NET Ajax Library

ASP.NET Ajax Library 0911 Beta

Released:	Nov 17 2009
Updated:	Nov 18 2009 by Stephen Walther
Dev status:	Stable
Rated:	based on 3 ratings (Add your own rating)
Reviewed:	3 reviews
Downloaded:	22307 downloads
Wiki link:	[release:35895]

Recommended Download

ASP.NET Ajax Library Beta (0911)
application, 6490K, uploaded Nov 18 - 22307 downloads

Releases Released | Planned
* ASP.NET Ajax Library 0911 Beta
Nov 17 2009, Stable

Release Notification
Sign in to display notification settings.

Release Notes
The ASP.NET Ajax Library provides the following features for developers:

» Powerful developer libraries and tooling support
» Performance – build high performance websites
» Interoperability – use it with any server platform and alongside jQuery
» Extensible – build on top of the library and inherit from controls like DataView.

You can use the ASP.NET Ajax Library with ASP.NET Web Forms applications, ASP.NET MVC applications, and even pure HTML applications. The library is compatible with all modern Web browsers including Mozilla Firefox, Microsoft

ASP.NET AJAX is the foundation on which to build richer Web applications that leverage the browser more fully, but it does not have the rich UI elements that really blur the distinction between Web and desktop applications. With the Microsoft AJAX Library, you can apply familiar object-oriented design concepts and use the scripts across a variety of modern browsers. ASP.NET AJAX includes several powerful ASP.NET controls that make adding AJAX functionality to an existing application or building better user experiences into a new application easy. The AJAX Toolkit, however, was developed to provide rich ASP.NET AJAX controls that you can use to make your Web applications really come to life. The Toolkit makes pushing the user interface of an application beyond what users expect from a Web application easy.

The Toolkit is a shared source project with code contributions from developers from Microsoft and elsewhere. Most developers who work with ASP.NET AJAX should also download the Toolkit for the additional set of controls it contains. The AJAX Library download mentioned earlier allows you to download a compiled DLL with the controls and extenders, or you can download the source code and project files and compile it yourself. Either way, make sure you add the DLL to your Toolbox in Visual Studio, as described shortly.

The Toolkit contains some controls that have AJAX functionality and many control extenders. The control extenders attach to another control to enhance or “extend” the control’s functionality. When you install the Toolkit, it creates a sample Web application that has some examples of using the controls. Because the controls cover such a wide variety of application-development areas, they are presented in categories: page layout controls and user interface effects. Within each category, the controls are listed alphabetically and the control names are self-explanatory to make locating the information you need for later reference easy.

Also, note that the Toolkit project is ongoing and will continue to evolve as developers contribute to it. This text is up-to-date as of the time of this writing, but the expectation is that more will be added to the Toolkit regularly. Most interesting is that even though this is an open source project, now that the AJAX Control Toolkit is included with the AJAX Library from Microsoft, it is also a supported product from Microsoft.

Downloading and Installing the AJAX Control Toolkit

Because the ASP.NET AJAX Control Toolkit is not part of the default install of Visual Studio 2010, you must set up the controls yourself. Again, the control toolkit’s site on Microsoft’s AJAX site offers a couple of options.

One option is the control toolkit that is specifically targeted at Visual Studio 2008 or greater. This section focuses on using the control toolkit with Visual Studio 2010. Another option is you can download the ASP.NET AJAX Control Toolkit as either source code or a compiled DLL, depending on your needs.

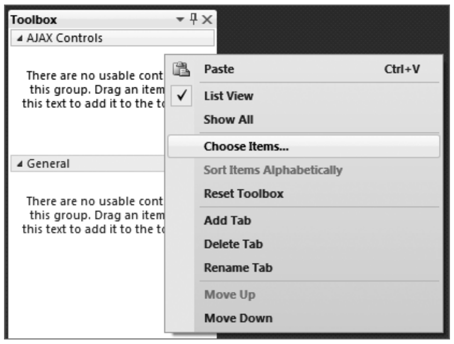
The source code option enables you to take the code for the controls and ASP.NET AJAX extenders and change the behavior of the controls or extenders yourself. The DLL option is a single Visual Studio installer file and a sample application.

There are a couple of parts to this install. One part provides a series of controls that were built with AJAX capabilities in mind. Another part is a series of control extenders (extensions for pre-existing controls).

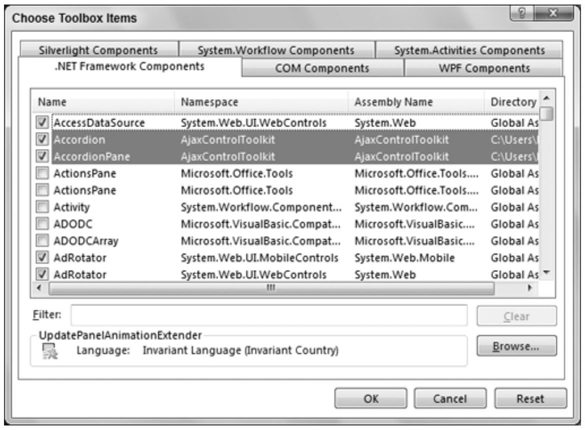
To get set up, download the `.zip` file from the aforementioned site at `www.asp.net/ajaxlibrary` and unzip it where you want on your machine. Then follow these steps:

1. Install the controls into Visual Studio. Adding the controls to your Visual Studio 2010 Toolbox is very easy. Right-click in the Toolbox and select Add Tab from the provided menu. Name the tab as you want — for this example, the tab is called AJAX Controls.

2. With the new tab in your Toolbox, right-click the tab and select Choose Items from the provided menu, as illustrated in the next figure. The Choose Toolbox Items dialog appears.


Toolbox
AJAX Controls
There are no usable cont this group. Drag an item this text to add it to the t
General
There are no usable cont this group. Drag an item this text to add it to the t
Paste Ctrl+V
List View
Show All
Choose Items...
Sort Items Alphabetically
Reset Toolbox
Add Tab
Delete Tab
Rename Tab
Move Up
Move Down

3. Select the `AjaxControlToolkit.dll` from the download. When you find the DLL and click Open, the Choose Toolbox Items dialog changes to include the controls that are contained within this DLL. The controls are highlighted in the dialog and are already selected for you (as shown in the next figure).

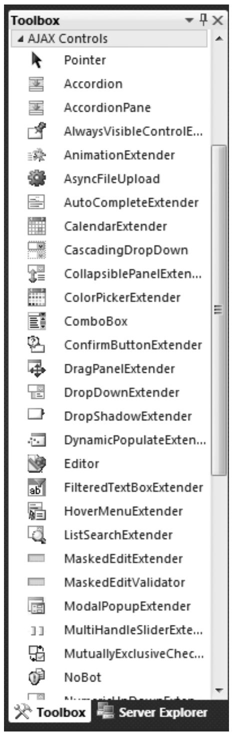

Choose Toolbox Items
Silverlight Components
System.Workflow Components
System.Activities Components
.NET Framework Components
COM Components
WPF Components

Name	Namespace	Assembly Name	Directory
✓ AccessDataSource	System.Web.UI.WebControls	System.Web	Global As
✓ Accordion	AjaxControlToolkit	AjaxControlToolkit	C:\Users\
✓ AccordionPane	AjaxControlToolkit	AjaxControlToolkit	C:\Users\
ActionsPane	Microsoft.Office.Tools	Microsoft.Office.Tools....	Global As
ActionsPane	Microsoft.Office.Tools	Microsoft.Office.Tools....	Global As
Activity	System.Workflow.Component...	System.Workflow.Com...	Global As
ADODC	Microsoft.VisualBasic.Compat...	Microsoft.VisualBasic....	Global As
ADODCArray	Microsoft.VisualBasic.Compat...	Microsoft.VisualBasic....	Global As
✓ AdRotator	System.Web.UI.MobileControls	System.Web.Mobile	Global As
✓ AdRotator	System.Web.UI.WebControls	System.Web	Global As

Filter:

UpdatePanelAnimationExtender
Language: Invariant Language (Invariant Country)

4. Click OK, and the ASP.NET AJAX Control Toolkit's controls will be added to your Visual Studio Toolbox. The next figure presents the end result.



More than 40 controls and extenders have been added to the Toolbox for use in your ASP.NET applications.

ColorPickerExtender

One of the difficult data points to retrieve from an end user is color. This particular data point is tough to define if you are using just text. If you have an open selection of colors, how does the end user define a darker shade of blue? For this reason, you have the ColorPickerExtender to quickly and easily extend something like a TextBox control to a tool that makes this selection process a breeze. The following code listing shows a quick and easy way to do this task.

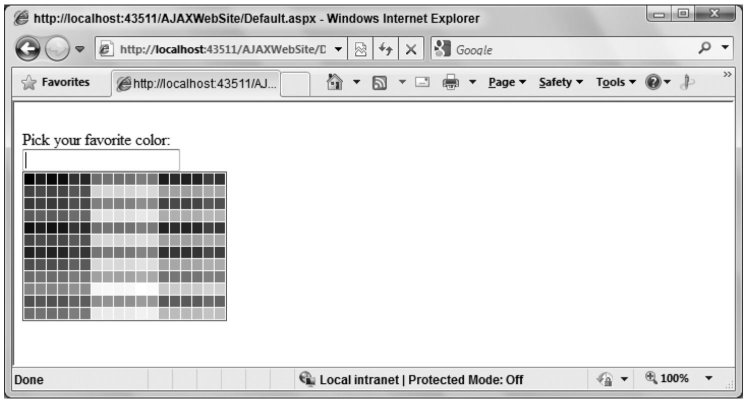
Available for download on Wrox.com

Pro ASP.NET 9780470502204 Listing 19-9: Using the ColorPickerExtender control to allow for color selection

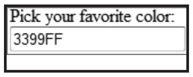
```
<%@ Page Language="C#" %>

<%@ Register Assembly="AjaxControlToolkit"
    Namespace="AjaxControlToolkit" TagPrefix="asp" %>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>ColorPickerExtender</title>
</head>
<body>
```

```
        <form id="form1" runat="server">
        <div>
            <asp:ToolkitScriptManager ID="ToolkitScriptManager1" runat="server">
            </asp:ToolkitScriptManager>
            <br />
            Pick your favorite color:<br />
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            <asp:ColorPickerExtender ID="ColorPickerExtender1"
             runat="server" TargetControlID="TextBox1">
            </asp:ColorPickerExtender>
        </div>
        </form>
    </body>
    </html>
```

When this page opens, you simply have a single TextBox server control on the page. Applying focus to this TextBox control pops up the color selector, as illustrated here in black and white in the next figure.


http://localhost:43511/AJAXWebSite/Default.aspx - Windows Internet Explorer
Favorites
Page
Safety
Tools
Pick your favorite color:
Done
Local intranet | Protected Mode: Off
100%

The end user then can scroll across the color options and after the user selects one of these colors, the pop-up disappears and the hexadecimal color code is shown in the TextBox. This end result is presented here.



EXTENDING <OUTPUTCACHE>

With the release of ASP.NET 4, you are now able to extend how the `OutputCache` directive works and have it instead work off your own custom means to caching. This means that you can wire the `OutputCache` directive to any type of caching means including distributed caches, cloud caches, disc, XML, or anything else you can dream up.

To accomplish this, you are required to create a custom output-cache provider as a class and this class will need to inherit from the new `System.Web.Caching.OutputCacheProvider` class. To inherit from `OutputCacheProvider`, you must override the `Add()`, `Get()`, `Remove()`, and `Set()` methods to implement your custom version of output caching.

After you have your custom implementation in place, the next step is to configure this in a configuration file: the `machine.config` or the `web.config` file. Some changes have been made to the `<outputCache>` element in the configuration file to allow you to apply your custom cache extensions.

The `<outputCache>` element is found within the `<caching>` section of the configuration file and it now includes a new `<providers>` subelement.

```
<caching>
   <outputCache>
      <providers>
      </providers>
   </outputCache>
</caching>
```

Within the new `<providers>` element, you can nest an `<add>` element to make the appropriate references to your new output cache capability you built by deriving from the `OutputCacheProvider` class.

```
<caching>
   <outputCache defaultProvider="AspNetInternalProvider">
      <providers>
         <add name="myDistributedCacheExtension"
          type="Wrox.OutputCacheExtension.DistributedCacheProvider,
                  DistributedCacheProvider" />
      </providers>
   </outputCache>
</caching>
```

With this new `<add>` element in place, your new extended output cache is available to use. One new addition here to also pay attention to is the new `defaultProvider` attribute within the `<outputCache>` element. In this case, it is set to `AspNetInternalProvider`, which is the default setting in the configuration file. This means that by default the output cache works as it always has done and stores its cache in the memory of the computer that the program is running.

With your own output cache provider in place, you can now point to this provider through the `OutputCache` directive on the page as defined here:

```
<%@ OutputCache Duration="90" VaryByParam="*"
    providerName="myDistributedCacheExtension" %>
```

If the provider name isn't defined, then the provider that is defined in the configuration's `defaultProvider` attribute is utilized.

.NET 4'S NEW OBJECT CACHING OPTION

From what you have seen so far with the `System.Web.Caching.Cache` object, you can see that it is quite powerful and allows for you to even create a custom cache. This extensibility and power has changed under the hood of the `Cache` object, though.

Driving this is the `System.Runtime.Caching.dll`, as what was in the `System.Web` version has been refactored out and everything was rebuilt into the new namespace of `System.Runtime.Caching`.

The reason for this change wasn't so much for the ASP.NET developer, but instead for other application types such as Windows Forms, Windows Presentation Foundation apps, and more. The reason for this is that the `System.Web.Caching.Cache` object was so useful that other application developers were bringing over the `System.Web` namespace into their projects to make use of this object. So, to get away from a Windows Forms developer needing to bring the `System.Web.dll` into their project just to use the `Cache` object it provided, this was all extracted out and extended with the `System.Runtime.Caching` namespace.

As an ASP.NET developer, you can still make use of the `System.Web.Caching.Cache` object just as you did in all the prior versions of ASP.NET. It isn't going away. However, it is important to note that as the .NET Framework evolves, the .NET team will be making its investments into the `System.Runtime.Caching` namespace rather than `System.Web.Caching`. This means that over time, you will most likely see additional enhancements in the `System.Runtime.Caching` version that don't appear in the `System.Web.Caching` namespace as you might expect. With that said, it doesn't also mean that you need to move everything over to the new `System.Runtime.Caching` namespace to make sure you are following the strategic path of Microsoft, because the two caches are managed together under the covers.

This section runs through an example of using the cache from the `System.Runtime.Caching` namespace. For this example, the ASP.NET page simply uses a Label control that shows the name of a user that is stored in an XML file. The first step is to create an XML file and name the file `Username.xml`. This simple file is presented here in in the following listing.

Available for download on Wrox.com

Pro ASP.NET 9780470502204 Listing 22-5: The contents of the Username.xml file

```
<?xml version="1.0" encoding="utf-8" ?>
<usernames>
  <user>Bill Evjen</user>
</usernames>
```

With this XML file sitting in the root of your drive, now turn your attention to the `Default.aspx` code behind page to use the name in the file and present it into a single Label control on the page. The code behind for the Default.aspx page is presented here in the following listing.

Available for download on Wrox.com

VB

Pro ASP.NET 9780470502204 Listing 22-6: Using the System.Runtime.Caching namespace

```
Imports System
Imports System.Collections.Generic
Imports System.Linq
Imports System.Runtime.Caching
Imports System.Xml.Linq

Partial Public Class _Default
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(ByVal sender As Object,
      ByVal e As EventArgs) Handles Me.Load

        Dim cache As ObjectCache = MemoryCache.Default

        Dim usernameFromXml As String =
          TryCast(cache("userFromXml"), String)

        If usernameFromXml Is Nothing Then
```

```
            Dim userFilePath As New List(Of String)()
            userFilePath.Add("C:\Username.xml")

            Dim policy As New CacheItemPolicy()
            policy.ChangeMonitors.Add(New
              HostFileChangeMonitor(userFilePath))

            Dim xdoc As XDocument =
              XDocument.Load("C:\Username.xml")
            Dim query = From u In xdoc.Elements("usernames")
                Select u.Value

            usernameFromXml = query.First().ToString()

            cache.Set("userFromXml", usernameFromXml, policy)
        End If

        Label1.Text = usernameFromXml
    End Sub
End Class
```

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Caching;
using System.Xml.Linq;

namespace RuntimeCaching
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            ObjectCache cache = MemoryCache.Default;

            string usernameFromXml = cache["userFromXml"] as string;

            if (usernameFromXml == null)
            {
                List<string> userFilePath = new List<string>();
                userFilePath.Add(@"C:\Username.xml");

                CacheItemPolicy policy = new CacheItemPolicy();
                policy.ChangeMonitors.Add(new
                   HostFileChangeMonitor(userFilePath));

                XDocument xdoc = XDocument.Load(@"C:\Username.xml");
                var query = from u in xdoc.Elements("usernames")
                            select u.Value;

                usernameFromXml = query.First().ToString();

                cache.Set("userFromXml", usernameFromXml, policy);
            }

            Label1.Text = usernameFromXml;
        }
    }
}
```

The preceding example from makes use of the new cache at `System.Runtime.Caching`. You need to reference this namespace in your ASP.NET project for this to work.

To start, you create a default instance of the cache object.

```
ObjectCache cache = MemoryCache.Default;
```

You then can work with this cache as you would with the traditional ASP.NET cache object.

```
string usernameFromXml = cache["userFromXml"] as string;
```

To get the cache started, you need to create an object that defines what type of cache you are dealing with. You can build a custom implementation, or you can use one of the default implementations provided with .NET 4.

```
CacheItemPolicy policy = new CacheItemPolicy();
policy.ChangeMonitors.Add(new HostFileChangeMonitor(userFilePath));
```

The `HostFileChangeMonitor` is a means to look at directories and file paths and monitor for change. So, for instance, when the XML file changes, that triggers an invalidation of the cache. Other implementations of the `ChangeMonitor` object include the `FileChangeMonitor` and the `SqlChangeMonitor`.

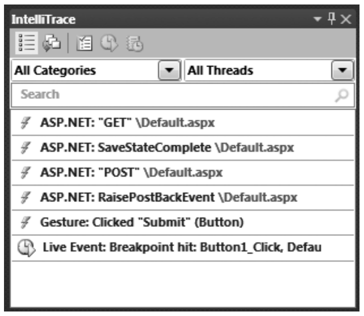
Running this example, notice that the text *Bill Evjen* is loaded into the cache on the first run, and this text appears in the Label1 control. Keep your application running and then you can go back to the XML file and change the value, and you will then notice that this causes the cache to be invalidated on the page's refresh.

HISTORICAL DEBUGGING WITH INTELLITRACE

One of the more exciting features of Visual Studio 2010 is a new feature called IntelliTrace. IntelliTrace provides you a historical debugger that allows you to look back in the history of your application running and allows you to jump at previous points in time to see what was occurring.

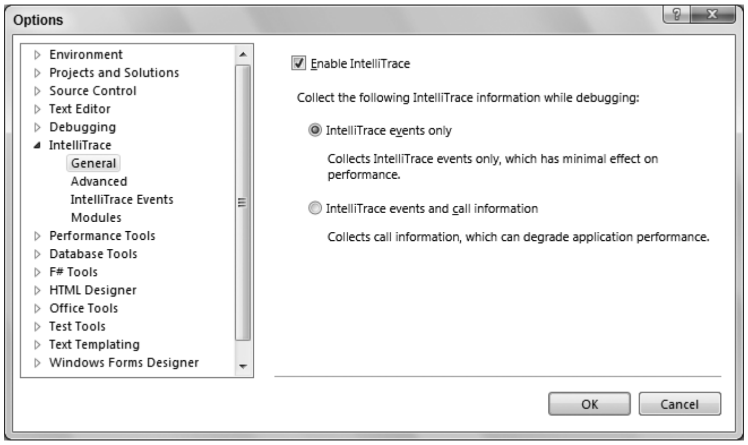
This feature only works with Visual Studio 2010 Ultimate Edition and is not available in the other editions of the IDE. As of today, it also does not work on 64-bit machines. Therefore, you need to be working with a 32-bit computer to work with the historical debugging feature.

When stopping at a breakpoint within your application, you will now be able to see this history in the IntelliTrace dialog directly as illustrated in the next figure.



ASP.NET: "GET" \Default.aspx
ASP.NET: SaveStateComplete \Default.aspx
ASP.NET: "POST" \Default.aspx
ASP.NET: RaisePostBackEvent \Default.aspx
Gesture: Clicked "Submit" (Button)
Live Event: Breakpoint hit: Button1_Click, Defau

From here, you can jump to any point in the past and view the Locals dialog values and the Call Stack dialog values for that moment in time. The IntelliTrace capability is configurable as well. You can get to these settings by either clicking on the Open IntelliTrace Settings from the IntelliTrace toolbar or by selecting Tools ➪ Options and selecting IntelliTrace from the items in the left side of the dialog. On the General tab of the IntelliTrace section (shown in the next figure), you can enable or disable the IntelliTrace option and, if it is enabled, choose whether to record just the events or to enable even more and record events and call information.



[✓] Enable IntelliTrace

Collect the following IntelliTrace information while debugging:

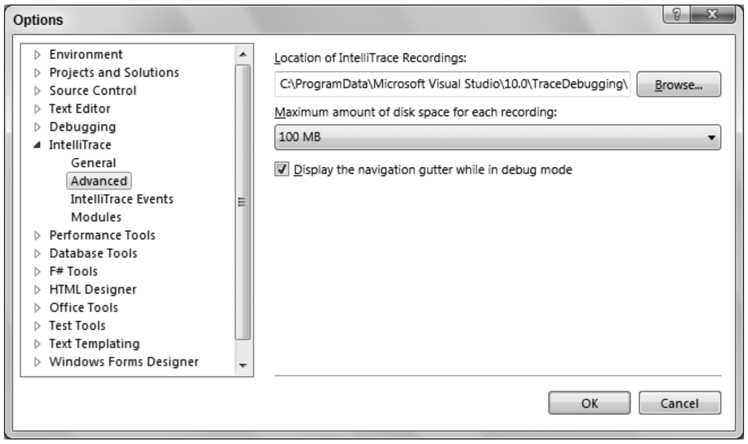
[● IntelliTrace events only]

Collects IntelliTrace events only, which has minimal effect on performance.

○ IntelliTrace events and call information

Collects call information, which can degrade application performance.

On the Advanced tab (shown in the next figure), you can set the amount of disk space to use for recording what is going on in the application. Here you can see that the amount of space reserved is 100MB.



Location of IntelliTrace Recordings:

C:\ProgramData\Microsoft Visual Studio\10.0\TraceDebugging\

Maximum amount of disk space for each recording:

100 MB

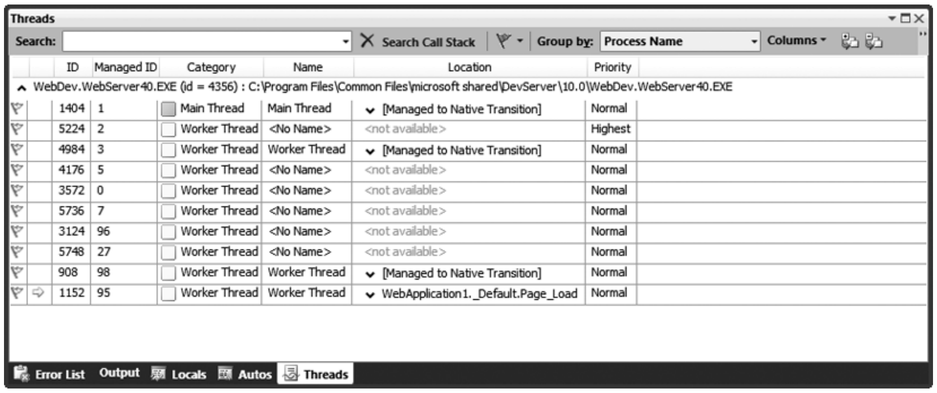
[✓] Display the navigation gutter while in debug mode

From the IntelliTrace Events tab, you can select the events that you want to work with. You can choose from a large series of technologies and situations by making the appropriate selections.

Finally, the Modules tab allows you to specify particular modules that IntelliTrace should work with.

DEBUGGING MULTIPLE THREADS

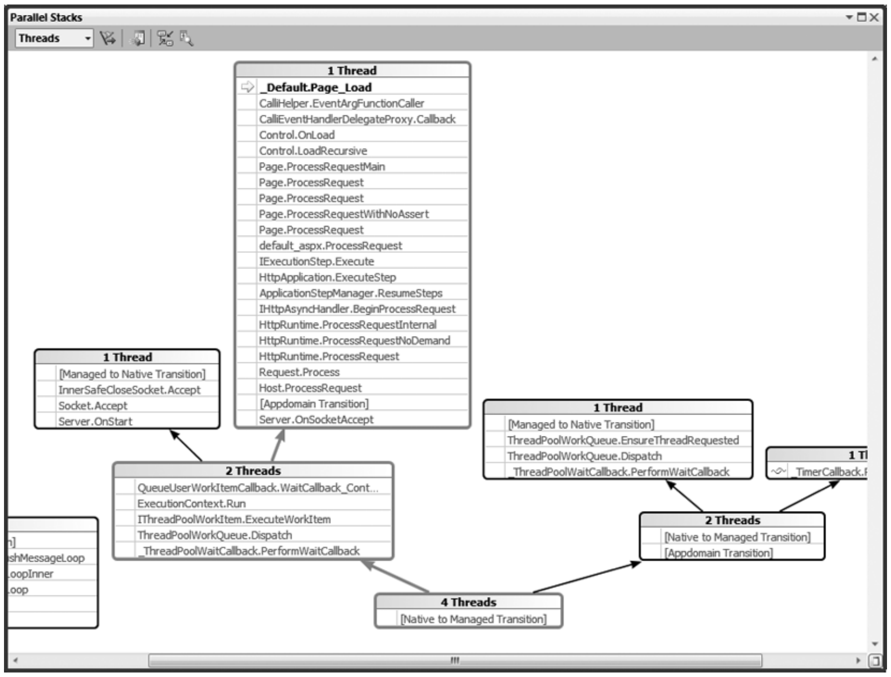
Visual Studio 2010 includes a dialog that allows you to see what is going on in each of the threads your application is using. This is done through the Threads dialog shown in the next figure.



WebDev.WebServer40.EXE (id = 4356) : C:\Program Files\Common Files\microsoft shared\DevServer\10.0\WebDev.WebServer40.EXE

	ID	Managed ID	Category	Name	Location	Priority
	1404	1	Main Thread	Main Thread	[Managed to Native Transition]	Normal
	5224	2	Worker Thread	<No Name>	<not available>	Highest
	4984	3	Worker Thread	Worker Thread	[Managed to Native Transition]	Normal
	4176	5	Worker Thread	<No Name>	<not available>	Normal
	3572	0	Worker Thread	<No Name>	<not available>	Normal
	5736	7	Worker Thread	<No Name>	<not available>	Normal
	3124	96	Worker Thread	<No Name>	<not available>	Normal
	5748	27	Worker Thread	<No Name>	<not available>	Normal
	908	98	Worker Thread	Worker Thread	[Managed to Native Transition]	Normal
⇨	1152	95	Worker Thread	Worker Thread	WebApplication1._Default.Page_Load	Normal

You can double-click on a thread and go to where that thread is in the code when debugging. You can also get a visual family tree view of the threads at work by opening up the Parallel Stacks dialog. The next figure shows this dialog.



1 Thread
_Default.Page_Load
CallHelper.EventArgFunctionCaller
CallEventHandlerDelegateProxy.Callback
Control.OnLoad
Control.LoadRecursive
Page.ProcessRequestMain
Page.ProcessRequest
Page.ProcessRequest
Page.ProcessRequestWithNoAssert
Page.ProcessRequest
default_aspx.ProcessRequest
IExecutionStep.Execute
HttpApplication.ExecuteStep
ApplicationStepManager.ResumeSteps
IHttpAsyncHandler.BeginProcessRequest
HttpRuntime.ProcessRequestInternal
HttpRuntime.ProcessRequestNoDemand
HttpRuntime.ProcessRequest
Request.Process
Host.ProcessRequest
[Appdomain Transition]
Server.OnSocketAccept

1 Thread
[Managed to Native Transition]
InnerSafeCloseSocket.Accept
Socket.Accept
Server.OnStart

2 Threads
QueueUserWorkItemCallback.WaitCallback_Cont...
ExecutionContext.Run
IThreadPoolWorkItem.ExecuteWorkItem
ThreadPoolWorkQueue.Dispatch
_ThreadPoolWaitCallback.PerformWaitCallback

1 Thread
[Managed to Native Transition]
ThreadPoolWorkQueue.EnsureThreadRequested
ThreadPoolWorkQueue.Dispatch
_ThreadPoolWaitCallback.PerformWaitCallback

1 Tl
TimerCallback.F

2 Threads
[Native to Managed Transition]
[Appdomain Transition]

4 Threads
[Native to Managed Transition]

n]
shMessageLoop
.oopInner
.oop

ASP.NET MVC

Model-View-Controller (MVC) has been an important architectural pattern in computer science for many years. Originally named Thing-Model-View-Editor in 1979, it was later simplified to Model-View-Controller. It is a powerful and elegant means of separating concerns within an application and applies itself extremely well to Web applications. Its explicit separation of concerns does add a small amount of extra complexity to an application's design, but the extraordinary benefits outweigh the extra effort. It's been used in dozens of frameworks since its introduction. You can find MVC in Java and C++, on Mac and on Windows, and inside literally dozens of frameworks.

Understanding the core concepts of MVC is critical to using it effectively. This section discusses the history of the MVC pattern, as well as how it is used in Web programming today.

ASP.NET MVC 1.0 shipped as a downloadable add-on to Visual Studio 2008. Now in Visual Studio 2010, ASP.NET MVC 2 ships built-in. This section also covers some of the limitations of ASP.NET Web forms and how ASP.NET MVC attempts to release the developer from those limitations.

The content of this selection is adapted from Professional ASP.NET MVC 1.0 *by Rob Conery, Scott Hanselman, Phil Haack, and Scott Guthrie (Wiley, 2009). For more in-depth coverage of this topic, we recommend you check out that book.*

Defining Model-View-Controller

Model-View-Controller (MVC) is an architectural pattern used to separate an application into three main aspects:

- ➤ The Model: A set of classes that describes the data you're working with as well as the business rules for how the data can be changed and manipulated
- ➤ The View: The application's user interface (UI)
- ➤ The Controller: A set of classes that handles communication from the user, overall application flow, and application-specific logic

This pattern is used frequently in Web programming. With ASP.NET MVC, it's translated roughly to the following:

- ➤ The Models are the classes that represent the domain in which you are interested. These domain objects often encapsulate data stored in a database as well as code used to manipulate the data and enforce domain-specific business logic. With ASP.NET MVC, this is most likely a data access layer of some kind using a tool like LINQ to SQL, Entity Framework, or NHibernate, combined with custom code containing domain-specific logic.
- ➤ The View is a dynamically generated page. In ASP.NET MVC, you implement it via the `System.Web .Mvc.ViewPage` class, which inherits from `System.Web.UI.Page`.
- ➤ The Controller is a special class that manages the relationship between the View and Model. It talks to the Model, and it decides which View to render (if any). In ASP.NET MVC, this class is conventionally denoted by the suffix "`Controller`."

MVC on the Web Today

For many, the Web didn't really become prominent until the first graphical browsers began to flood the market, starting with Mosaic in 1993. Shortly after, dynamic Web pages began showing up using languages such as Perl and enabled by technologies like the Common Gateway Interface (CGI). The technology available in the early stages of the Web was focused more around the concept of scripting HTML to do light content-driven work, as opposed to deep application logic, which just wasn't needed back then.

As the Web grew and HTML standards began to allow for richer interactions, the notion of the Web as an application platform began to take off. In the Microsoft realm, the focus was on quick and simple (in line with the simplicity of VB), and Active Server Pages (ASP) was born in 1996.

ASP used VBScript, a very simple, lightweight language that gave developers a lot of "un-prescribed freedom" in terms of the applications they could create. A request for an ASP page would be handled by a file with the `.asp` extension, which consisted of a server-side script intermixed with HTML markup. Written in a procedural language, many ASP pages often devolved into "spaghetti code" in which the markup was intertwined with code in difficult-to-manage ways. Although writing clean ASP code was possible, it took a lot of work, and the language and tools were not sufficiently helpful. Even so, ASP did provide full control over the markup produced, it just took a lot of work.

In January of 2002, Microsoft released version 1.0 of the .NET platform, which included the original version of ASP.NET, and thus Web forms was born. Its birth provided access to advanced tools and object-oriented languages for building a Web site.

ASP.NET has grown tremendously over the last 8 years (almost a decade!) and has made developing Web pages very productive and simple by abstracting the repetitive tasks of Web development into simple drag-and-drop controls. This abstraction can be a tremendous help, but some developers have found that they want more control over the generated HTML and browser scripting, and they also want to be able to easily test their Web page logic.

As languages matured and Web server software grew in capability, MVC soon found its way into Web application architectures. But MVC didn't hit its mainstream stride until July of 2004, when a 24-year old developer at 37Signals in Chicago, Illinois, named David Heinemeier Hansson, introduced the world to his fresh way of thinking about MVC.

David, or DHH as he's known in the community, created Ruby on Rails, a Web development framework that used the Ruby language and the MVC pattern to create something special.

Now let's further delve into the new kid on the block, ASP.NET MVC, and answer the question, "Why not Web forms?"

In February of 2007, Scott Guthrie of Microsoft sketched out the core of ASP.NET MVC while flying on a plane to a conference on the east coast of the United States. It was a simple application, containing a few hundred lines of code, but the promise and potential it offered for parts of the Microsoft Web developer audience was huge.

PRODUCT TEAM ASIDE

ScottGu, or "The Gu," is legendary for prototyping cool stuff, and if he sees you in the hallway and he has his laptop, you won't be able to escape as he says, "Dude! Check this out!" His enthusiasm is infectious.

As the legend goes, at the Austin ALT.NET conference in October of 2007 in Redmond, Washington, ScottGu showed a group of developers "this cool thing he wrote on a plane" and asked whether they saw the need and what they thought of it. It was a hit. In fact, a number of people were involved with the original prototype, codenamed "Scalene." Eilon Lipton e-mailed the first prototype to the team in September of 2007, and he and ScottGu bounced prototypes, code, and ideas back and forth via e-mail, and still do!

Model-View-Controller and ASP.NET

ASP.NET MVC relies on many of the same core strategies that the other MVC platforms use, plus it offers the benefits of compiled and managed code and exploits new language features in .NET 3.5 and above within VB9 and C#3 such as lambdas and anonymous types. Each of the MVC frameworks used on the web usually share in some fundamental tenets:

- Convention over Configuration
- Don't repeat yourself (also known as the DRY principle)
- Plugability whenever possible
- Try to be helpful, but if necessary, get out of the developer's way

Serving Methods, Not Files

Web servers initially served up HTML stored in static files on disk. As dynamic Web pages gained prominence, Web servers served HTML generated on the fly from dynamic scripts that were also located on disk. With MVC, serving up HTML is a little different. The URL tells the routing mechanism which Controller to instantiate and which action method to call and supplies the required arguments to that method. The Controller's method then decides which View to use, and that View then does the rendering.

Rather than having a direct relationship between the URL and a file living on the Web server's hard drive, a relationship exists between the URL and a method on a controller object. ASP.NET MVC implements the "front controller" variant of the MVC pattern, and the Controller sits in front of everything except the routing subsystem.

A good way to conceive of the way that MVC works in a Web scenario is that MVC serves up the results of method calls, not dynamically generated (also known as scripted) pages. In fact, we heard a speaker once call this "RPC for the Web," which is particularly apt, although quite a bit narrower in scope.

Is This Web Forms 4.0?

One of the major concerns that we've heard when talking to people about ASP.NET MVC is that its release means the death of Web forms. This just isn't true. ASP.NET MVC is not ASP.NET Web forms 4.0. It's an alternative to Web forms, and it's a fully supported part of the framework. While Web forms continues to march on with new innovations and new developments, ASP.NET MVC will continue as a parallel alternative that's totally supported by Microsoft.

One interesting way to look at this is to refer to the namespaces these technologies live in. If you could point to a namespace and say, "That's where ASP.NET lives," it would be the `System.Web` namespace. ASP.NET MVC lives in the `System.Web.Mvc` namespace. It's not `System.Mvc`, and it's not `System.Web2`.

Although ASP.NET MVC is a separately downloadable Web component today for users of Visual Studio 2008 (often referred to by the ASP.NET team as an out-of-band [OOB] release), it has been folded into .NET Framework 4 and it's built into Visual Studio 2010 out of the box. This cements ASP.NET MVC's place as a fundamental part of ASP.NET itself.

Why Not Web Forms?

In ASP.NET Web forms, you create an instance of `System.Web.UI.Page` and put "server controls" on it (for example, a calendar and some buttons) so that the user can enter or view information. You then wire these controls to events on the `System.Web.UI.Page` to allow for interactivity. This page is then compiled, and when it's called by the ASP.NET runtime, a server-side control tree is created, each control in the tree goes through an event lifecycle, it renders itself, and the result is served back as HTML. As a result, a new Web aesthetic started to emerge — Web forms layers eventing and state management on top of HTTP — a truly stateless protocol.

Why was this abstraction necessary? Remember that Web forms was introduced to a Microsoft development community that was very accustomed to Visual Basic 6. Developers using VB6 would drag buttons onto the design surface, double-click the button, and a `Button_Click` event handler method was instantly created for them. This was an incredibly powerful way to create business applications and had everyone excited about Rapid Application Development (RAD) tools. When developers started using classic ASP, it was quite a step backward from the rich environment they were used to in Visual Basic. For better or worse, Web forms brought that Rapid Application Development experience to the Web.

However, as the Web matured and more and more people came to terms with their own understanding of HTML as well as the introduction of CSS (Cascading Style Sheets) and XHTML, a new Web aesthetic started to emerge. Web forms is still incredibly productive for developers, enabling them to create a Web-based line of business applications very quickly. However, the HTML it generates looks, well, generated and can sometimes offend the sensibilities of those who handcraft their XHTML and CSS sites. Web forms concepts like ViewState and the Postback event model have their place, but many developers want a lower-level alternative that embraces not only HTML but also HTTP itself.

Additionally, the architecture of Web forms also makes testing via the current unit testing tools such as NUnit, MbUnit, and xUnit.NET difficult. ASP.NET Web forms wasn't designed with unit testing in mind, and although a number of hacks can be found on the Web, it's fair to say that Web forms does not lend itself well to test-driven development. ASP.NET MVC offers absolute control over HTML, doesn't deny the existence of HTTP, and was designed from the ground up with an eye towards testability.

ASP.NET MVC Is Totally Different!

Yes, ASP.NET MVC is totally different. That's the whole point. It's built on top of a system of values and architectural principles that is very different from those in Web forms. ASP.NET MVC values extensibility, testability, and flexibility. It's very lightweight and doesn't make a lot of assumptions on how you will use it — aside from the fact that it assumes you appreciate the Model-View-Controller pattern.

Different developers and different projects have different needs. If ASP.NET MVC meets your needs, use it. If it doesn't, don't use it. Either way, don't be afraid.

Why "(ASP.NET > ASP.NET MVC) == True"

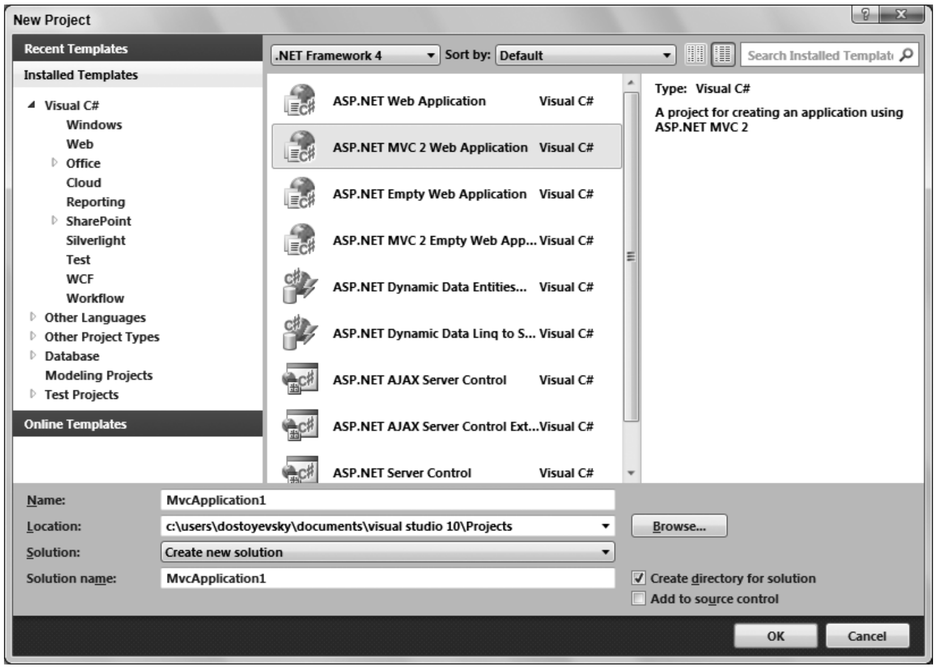
Creating your first MVC application is fairly straightforward. You can use any version of Visual Studio 2010 to create the basic application, including Express, Standard, Professional, or Team Edition.

If you're NOT using Visual Studio 2010, the first order of business is to install the MVC Framework on your 2008 development box. Start at `www.asp.net/mvc` by downloading the latest release. If you like living on the edge, you can often get ASP.NET MVC future releases at `www.codeplex.com/aspnet`.

What you're downloading is a set of Visual Studio project templates that will create your ASP.NET MVC Web application for you. You've used these before — every new ASP.NET Web site and ASP.NET Web application is based on a set of templates. The templates will be installed in Visual Studio, and the reference assemblies will be installed in `C:\Program Files\Microsoft ASP.NET`.

After you've installed ASP.NET MVC, you're ready to create an ASP.NET MVC application using Visual Studio 2008. Though, if you are using Visual Studio 10, then you are going to want to follow these steps:

1. Open Visual Studio 2010 by selecting File ➪ New Project.
2. From the New Project dialog box, shown in the next figure, select ASP.NET MVC 2 Web Application.

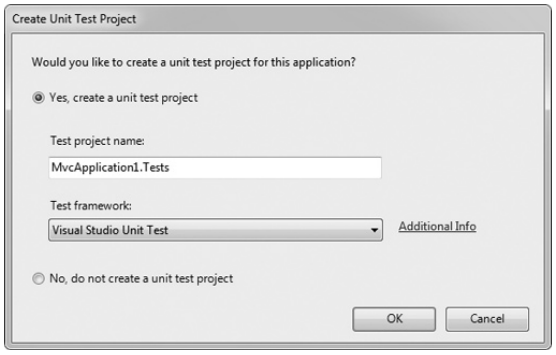

New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Modeling Projects
Test Projects
Online Templates
.NET Framework 4
Sort by: Default
Search Installed Templates
ASP.NET Web Application Visual C#
ASP.NET MVC 2 Web Application Visual C#
ASP.NET Empty Web Application Visual C#
ASP.NET MVC 2 Empty Web App... Visual C#
ASP.NET Dynamic Data Entities... Visual C#
ASP.NET Dynamic Data Linq to S... Visual C#
ASP.NET AJAX Server Control Visual C#
ASP.NET AJAX Server Control Ext... Visual C#
ASP.NET Server Control Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project for creating an application using ASP.NET MVC 2

Name: MvcApplication1
Location: c:\users\dostoyevsky\documents\visual studio 10\Projects
Solution: Create new solution
Solution name: MvcApplication1
[✓] Create directory for solution
[ ] Add to source control

3. Pick your project name and where it's going to live on disk, and click OK. The Create Unit Test Project dialog appears, as shown in the next figure.



Create Unit Test Project

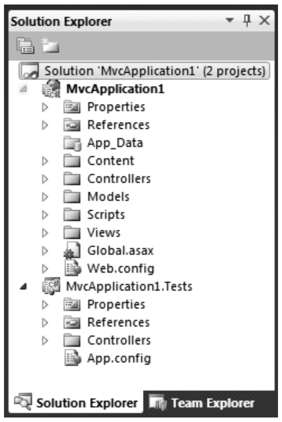
Would you like to create a unit test project for this application?

(●) Yes, create a unit test project

Test project name: MvcApplication1.Tests

Test framework: Visual Studio Unit Test

4. By default the Test Framework drop-down list includes Visual Studio Unit Test as an option. Select Yes to create a solution that includes both a basic ASP.NET MVC project but also an additional MSTest Unit Test project. If you've installed a third-party unit-testing framework like MbUnit or NUnit, additional options appear in this dialog.
5. Click OK, and a solution with projects that look like that shown in the following figure. Note that, although this is an ASP.NET application, along with a standard class library, it has some additional folders you haven't seen before.



```
Solution Explorer
Solution 'MvcApplication1' (2 projects)
  MvcApplication1
    Properties
    References
    App_Data
    Content
    Controllers
    Models
    Scripts
    Views
    Global.asax
    Web.config
  MvcApplication1.Tests
    Properties
    References
    Controllers
    App.config
```

In fact, the application has quite a few more directories than you might be used to; this is by design. ASP.NET MVC, like other MVC frameworks, relies heavily on the idea that you can reduce effort and code by relying on some basic structural rules in your application. Ruby on Rails expresses this powerful idea very succinctly: *Convention over Configuration*.

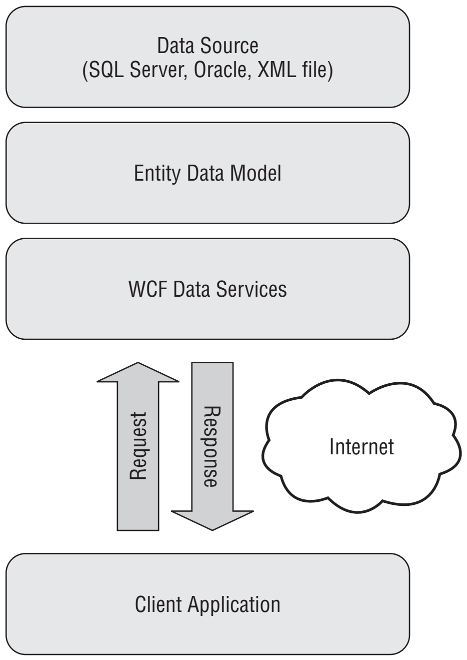
USING WCF DATA SERVICES

Prior to the release of the .NET Framework 4, the release of the .NET Framework 3.5 SP1 provided tremendous focus on the concept of the Entity Data Model (EDM) and what these models bring to your application's reusability. Again, an EDM is an abstract conceptual model of data as you want to represent it in your code.

Another outstanding feature found in this release of the .NET Framework is the WCF Data Services framework. This feature enables you to easily create a cloud interface to your client applications that provides everything from simple read capabilities to a full CRUD model (create, read, update, and delete functions). This feature was previously referred to as ADO.NET Data Services.

When you are working with ASP.NET, there is now a lot more focus on putting work down on the client as well as the server. In the past, much of the development focused on performing as much on the server as possible and delivering completed operations and UI back down to the client simply for viewing. With the release of "smart clients" and AJAX-enabled applications, much of the work is now being offloaded to the client. WCF Data Services makes exposing back-end capabilities over the wire to enable more work to be performed on the client even easier than before.

WCF Data Services works to create a services layer to your back-end data source. Doing so yourself, especially if you are working with a full CRUD model, means a lot of work. WCF Data Services allow you to get a service layer that is URI-driven. The following figure shows the general architecture when you're working with WCF Data Services.



As you can see from this figure, the WCF Data Services layer is not the layer that interacts with the database. Instead, you are working with an EDM layer that is the mapping layer between the data store and the cloud-based interface. When working with your EDM, you are able to use LINQ to SQL and LINQ to Entities.

WCF Data Services allow you to quickly expose interactions with the application's underlying data source as RESTful-based services. The current version of WCF Data Services allows you to work with the datastores using JSON or Atom-based XML. Again, JSON is what ASP.NET AJAX uses for doing out-of-bounds calls to get around doing a complete page refresh.

CREATING YOUR FIRST SERVICE

Figuring out how to build a complete services layer to your database for all create, read, update, and delete functions would take some serious time and thought. However, WCF Data Services makes this task much more feasible, as you will see as you work through this first example.

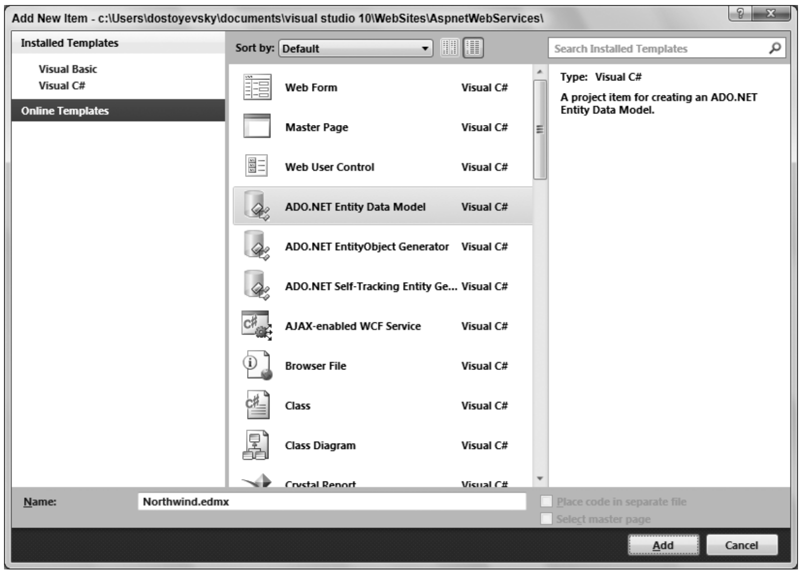
To build a services layer, first create a standard ASP.NET Web Application called Web_ADONETDS in either Visual Basic or C#. Note, you need to use the .NET Framework 3.5 SP1 along with Visual Studio 2008 SP1 or the .NET Framework 4 with Visual Studio 2010 for this example to work.

This, of course, creates a standard Web application that is not different from normal. Because WCF Data Services works from an underlying database, you will need to add one. For this example, add the `NORTHWND.mdf` database. Place this database within the App_Data folder of your project.

Adding Your Entity Data Model

After you have the database in place, you next create an Entity Data Model that WCF Data Services will work with. To do this, right-click on your project and select Add ➪ New Item from the list of options in the provided menu.

The Add New Item dialog appears. As illustrated here, add an ADO.NET Entity Data Model to your project.


Add New Item - c:\Users\dostoyevsky\documents\visual studio 10\WebSites\AspnetWebServices\

Sort by: Default

Type: Visual C#

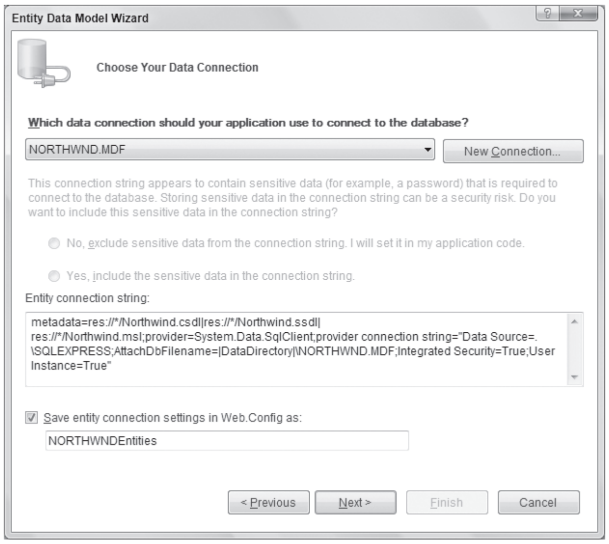
A project item for creating an ADO.NET Entity Data Model.

Name: Northwind.edmx

Installed Templates
Visual Basic
Visual C#
Online Templates
Search Installed Templates
Web Form Visual C#
Master Page Visual C#
Web User Control Visual C#
ADO.NET Entity Data Model Visual C#
ADO.NET EntityObject Generator Visual C#
ADO.NET Self-Tracking Entity Ge... Visual C#
AJAX-enabled WCF Service Visual C#
Browser File Visual C#
Class Visual C#
Class Diagram Visual C#
Crystal Report Visual C#
Place code in separate file
Select master page
Add
Cancel

As shown in the figure, name your ADO.NET Entity Data Model file `Northwind.edmx`. When you create the `Northwind.edmx` file by clicking Add, the Entity Data Model Wizard appears, offering you the option of creating an empty EDM or creating one from a pre-existing database. For this example, choose the option to create one from the pre-existing (Northwind) database. Then click Next in the wizard.

The next screen of the wizard will find your `NORTHWND.mdf` database and pre-define the connection settings and how they will be stored within your application. The next figure presents this second screen of the wizard.


Entity Data Model Wizard

Choose Your Data Connection

Which data connection should your application use to connect to the database?

NORTHWND.MDF

New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

○ No, exclude sensitive data from the connection string. I will set it in my application code.

○ Yes, include the sensitive data in the connection string.

Entity connection string:

```
metadata=res://*/Northwind.csdl|res://*/Northwind.ssdl|
res://*/Northwind.msl;provider=System.Data.SqlClient;provider connection string="Data Source=.
\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;Integrated Security=True;User
Instance=True"
```

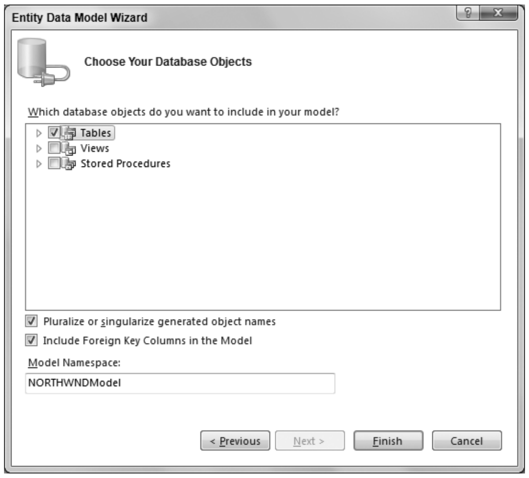
☑ Save entity connection settings in Web.Config as:

NORTHWNDEntities

< Previous | Next > | Finish | Cancel

In the previous screenshot, notice that the connection string and the locations of the mapping details are going to be stored within the `web.config` file. You can also see on this screen that you are naming the instance of the model `NORTHWNDEntities` in the text box at the bottom of the wizard. This name is important to note because you will use it later in this example.

The next screen allows you to select the tables, views, or stored procedures that will be part of the model. For this example, select the check box next to the Table item in the tree view to select all the tables in the database, as shown in the next figure.


Entity Data Model Wizard

Choose Your Database Objects

Which database objects do you want to include in your model?

- ☑ Tables
- ☐ Views
- ☐ Stored Procedures

☑ Pluralize or singularize generated object names

☑ Include Foreign Key Columns in the Model

Model Namespace:

NORTHWNDModel

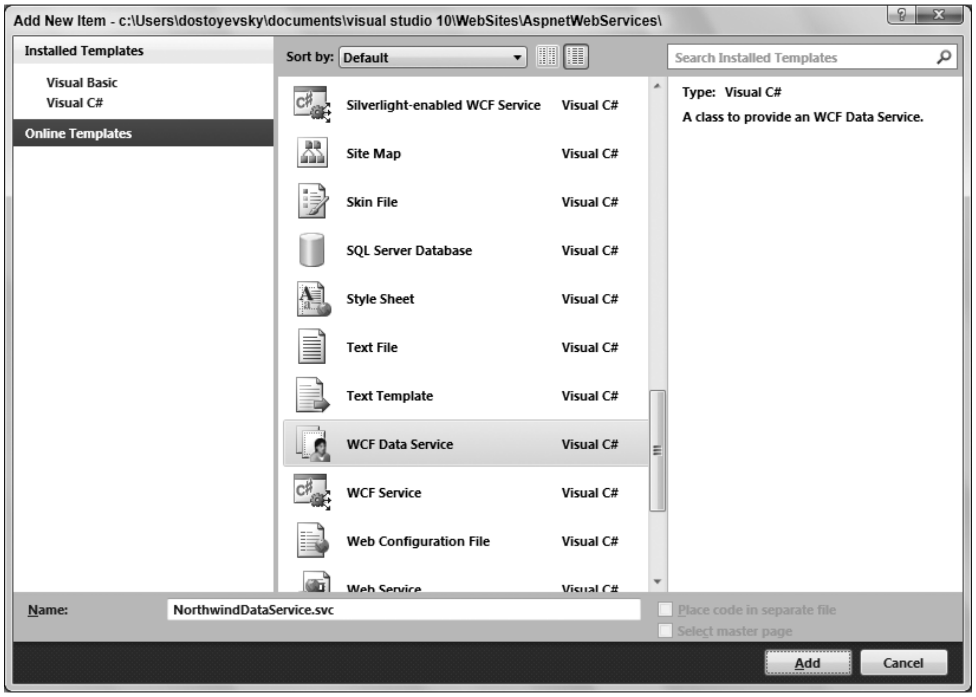
< Previous | Next > | Finish | Cancel

After selecting the Table check box, click the Finish button to have Visual Studio create the EDM for you. You will notice that Visual Studio will create a visual representation of the model for you in the O/R Designer.

If you look at the `Northwind.designer.vb` or the `Northwind.designer.cs` file in your solution, you will see all the generated code for your EDM in place. This class file is named `NORTHWNDEntities`.

Creating the Service

Now that the EDM is in place along with the database, the next step is to add your WCF Data Service. To accomplish this, right-click on your project within the Visual Studio Solution Explorer and select Add ➪ New Item from the provided menu. The Add New Item dialog appears again; select WCF Data Service from the middle section of the provided dialog (see the next figure).


Add New Item - c:\Users\dostoyevsky\documents\visual studio 10\WebSites\AspnetWebServices\
Installed Templates
Visual Basic
Visual C#
Online Templates
Sort by: Default
Search Installed Templates
Silverlight-enabled WCF Service Visual C#
Site Map Visual C#
Skin File Visual C#
SQL Server Database Visual C#
Style Sheet Visual C#
Text File Visual C#
Text Template Visual C#
WCF Data Service Visual C#
WCF Service Visual C#
Web Configuration File Visual C#
Web Service Visual C#
Type: Visual C#
A class to provide an WCF Data Service.
Name: NorthwindDataService.svc
Place code in separate file
Select master page
Add
Cancel

As shown in the figure, name your WCF Data Service `NorthwindDataService.svc`. When done, click the Add button and Visual Studio will then generate a WCF service for you on your behalf. The following listing shows the code of the default service file.

Pro ASP.NET 9780470502204 Listing 31-33: The default .svc file for a WCF Data Service

VB

```
Imports System.Data.Services
Imports System.Linq
Imports System.ServiceModel.Web

Public Class NorthwindDataService
```

```
    ' TODO: replace [[class name]] with your data class name

        Inherits DataService(Of [[class name]])

    ' This method is called only once to initialize service-wide
        policies.
    Public Shared Sub InitializeService(ByVal config As
      IDataServiceConfiguration)
        ' TODO: set rules to indicate which entity sets and service
            operations are visible, updatable, etc.
        ' Examples:
        ' config.SetEntitySetAccessRule("MyEntityset",
             EntitySetRights.AllRead)
        ' config.SetServiceOperationAccessRule("MyServiceOperation",
             ServiceOperationRights.All)
    End Sub

End Class
```

C#

```
using System;
using System.Collections.Generic;
using System.Data.Services;
using System.Linq;
using System.ServiceModel.Web;

namespace Web_ADONETDS
{
    public class NorthwindDataService :
      DataService< /* TODO: put your data source class name here */ >
    {
        // This method is called only once to initialize
             service-wide policies.
        public static void
           InitializeService(IDataServiceConfiguration config)
        {
            // TODO: set rules to indicate which entity sets and
                 service operations are visible, updatable, etc.
            // Examples:
            // config.SetEntitySetAccessRule("MyEntityset",
                 EntitySetRights.AllRead);
            // config.SetServiceOperationAccessRule
                 ("MyServiceOperation", ServiceOperationRights.All);
        }
    }
}
```

The code generated here is the base framework for what you are going to expose through WCF Data Services. It will not work, however, until you accomplish the big TODO that the code specifies. The first step is to put in the name of the EDM instance using the code presented in the following listing.

PRO ASP.NET 9780470502204 Listing 31-34: Changing the WCF Data Service to work with your EDM

VB

```
Public Class NorthwindDataService

     Inherits DataService(Of NORTHWNDEntities)

    ' Code removed for clarity

End Class
```

C#

```
public class NorthwindDataService : DataService<NORTHWNDEntities>
{

   // Code removed for clarity

}
```

Now your application is at a state in which the database, the EDM, and the service to work with the EDM are in place. Upon compiling and pulling up the `NorthwindDataService.svc` file in the browser, you are presented with the following bit of XML:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<service xml:base="http://localhost:4113/NorthwindDataService.svc/"
 xmlns:atom="http://www.w3.org/2005/Atom"
 xmlns:app="http://www.w3.org/2007/app"
 xmlns="http://www.w3.org/2007/app">
  <workspace>
    <atom:title>Default</atom:title>
  </workspace>
</service>
```

If you don't see this XML, then you need to turn off the feed reading capabilities of your IE browser by selecting Tools ➪ Internet Options. From the provided dialog, select the Content tab and, within the Feeds section, click on the Select button. From there, you will be able to uncheck the "Turn on feed reading" check box.

The result of the earlier XML is supposed to be a list of all the available sets that are present in the model, but by default, WCF Data Services locks everything down. To unlock these sets from the model, go back to the `InitializeService()` function and add the following bolded code, as illustrated in the following listing.

Pro ASP.NET 9780470502204 Listing 31-35: Opening up the service for reading from the available tables

Available for download on Wrox.com

VB

```
Imports System.Data.Services
Imports System.Linq
Imports System.ServiceModel.Web

Public Class NorthwindDataService
    Inherits DataService(Of NORTHWNDEntities)

    Public Shared Sub InitializeService(ByVal config _
      As IDataServiceConfiguration)
```

```
            config.SetEntitySetAccessRule("*", EntitySetRights.AllRead)
        End Sub

    End Class
```

C#

```
using System;
using System.Collections.Generic;
using System.Data.Services;
using System.Linq;
using System.ServiceModel.Web;
using System.Web;

namespace Web_ADONETDS
{
    public class NorthwindDataService : DataService<NORTHWNDEntities>
    {
        public static void
          InitializeService(IDataServiceConfiguration config)
        {

            config.SetEntitySetAccessRule("*",
              EntitySetRights.AllRead);
        }
    }
}
```

In this case, every table is opened up to access. Everyone who accesses the tables has the ability to read from them but no writing or deleting abilities. All tables are specified through the use of the asterisk (`*`), and the right to the underlying data is set to read-only through the `EntitySetRights` enum being set to `AllRead`.

Now, when you compile and run this service in the browser, you see the following bit of XML:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<service xml:base="http://localhost:4113/NorthwindDataService.svc/"
 xmlns:atom="http://www.w3.org/2005/Atom"
 xmlns:app="http://www.w3.org/2007/app"
 xmlns="http://www.w3.org/2007/app">
  <workspace>
    <atom:title>Default</atom:title>
    <collection href="Categories">
      <atom:title>Categories</atom:title>
    </collection>
    <collection href="CustomerDemographics">
      <atom:title>CustomerDemographics</atom:title>
    </collection>
    <collection href="Customers">
      <atom:title>Customers</atom:title>
    </collection>
    <collection href="Employees">
      <atom:title>Employees</atom:title>
    </collection>
    <collection href="Order_Details">
      <atom:title>Order_Details</atom:title>
    </collection>
```

```
        <collection href="Orders">
          <atom:title>Orders</atom:title>
        </collection>
        <collection href="Products">
          <atom:title>Products</atom:title>
        </collection>
        <collection href="Region">
          <atom:title>Region</atom:title>
        </collection>
        <collection href="Shippers">
          <atom:title>Shippers</atom:title>
        </collection>
        <collection href="Suppliers">
          <atom:title>Suppliers</atom:title>
        </collection>
        <collection href="Territories">
          <atom:title>Territories</atom:title>
        </collection>
      </workspace>
    </service>
```

The output of this XML is in the AtomPub format, one of the two available formats of XML that are made available from WCF Data Services. The other format is JSON, which is used in the AJAX world. The AtomPub example was retrieved due to the following header being in place:

```
GET /NorthwindDataService.svc/ HTTP/1.1
Accept: application/atom+xml
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/4.0; SLCC2;
.NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0;
.NET4.0C; .NET4.0E; .NET CLR 1.1.4322)
UA-CPU: x86
Accept-Encoding: gzip, deflate
Host: localhost.:4113
Connection: Keep-Alive

            GET /NorthwindDataService.svc/ HTTP/1.1
            Accept: */*
            Accept-Language: en-US,fi-FI;q=0.7,ru-RU;q=0.3
            User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1;
Trident/4.0;
            SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
            Media Center PC 6.0; .NET4.0C; .NET4.0E; .NET CLR 1.1.4322)
            UA-CPU: x86
            Accept-Encoding: gzip, deflate
            Host: localhost:50122
            Connection: Keep-Alive
            Cache-Control: no-cache
            <?xml version="1.0" encoding="utf-8" standalone="yes"?>
            <error
            xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices/
               metadata">
              <code></code>
              <message xml:lang="en-US">Unsupported media type requested.</message>
           </error>
```

Here, `application/atom+xml` is used, and therefore, the AtomPub format is used. Changing the Accept header to read `application/json` will instead give you the following response:

```
{ "d" : {
"EntitySets": [
"Categories", "CustomerDemographics", "Customers", "Employees",
"Order_Details", "Orders", "Products", "Region", "Shippers",
"Suppliers", "Territories"
]
} }
```

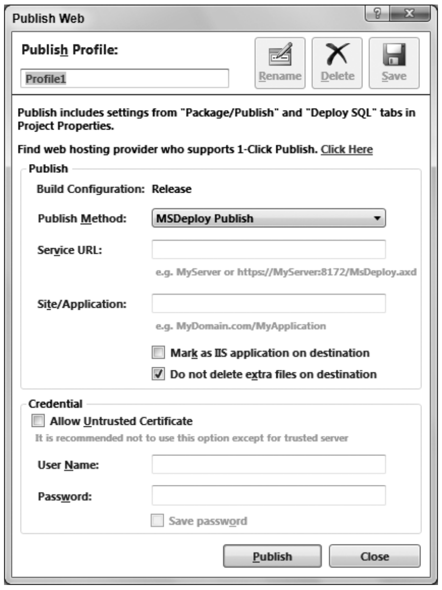
The format of the preceding code is what you would need just for an ASP.NET AJAX page.

BUILDING AN ASP.NET WEB PACKAGE

One of the easiest ways to deploy your Web application is to use the new built-in publishing features from Visual Studio 2010. Behind the scenes, this new capability uses Microsoft's Web Deployment Tool, also known as MSDeploy, which means that if you want to deploy this tool to your server this server must have MSDeploy on the machine for the hydration of your application to actually work.

The package that is created and passed around is an actual physical file — a `.zip` file. This `.zip` file contains everything you need to redeploy your Web application with all of its settings into the new environment. In addition to the `.zip` file, it also includes a manifest file and a command file that are used to initiate the installation of the package on the host machine.

The first way in which this application can be deployed is to use the new 1-Click Publishing capability found in Visual Studio 2010. To do this, right-click on the project within the Visual Studio Solution Explorer and select the Publish option from the provided menu. The Publish Web dialog appears, as shown in the following figure.


Publish Web

Publish Profile: Profile1

Publish includes settings from "Package/Publish" and "Deploy SQL" tabs in Project Properties.

Find web hosting provider who supports 1-Click Publish. Click Here

Publish

Build Configuration: Release

Publish Method: MSDeploy Publish

Service URL:

e.g. MyServer or https://MyServer:8172/MsDeploy.axd

Site/Application:

e.g. MyDomain.com/MyApplication

☐ Mark as IIS application on destination

☑ Do not delete extra files on destination

Credential

☐ Allow Untrusted Certificate

It is recommended not to use this option except for trusted server

User Name:

Password:

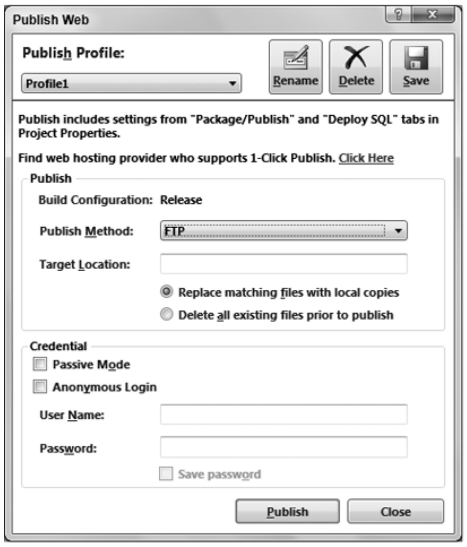
☐ Save password

Because many developers are continually deploying their Web applications to their testing, staging, and production environments, there is a capability to store your deployment options in a profile that you can use again and again. Also notice in the dialog in the previous figure that the build configuration is set to Release for this deployment. This setting comes from what you have set in the toolbar of Visual Studio 2010 before you select the Publish option.

The Publish Web dialog contains the following options as defined in the following table.

SETTING	DESCRIPTION
Profile name	The name of your saved profile. This provides you the ability to reuse your settings as a default for repeated deployments.
Build Configuration	Specifies whether the build compilation will be done in either the Debug or Release mode. You can first establish this setting from the toolbar of Visual Studio.
Publish Method	The method you want to employ for the deployment. The possible options include MSDeploy Publish, FTP, File System, and FPSE.
Service URL	Specifies the location of the MSDeploy on the host server. This URL points to the actual IIS handler that has the MSDeploy capability and will be constructed similar to `http://myhostserver:8172/MsDeploy .axd`.
Site/Application	Specifies the location where your application will be deployed. Here you can specify the site as well as the virtual directory to place the application. An example of this is `MyDomain.com/MyApplication`.
Mark as IIS application on destination	If this option is selected, IIS will treat the endpoint defined in the Site/Application textbox as the application root.
Do not delete extra files on destination	If this option is not selected, the 1-Click Publishing option will first delete everything on the host server location before applying the files and settings.
Allow Untrusted Certificate	If this option is selected, you will trust certificates that are self-pub-lished or certificates that don't match the URL of the destination.
User Name	The username used for IIS connections.
Password	The password used for IIS connections.
Save password	Specifies to the Visual Studio publishing feature whether to save the password in the profile.

Besides using the MSDeploy option, you will find that the other options provided through the new Publish Web dialog are even simpler. The following figure shows you the settings for the other three deployment options provided.


Publish Web

Publish Profile: Profile1

Publish includes settings from "Package/Publish" and "Deploy SQL" tabs in Project Properties.

Find web hosting provider who supports 1-Click Publish. Click Here

Publish

Build Configuration: Release

Publish Method: FTP

Target Location:

(●) Replace matching files with local copies

( ) Delete all existing files prior to publish

Credential

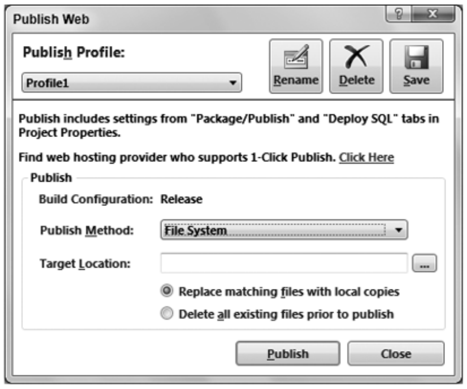
[ ] Passive Mode

[ ] Anonymous Login

User Name:

Password:

[ ] Save password


Publish Web

Publish Profile: Profile1

Publish includes settings from "Package/Publish" and "Deploy SQL" tabs in Project Properties.

Find web hosting provider who supports 1-Click Publish. Click Here

Publish

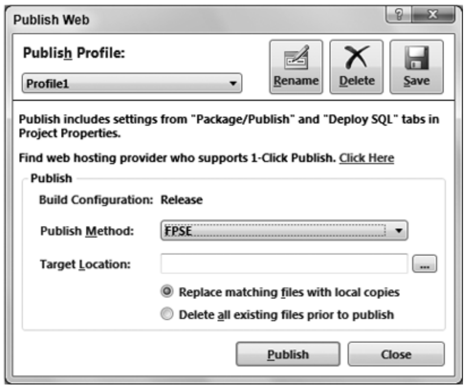
Build Configuration: Release

Publish Method: File System

Target Location:

(●) Replace matching files with local copies

( ) Delete all existing files prior to publish


Publish Web

Publish Profile: Profile1

Publish includes settings from "Package/Publish" and "Deploy SQL" tabs in Project Properties.

Find web hosting provider who supports 1-Click Publish. Click Here

Publish

Build Configuration: Release

Publish Method: FPSE

Target Location:

(●) Replace matching files with local copies

( ) Delete all existing files prior to publish

In this figure you can see some standard settings for doing deployments using FTP, the file system, or FrontPage Server Extensions.

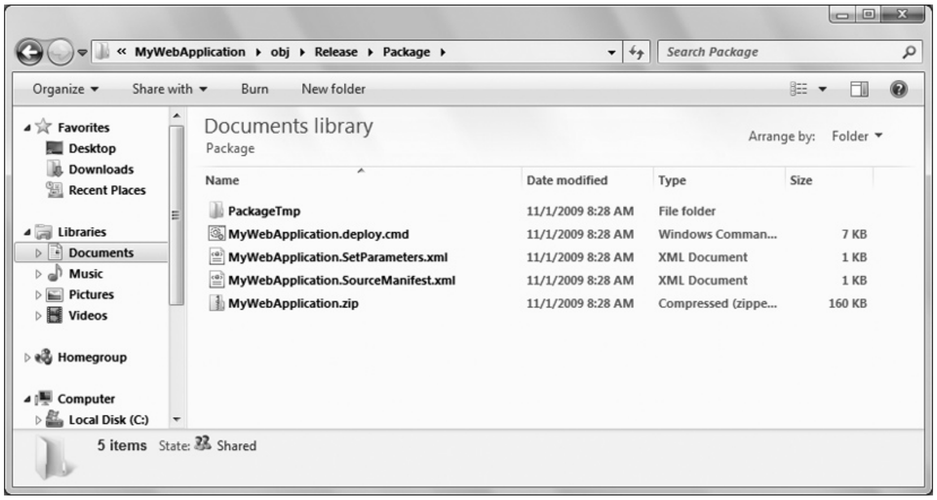
The interesting thing with the MSDeploy capabilities that Visual Studio 2010 now works with is that instead of just connecting to a remote MSDeploy handler and running your deployment real-time, you can also create an MSDeploy package that can be run at any time.

Visual Studio 2010 now includes the ability to create these packages that can then be e-mailed or by other means provided to someone else to run on their systems. To create a Web deployment package, right-click on the project within the Visual Studio Solution Explorer and choose the Create Package option from the provided menu.

When you select the Create Package option, the progress of the package creation appears in the Visual Studio status bar. After you're notified that the Publish succeeded, you can find the entire package in your application's obj folder, such as

```
C:\Users\Evjen\Documents\Visual Studio 10\
 Projects\MyWebApplication\MyWebApplication\obj\Release\Package
```

Within this folder are all the files that constitute the package. The following figure presents these files.



Name	Date modified	Type	Size
PackageTmp	11/1/2009 8:28 AM	File folder	
MyWebApplication.deploy.cmd	11/1/2009 8:28 AM	Windows Comman...	7 KB
MyWebApplication.SetParameters.xml	11/1/2009 8:28 AM	XML Document	1 KB
MyWebApplication.SourceManifest.xml	11/1/2009 8:28 AM	XML Document	1 KB
MyWebApplication.zip	11/1/2009 8:28 AM	Compressed (zippe...	160 KB

All of these files constitute the package that you can use in the deployment process.

- The `MyWebApplication.deploy.cmd` file is what your infrastructure team would use to run on the host machine (which has MSDeploy) to install the application and all the settings that are required for the application.
- The other files, `MyWebApplication.SetParameters.xml` and `MyWebApplication.SourceManifest.xml`, are used to define the settings used in the installation process.
- The final file, `MyWebApplication.zip`, contains the contents of the application.

With this package, passing another team the installation required for your application is easy. A nice example of this usage is if you are deploying to a Web farm because multiple deployments need to occur and you want these deployments to all be the same. Running your installations from this single package ensures this similarity.

The other nice advantage to this package system is that it allows you to save your previous deployments, and if you need to go back and verify deployments, you can easily grab hold of a saved package. It also makes rollbacks easy for your deployment process.

Join the discussion @ p2p.wrox.com

wrox

Wrox Programmer to Programmer™

FULL COLOR

WPF Programmer's Reference

Windows® Presentation Foundation with C# 2010 and .NET 4

Rod Stephens

WPF

PROGRAMMER'S REFERENCE

Windows Presentation Foundation with C# 2010 and .NET 4

CODE-BEHIND FILES

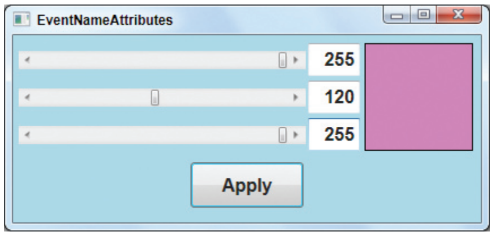
Whenever you add a window to a project, Expression Blend or Visual Studio adds a corresponding code-behind file. The file has the same name as the XAML file with the extra extension *cs* (for C#) or *vb* (for Visual Basic). For example, when you create a new WPF project in C#, the project includes the initial window in the file Window1.xaml and the corresponding code-behind file Window1.xaml.cs.

If you're using Visual Studio, then you can edit the code-behind file and take full advantage of Visual Studio's powerful code-editing features such as keyword highlighting and IntelliSense.

EXAMPLE CODE

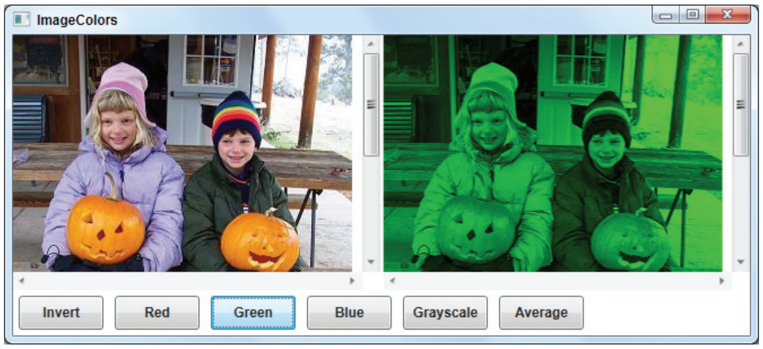
The source code, which is available for download on the book's web site at `www.wrox.com`, includes several example applications that all do the same thing but use different techniques for attaching code-behind to the user interface.

This figure shows one of the programs in action. You can drag the scrollbars or enter values in the textboxes to set the red, green, and blue components of the color displayed on the right. If you click on the Apply button, the program sets the window's background color to match the current sample.



YOU CAN DOWNLOAD THE CODE FOUND IN THIS SECTION. VISIT **WROX.COM** AND SEARCH FOR ISBN 9780470477229

The ImageColors example program, which is shown in the next figure, demonstrates all of the different techniques for attaching UI elements to code-behind in a single program. The program's different buttons are attached to event handlers in different ways. Download the C# or Visual Basic version of this program from the book's web site to see the details.



EVENT NAME ATTRIBUTES

The first way to attach code-behind to a control's events uses an attribute in the XAML code. The attribute's name is the same as the event that you want to catch. Its value is the name of the event handler in the code-behind.

For example, the following XAML code defines a `Button` named `btnApply`. The `Click` attribute indicates that the routine `btnApply_Click` catches the control's `Click` event handler.

```
<Button Content="Apply" IsDefault="True"
 Name="btnApply" Click="btnApply_Click"/>
```

Unfortunately, if you simply add the event handler declaration to the XAML code, the program won't run. If you try to execute it in Expression Blend or Visual Studio, you'll receive an error that says your program doesn't contain a definition for the event handler routine.

The following C# code handles the event declared in the previous XAML code:

```
private void btnApply_Click(object sender, RoutedEventArgs e)
{
  this.Background = borSample.Background;
}
```

This code catches the `btnApply Button`'s `Click` event and sets the window's `Background` property to the value used by the `borSample` control's `Background` property (more details on this shortly).

Your code doesn't need to do anything special to wire up the event handler to catch the event — Expression Blend and Visual Studio take care of that for you.

The following XAML fragment shows how the EventNameAttributes example program determines which event handlers catch the key events raised by the program's Apply button, and the scrollbar and textbox that control the color's red component. The code for the other scrollbars and textboxes is similar.

Available for download on Wrox.com

```
<ScrollBar Orientation="Horizontal" Minimum="0" Maximum="255"
 Grid.Row="0" Grid.Column="0" Value="255"
 Name="scrRed" ValueChanged="scrRed_ValueChanged"/>

<TextBox Margin="2" Grid.Row="0" Grid.Column="1" Text="255"
 HorizontalContentAlignment="Right" VerticalContentAlignment="Center"
 Name="txtRed" TextChanged="txtRed_TextChanged" />

<Button Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="3" Width="100" Height="40"
 Content="Apply" IsDefault="True"
 Name="btnApply" Click="btnApply_Click"/>
```

WPF Programmer's Reference 9780470477229 EventNameAttributes

The following C# fragment shows how the EventNameAttributes program responds when you use the first scrollbar or textbox to change the red color component, and when you click on the Apply button. The event handlers for the other scrollbars and textboxes are similar.

Available for download on Wrox.com

```
// A ScrollBar has been changed. Update the corresponding TextBox.
private void scrRed_ValueChanged(object sender,
 RoutedPropertyChangedEventArgs<double> e)
{
  if (txtRed == null) return;
  txtRed.Text = ((int)scrRed.Value).ToString();
  ShowSample();
}

// A TextBox has been changed. Update the corresponding ScrollBar.
private void txtRed_TextChanged(object sender,
 TextChangedEventArgs e)
{
  // Keep the value within bounds.
  int value;
  try {
    value = int.Parse(txtRed.Text);
  } catch {
    value = 0;
  }

  if (value < 0) value = 0;
  if (value > 255) value = 255;
  txtRed.Text = value.ToString();

  if (scrRed == null) return;
  scrRed.Value = value;
  ShowSample();
}

// Display a sample of the color.
private void ShowSample()
{
  if (borSample == null) return;

  byte r, g, b;
  try {
    r = byte.Parse(txtRed.Text);
  } catch {
    r = 0;
  }
  try {
    g = byte.Parse(txtGreen.Text);
  } catch {
    g = 0;
  }
  try {
    b = byte.Parse(txtBlue.Text);
  } catch {
    b = 0;
  }
  borSample.Background = new SolidColorBrush(Color.FromRgb(r, g, b));
}
```

WPF Programmer's Reference 9780470477229 EventNameAttributes

RESOURCES

One of the most important concepts in any kind of programming is code reuse. Subroutines, functions, scripts, classes, inheritance, loops, and many other programming constructs let you reuse code in one way or another. For example, if your program needs to perform the same task in many places, you can create a subroutine that performs the task and then call it from those places.

Just as routines and functions let you reuse code in languages such as C# and Visual Basic, *resources* let you reuse XAML code. They let you define values that you can then use from many places in a XAML file.

Defining Resources

Creating and using a simple resource is fairly easy. (In fact, it's easier to understand from an example than from a description, so, if you have trouble understanding the following explanation, look at the example text and then read the explanation again.)

To define a resource, add a `Resources` property element to an object such as a `Window`, `Grid`, or other container. Inside that element, place the resources that you will want to use later. Each resource is an element such as an object (e.g., a `LinearGradientBrush`, `Thickness`, or `Label`) or a simple value (e.g., a string or an integer).

You must give each resource a unique `x:Key` attribute value to identify it.

For example, the following XAML code defines a `Resources` element that contains `RadialGradientBrush` named `brButton` and a `BitmapEffectGroup` named `bmeButton`. This `Resources` element is contained in the file's `Window` element (hence the tag `Window.Resources`).

Available for download on Wrox.com

```
<Window.Resources>
  <LinearGradientBrush x:Key="brButton" StartPoint="0,0" EndPoint="0,1">
    <GradientStop Color="Red" Offset="0"/>
    <GradientStop Color="White" Offset="0.5"/>
    <GradientStop Color="Blue" Offset="1"/>
  </LinearGradientBrush>
  <BitmapEffectGroup x:Key="bmeButton">
    <DropShadowBitmapEffect/>
  </BitmapEffectGroup>
</Window.Resources>
```

WPF Programmer's Reference 9780470477229 ButtonResources

Any object contained (directly or indirectly) in the object that holds the `Resources` element can use the resources. In the previous code, the `Window` contains the `Resources` element, so any object in the `Window` can use its resources.

After you have defined a simple property resource such as this one, you can use it with property attribute syntax. The value you give the attribute should have the format `{StaticResource` *`resource_name`*`}`, where you replace *`resource_name`* with the name of the resource.

For example, the following XAML code defines a `Button`. It sets the `Button`'s `Background` property to the `brButton` resource and its `BitmapEffect` property to the `bmeButton` resource.

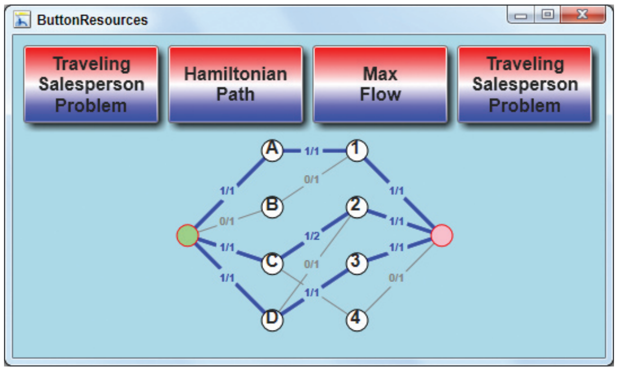
Available for download on Wrox.com

```
<Button Margin="4" Width="120"
 Background="{StaticResource brButton}"
 BitmapEffect="{StaticResource bmeButton}">
  <TextBlock TextAlignment="Center" Margin="4">
    Traveling<LineBreak/>Salesperson<LineBreak/>Problem
  </TextBlock>
</Button>
```

WPF Programmer's Reference 9780470477229 ButtonResources

The ButtonResources example program shown here displays several buttons that use similar code to define their backgrounds and bitmap effects. The only difference between the buttons is the contents of their `TextBlocks`.

Using resources has several advantages. Because the buttons shown next use the same resources, they are guaranteed to have a consistent appearance (at least as far as the `Background` and `BitmapEffect` properties are concerned).



Using resources simplifies the code by converting relatively complex property elements into simpler attribute elements.

The resources also allow you to easily change the appearance of all of the buttons at once.

Merged Resource Dictionaries

A resource dictionary lets several controls share the same values. *Merged resource dictionaries* let several windows or even applications share the same resource values.

To merge a resource dictionary, create a XAML file that has a `ResourceDictionary` object as its root element. Give the element namespace attributes as you would a `Window`, and place resources inside the dictionary.

The following code shows part of a resource dictionary file named *RectangleResources.xaml*. This file defines resources that the MultiWindowResources example program uses to define `Rectangle` properties. (To save space, I've omitted some of the resources.)

Available for download on Wrox.com

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <!--  Resource dictionary entries should be defined here.  -->
  <sys:Double x:Key="rectWidth">140</sys:Double>
  <sys:Double x:Key="rectHeight">50</sys:Double>
  <sys:Double x:Key="rectRadX">5</sys:Double>
  <sys:Double x:Key="rectRadY">20</sys:Double>
  ... More resources omitted ...
</ResourceDictionary>
```

WPF Programmer's Reference 9780470477229 MultiWindowResources

The following code shows how the MultiWindowResources program uses this resource dictionary. The Window1 XAML file uses a `ResourceDictionary.MergedDictionaries` element that contains a reference to the dictionary file.

Available for download on Wrox.com

```
<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="RectangleResources.xaml"/>
```

```
        </ResourceDictionary.MergedDictionaries>
      </ResourceDictionary>
    </Window.Resources>
```

WPF Programmer's Reference 9780470477229 MultiWindowResources

The following code shows how the `Window` uses these resources. This code creates a `Rectangle` that refers to the resources defined in the external dictionary.

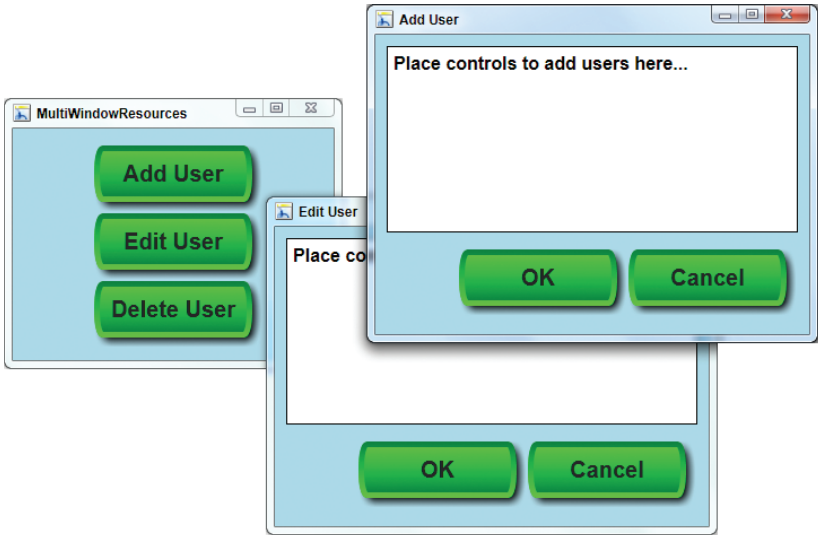
Available for download on Wrox.com

```
<Grid MouseDown="rectAddUser_MouseDown">
  <Rectangle
    Margin="{StaticResource rectMargin}"
    Width="{StaticResource rectWidth}"
    Height="{StaticResource rectHeight}"
    RadiusX="{StaticResource rectRadX}"
    RadiusY="{StaticResource rectRadY}"
    Fill="{StaticResource rectFill}"
    Stroke="{StaticResource rectStroke}"
    StrokeThickness="{StaticResource rectStrokeThickness}"
    BitmapEffect="{StaticResource rectBitmapEffect}"
  />
  <Label HorizontalAlignment="Center" VerticalAlignment="Center"
    Content="Add User"
    FontSize="{StaticResource rectFontSize}"
    FontWeight="{StaticResource rectFontWeight}"
  />
</Grid>
```

WPF Programmer's Reference 9780470477229 MultiWindowResources

The MultiWindowResources example program uses similar code to include the resource dictionary and refer to its resources in all four of its `Windows`. Because every form refers to the same resources, they all have a common appearance. If you decide to change the appearance, you can simply modify the resource dictionary, and all of the forms will pick up the changes automatically.

This figure shows the MultiWindowResources program displaying several forms that have similar button-like rectangles.



In addition to allowing multiple forms or applications to use the same resources, merged dictionaries allow a single window to load more than one resource dictionary. This can be handy if you want to use separate dictionaries to hold different groups of resources. For example, you might have separate dictionaries to hold button resources, label resources, and list resources.

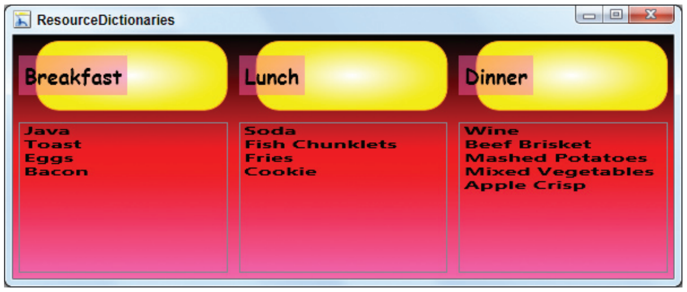
You can also use multiple resource dictionaries in the same application to switch easily between different appearances. For example, the ResourceDictionaries example program uses the following code to merge two resource dictionaries, ResBlue.xaml and ResRed.xaml:

Available for download on Wrox.com

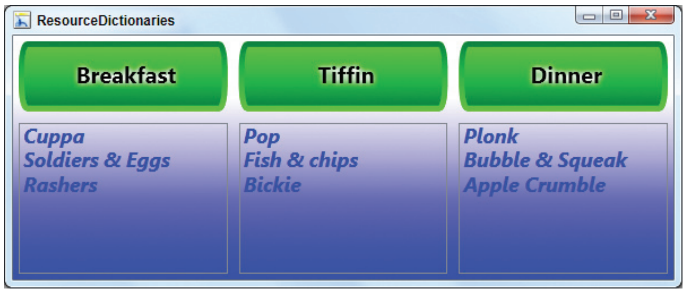
```
<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="ResBlue.xaml"/>
      <ResourceDictionary Source="ResRed.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>
```

WPF Programmer's Reference 9780470477229 ResourceDictionaries

Because the file ResRed.xaml is loaded second, the resource values in that file take precedence. This figure shows the ResourceDictionaries program using the values in this resource dictionary.



By simply switching the order of these two resource dictionaries, you can completely change the application's appearance. The following figure shows the ResourceDictionaries program using the values in the ResBlue.xaml resource dictionary.



The later section "Skins" explains how to switch an application's appearance in a similar manner at run time.

STYLES AND PROPERTY TRIGGERS

We just looked at *resources* — named property values that you can assign to controls to give them a common appearance and behavior.

Unfortunately, resources often lead to long and complicated code. If a group of controls shares many property values, then converting the properties into resources requires a lot of repetition using the verbose `StaticResource` or `DynamicResource` keywords.

A *Style* is a special kind of resource that lets you extract this redundancy and centralize it, much as other resources let you centralize property values. `Styles` let you define packages of property values that you can apply to a control all at once instead of applying individual resource values to the control one at a time.

Simplifying Properties

A style packages property values that should be set as a group. It begins with a `Style` element contained in a resource dictionary. You can place the `Style` in any resource dictionary depending on how widely you want it to be available. For example, if you want a `Style` to be visible to the entire project, place it in the `Application.Resources` section in the App.xaml file; if you want the `Style` visible to every control on a window, place it in the `Window.Resources` section; and if you want it to be visible only to controls inside a `StackPanel`, place it inside the `StackPanel.Resources` section.

A style can have an `x:Key` attribute to give it a name and a `TargetType` attribute to indicate the kind of control to which it should apply. (The next section says more about these attributes.)

Inside the `Style` element, `Setter` and `EventSetter` elements define the style's property values and event handlers, respectively.

A `Setter` sets a property's value. It takes two attributes — `Property` and `Value` — that give the property to be set and the value it should take. The value can be a simple attribute value like Red or 7, or it can be an element attribute specifying something more complex like a brush.

The RedRectangles example program uses the following code to define a `Style` named `RedRectStyle` that applies to `Rectangles`. The `Stroke`, `StrokeThickness`, `Width`, `Height`, and `Margin setters` use property attribute syntax to provide simple values, while the `Fill setter` uses property element syntax to set its value to a `RadialGradientBrush` object.

Available for download on Wrox.com

```
<Style x:Key="RedRectStyle" TargetType="Rectangle">
  <Setter Property="Stroke" Value="Red"/>
  <Setter Property="StrokeThickness" Value="5"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="50"/>
  <Setter Property="Margin" Value="5"/>
  <Setter Property="Fill">
    <Setter.Value>
      <RadialGradientBrush>
        <GradientStop Color="Red" Offset="0"/>
        <GradientStop Color="White" Offset="1"/>
      </RadialGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

WPF Programmer's Reference 9780470477229 RedRectangles

You can use a style just as you use any other resource. Simply set the "Style" property to the style's key.

The RedRectangles example program uses the following code to display three `recRectangles` that use the `RedRectStyle Style`:

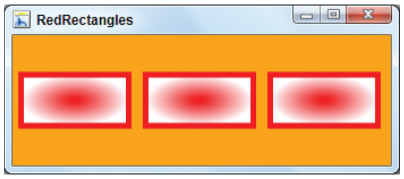
Available for download on Wrox.com

```
<StackPanel Orientation="Horizontal">
  <Rectangle Style="{StaticResource RedRectStyle}"/>
  <Rectangle Style="{StaticResource RedRectStyle}"/>
  <Rectangle Style="{StaticResource RedRectStyle}"/>
</StackPanel>
```

WPF Programmer's Reference 9780470477229 RedRectangles

Notice how much the style simplifies this code. Without the the style, each `Rectangle` would need to have its own `Stroke`, `StrokeThickness`, `Width`, `Height`, `Margin`, and `Fill` properties.

This figure shows the RedRectangles program in action.



Unnamed Styles

In all of the styles described in the previous sections, the `Style` element included an `x:Key` attribute to give the style a name. Then controls used the `StaticResource` keyword to refer to the name.

In contrast, if you omit the attribute and include a `TargetType`, then the resulting *unnamed style* applies to all controls of the `TargetType` within the style's scope. For example, the following `Style` applies to every `Button` within the style's scope, so every `Button` appears with a yellow background and a red foreground:

```
<Style TargetType="Button">
  <Setter Property="Background" Value="Yellow" />
  <Setter Property="Foreground" Value="Red" />
</Style>
```

The UnnamedStyles example program shown in the next figure defines four unnamed styles. The following list describes the properties that those styles set for their classes:

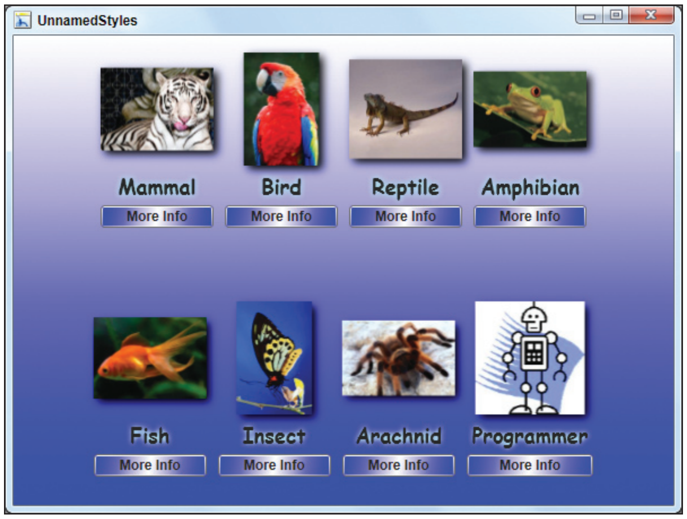
- `Image` — `Width` = 100, `Height` = 100, `Stretch` = Uniform, `BitmapEffect` = drop shadow
- `Label` — `FontFamily` = Comic Sans MS, `FontSize` = 18, `FontWeight` = bold, `HorizontalAlignment` = center, `BitmapEffect` = light blue outer glow
- `Button` — `FontFamily` = Arial, `FontSize` = 12, `Height` = 20, `Background` = blue and white linear gradient
- `StackPanel` — `Margin` = 5

The following code shows how the program defines its `Style` for `Images`. You can download the example program from the book's web site to see how the other `Styles` are defined.

Available for download on Wrox.com

```
<Style TargetType="Image">
  <Setter Property="Width" Value="100" />
  <Setter Property="Height" Value="100" />
  <Setter Property="Stretch" Value="Uniform" />
  <Setter Property="BitmapEffect">
    <Setter.Value>
      <DropShadowBitmapEffect/>
    </Setter.Value>
  </Setter>
</Style>
```

WPF Programmer's Reference 9780470477229 UnnamedStyles



Triggers

The styles described so far always apply their property values to their controls. If a style has a setter that makes the `Background` property green, then every control that uses the style gets a green background.

Styles can also define *triggers*, objects that apply setters or start other actions only under certain conditions. For example, a `trigger` might change a `Label`'s color, font, or size when the mouse was over it to provide some visual feedback to the user.

A *property trigger* performs its actions when a specific property has a certain value. For example, a property trigger could invoke one or more `Setters` if a `TextBox`'s `Text` property had the value *Test*. (They are called property triggers because it's a property value that triggers the action.)

When the property no longer has the triggering value, the `Trigger` deactivates, and the control's original property value returns. For example, when the user moves the mouse off the `Label`, the `Label` returns to its original color, font, and size.

To make a property `Trigger`, create a `Style` and give it a `Style.Triggers` property element. Inside the `Style.Triggers` section, you can add `Trigger` elements.

Use each `Trigger`'s `Property` and `Value` attributes to indicate the property value that should activate the `Trigger`. Inside the `Trigger` element, add whatever `Setters` you want the `Trigger` to execute.

IsMouseOver Triggers

One common type of property trigger takes action when the user moves the mouse over something. These triggers execute their `Setters` when the `IsMouseOver` property is `True`.

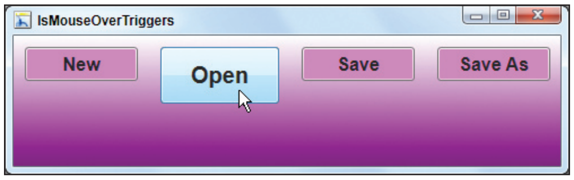
The following XAML code defines an unnamed `Button Style`. It sets some basic values and then defines a `Trigger` that executes when the `IsMouseOver` property is `True`. When the mouse moves over the `Button`, the `Trigger` makes the `Button` bigger and makes its font larger and bold.

Available for download on Wrox.com

```
<Style TargetType="Button">
  <Setter Property="VerticalAlignment" Value="Top"/>
  <Setter Property="Margin" Value="10"/>
  <Setter Property="Background" Value="Violet"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="30"/>
  <Setter Property="FontSize" Value="16"/>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Width" Value="105"/>
      <Setter Property="Height" Value="50"/>
      <Setter Property="FontSize" Value="20"/>
      <Setter Property="FontWeight" Value="Bold"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

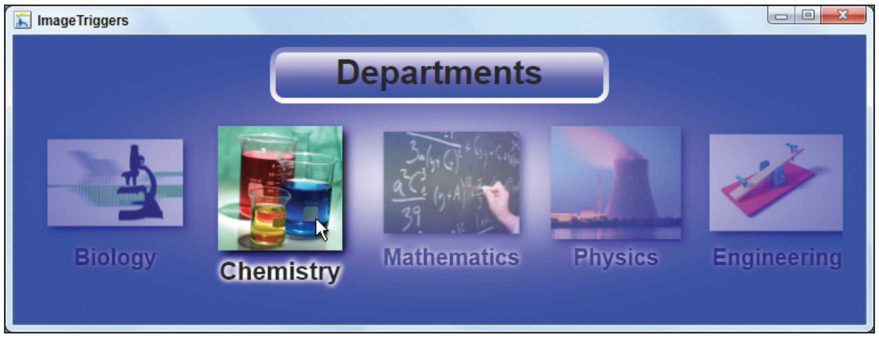
WPF Programmer's Reference 9780470477229 IsMouseOverTriggers

The IsMouseOverTriggers example program shown here uses this code to provide feedback when the mouse is over a button.



Setting Opacity

The ImageTriggers example program shown in the next figure uses `Triggers` to highlight the image under the mouse.



The program uses several `Styles` to make building the list of images easier. They're rather long, so they aren't shown here. You can download the example program from the book's web site and take a look if you like.

The bottom of the program's window contains a series of `StackPanel` controls, each of which holds an `Image` and a `Label`. The following code shows how the program displays the controls for the Engineering department on the right:

Available for download on Wrox.com

```
<StackPanel Style="{StaticResource DepartmentStackPanel}">
  <Image Source="Engineering.jpg" />
  <Label Style="{StaticResource ImageLabelStyle}" Content="Engineering"/>
</StackPanel>
```

WPF Programmer's Reference 9780470477229 ImageTriggers

The key is the `DepartmentStackPanel Style` defined in the following code:

Available for download on Wrox.com

```
<ScaleTransform x:Key="BigScale" ScaleX="1.1" ScaleY="1.1"/>

<Style x:Key="DepartmentStackPanel" TargetType="StackPanel">
  <Setter Property="Margin" Value="10"/>
  <Setter Property="Opacity" Value="0.5"/>
  <Setter Property="Background" Value="Transparent"/>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Opacity" Value="1"/>
      <Setter Property="LayoutTransform"
       Value="{StaticResource BigScale}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

WPF Programmer's Reference 9780470477229 ImageTriggers

EVENT TRIGGERS AND ANIMATION

An event trigger lets you detect when an event occurs. Animation lets you change property values in a series of smoothly varying steps. Together, for example, you can make a `Button` detect the `MouseOver` event and use animation to enlarge the `Button`, providing graphical feedback similar to the experience provided by the previous property event.

Event Triggers

A *property trigger* takes action when a control's property has a certain value.

In contrast, an *event trigger* takes action when a control event occurs. Rather than acting when the `IsMouseOver` property is `True`, an event trigger might take action when the `MouseEnter` event fires, indicating that the mouse moved over the control. Another event could take some other action when the `MouseLeave` event fires, which occurs when the mouse moves off the control.

To make an event trigger, make a `Triggers` element, either in a control's definition or in a `Style`. Inside the Triggers section, add `EventTrigger` elements.

Each `EventTrigger` should have a `RoutedEvent` attribute that indicates the event that executes the trigger. The `RoutedEvent` should include the name of the class raising the event (`Button`, `Label`, `Grid`, etc.), followed by a dot and the name of the event. For example, the `RoutedEvent` name for a `Button`'s `Click` event is *Button.Click*.

Inside the trigger, place an `Actions` section to hold the actions that you want the trigger to perform. Inside the `Actions` element, you can place code to run animations by executing storyboards.

Normally an event trigger's actions use a `BeginStoryboard` element to invoke a `Storyboard` object. Later sections describe storyboards in detail, but for now, assume that you have defined a `Storyboard` named `sbBigScale` in a window's Resources section. Then the following code shows how the previous `Button` could invoke the `Storyboard`:

Available for download on Wrox.com

```
<Button Width="100" Height="50" Content="Click Me">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
        <BeginStoryboard
         Storyboard="{StaticResource sbBigScale}"/>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

WPF Programmer's Reference 9780470477229 GrowingButtons

The `BeginStoryboard` object's `Storyboard` attribute gives the `Storyboard` that it should execute. This example uses the `StaticResource` to run the `Storyboard` defined in the window's resource dictionary.

Storyboards in Styles

Building a storyboard right into a control makes its definition quite long and requires some pretty deeply nested elements, so you may want to define storyboards in resources to simplify the code. If possible, you might want to put the triggers in a style to simplify the `Button`'s code. You can then make the `Style` simpler by placing the `Storyboard` in its own resource.

For example, the following code defines a `Button Style` that includes event triggers. When the `Button` raises its `MouseEnter` event, the code runs the `sbBigSize Storyboard`. When the `Button` raises its `MouseLeave` event, the code runs the `sbSmallSize Storyboard`.

Available for download on Wrox.com

```
<Style x:Key="btnEventSbSize" TargetType="Button">
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="50"/>
  <Style.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
        <BeginStoryboard Storyboard="{StaticResource sbBigSize}"/>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.MouseLeave">
```

```
          <EventTrigger.Actions>
            <BeginStoryboard Storyboard="{StaticResource sbSmallSize}"/>
          </EventTrigger.Actions>
        </EventTrigger>
      </Style.Triggers>
    </Style>
```

WPF Programmer's Reference 9780470477229 GrowingButtons

A program could use the following code to define a button that uses this style:

```
<Button Style="{StaticResource btnEventSbSize}"
 Content="New Size"/>
```

Storyboards

Now that you understand how to start a storyboard, it's time to learn how to build one. A *storyboard* is an object that defines a timeline for one or more animations.

An *animation* manages the transition of a property from one value to another over time. An animation can include a start time indicating when the transition should begin and a duration telling how long the animation should last.

The animation classes let you specify *what* you want done and then ignore *how* it is done. You say you want a property changed to a certain value over the next 3 seconds, and the animation classes figure out how to do it.

Specific kinds of animations handle changes to properties of different data types such as `Double`, `Color`, `String`, `Thickness`, and many others.

The following code shows a simple `Storyboard` containing five `DoubleAnimation` objects. This is the `Storyboard` that the CentralizedTriggers program uses to move the first thumbnail into the viewing area.

Available for download on Wrox.com

```
<Storyboard x:Key="sbImg1">
  <DoubleAnimation Duration="0:0:0.5" To="120"
        Storyboard.TargetName="img1"
        Storyboard.TargetProperty="(Canvas.Left)"/>
  <DoubleAnimation Duration="0:0:0.5" To="0"
        Storyboard.TargetName="img1"
        Storyboard.TargetProperty="(Canvas.Top)"/>
  <DoubleAnimation Duration="0:0:0.5" To="485"
        Storyboard.TargetName="img1"
        Storyboard.TargetProperty="Width"/>
  <DoubleAnimation Duration="0:0:0.5" To="400"
        Storyboard.TargetName="img1"
        Storyboard.TargetProperty="Height"/>
  <DoubleAnimation Duration="0:0:0.5" To="1"
        Storyboard.TargetName="img1"
        Storyboard.TargetProperty="Opacity"/>
</Storyboard>
```

WPF Programmer's Reference 9780470477229 CentralizedTriggers

The first animation makes the `Canvas.Left` property of the control `img1` change from its current value (whatever it is) to 120 over a 0.5-second interval. The second animation similarly changes the control's `Canvas.Top` property to 0 over the same 0.5 second.

The third and fourth animations change the `img1` control's `Width` and `Height` properties to 485 and 400, respectively. The final animation changes the control's `Opacity` to 1 so it is completely opaque.

Animation Types

WPF provides many classes for animating properties with different data types. The following table lists data types that have corresponding animation classes. For example, you can use the `Int32Animation` class to animate a 32-bit integer property.

`Boolean`	`Int16`	`Point3D`	`String`
`Byte`	`Int32`	`Quaternion`	`Thickness`
`Char`	`Int64`	`Rect`	`Vector`
`Color`	`Matrix`	`Rotation3D`	`Vector3D`
`Decimal`	`Object`	`Single`	
`Double`	`Point`	`Size`	

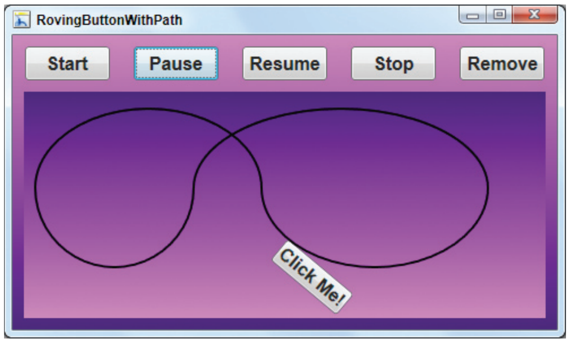
Most of these data types have two kinds of animation classes. The first has the suffix *Animation* and performs a simple linear interpolation of the property's values so that the property's value is set proportionally between the start and finish values based on the elapsed time. When the `Duration` is 1/4 over, the value will be one-fourth of the way between the start and finish values.

The second type of animation class has the suffix *AnimationUsingKeyFrames*. This type of animation changes a property from one value to another while visiting specific key values in between. You can use several different kinds of objects to specify key frame values, and the type of key frame class determines how the animation moves to the frame's value.

The most common types of key frame objects move to their key values linearly, using a spline, or discretely. A few other key frame animations can make their values follow a path.

For example, you can use a path animation to move a control along the border of an ellipse or a series of curves.

The RovingButtonWithPath example program shown in here moves a `Button` along the path shown in black. As it moves, the `Button` rotates to match the curve's angle at that point.



The RovingButtonWithPath program uses the following code to animate its `Button`:

Available for download on Wrox.com

```
<PathGeometry x:Key="pathMove"
 Figures="M 10,85
  A 100,70 0 1 1 210,85
  A 100,70 0 1 0 410,85
  A 130,70 0 1 0 150,85
  A 70,70 0 1 1 10,85"/>
<Storyboard x:Key="sbMoveButton" RepeatBehavior="Forever">
  <DoubleAnimationUsingPath Duration="0:0:4"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)"
   Source="X"
   PathGeometry="{StaticResource pathMove}"/>
  <DoubleAnimationUsingPath Duration="0:0:4"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)"
   Source="Y"
   PathGeometry="{StaticResource pathMove}"/>
  <DoubleAnimationUsingPath Duration="0:0:4"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="RenderTransform.Angle"
   Source="Angle"
   PathGeometry="{StaticResource pathMove}"/>
</Storyboard>
```

WPF Programmer's Reference 9780470477229 RovingButtonWithPath

The code first creates a `PathGeometry` object to define the path. It then uses that object in the `sbMoveButton Storyboard`.

The `Storyboard`'s first `DoubleAnimationUsingPath` object changes the `Button`'s `Canvas.Left` property. The `Source` attribute indicates that the property should be set to the path's `X` coordinate as the animation progresses over the path.

Similarly, the second `DoubleAnimationUsingPath` object makes the `Button`'s `Canvas.Top` property match the path's `Y` coordinate as the animation progresses.

The `Button`'s definition includes a `RenderTransform` object that initially rotates the `Button` by –90 degrees. The final animation object makes the `Button`'s `Transform.Angle` property match the path's `Angle` during the animation.

Media and Timelines

The `BeginStoryboard`, `PauseStoryboard`, and other `Storyboard` control classes are really just *action wrappers* — objects that perform actions rather than representing "physical" objects such as buttons, labels, and grids. Another useful action wrapper is `SoundPlayerAction`, which plays an audio file.

The BouncingBall example program shown in the next figure uses the following `Storyboard` to make its red ball bounce up and down while playing a sound effect each time the ball hits the "ground."

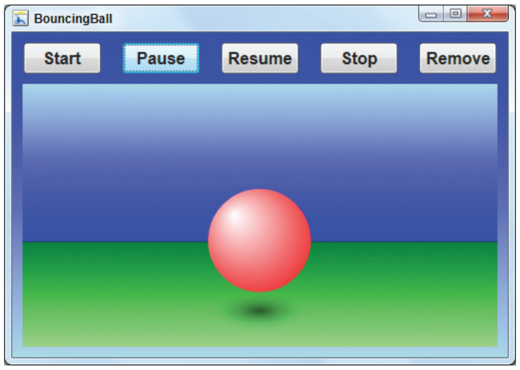
Available for download on Wrox.com

```
<!-- One Storyboard to rule them all. -->
<Storyboard x:Key="sbBounce" RepeatBehavior="Forever">
  <!-- Play the sound after 1 second. -->
  <ParallelTimeline BeginTime="0:0:0">
    <MediaTimeline BeginTime="0:0:1" Source="boing.wav"
```

```
         Storyboard.TargetName="medBoing"/>
      </ParallelTimeline>

      <!--  Move the ball and its shadow.  -->
      <ParallelTimeline BeginTime="0:0:0" AutoReverse="True">
        <DoubleAnimationUsingKeyFrames
         Storyboard.TargetName="ellBall"
         Storyboard.TargetProperty="(Canvas.Top)">
          <SplineDoubleKeyFrame KeyTime="0:0:1"
           KeySpline="0.5,0 1,1"
           Value="120"/>
        </DoubleAnimationUsingKeyFrames>
        <DoubleAnimationUsingKeyFrames
         Storyboard.TargetName="ellShadow"
         Storyboard.TargetProperty="Opacity">
          <SplineDoubleKeyFrame KeyTime="0:0:1"
           KeySpline="0.5,0 1,1"
           Value="1"/>
        </DoubleAnimationUsingKeyFrames>
      </ParallelTimeline>
    </Storyboard>
```

WPF Programmer's Reference 9780470477229 BouncingBall

The Storyboard contains two ParallelTimeline objects that run independently. The first waits 1 second until the ball is touching the "ground" and then uses a MediaTimeline object to play the sound effect.



The second ParallelTimeline contains two DoubleAnimationUsingKeyframes objects, one to control the ball's Canvas.Top property and one to control the shadow's Opacity. As the ball's animation moves the ball downward, the shadow's animation makes the shadow more opaque. When the animations end, the ball is touching the "ground," and the shadow is fully opaque.

TEMPLATES

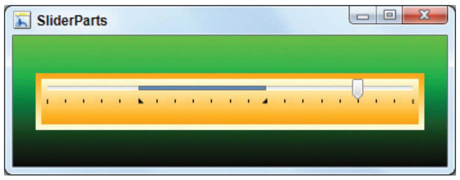
Properties and styles determine a control's appearance and behavior.

In contrast, *templates* determine a control's structure. They determine what components make up the control and how those components interact to provide the control's features.

Template Overview

If you look closely at the next figure, you can see that a `Slider` control has a bunch of parts including:

- A border
- Tick marks
- A background
- Clickable areas on the background (basically anywhere between the top of the control and its tick marks vertically) that change the current value
- A Thumb indicating the current value that you can drag back and forth
- Selection indicators (the little black triangles) that indicate a selected range



These features are provided by the pieces that make up the `Slider`.

A *template* determines what the pieces are that make up a control. It determines the control's components together with their styles, triggers, and everything else that is needed by the control.

ContentPresenter

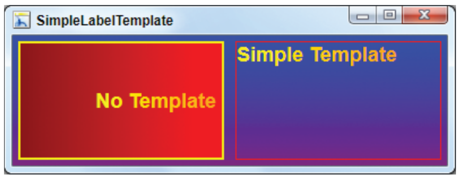
The `ContentProvider` is an object that WPF provides to display whatever it is that the control should display. You can place the `ContentProvider` in whatever control hierarchy you build for the template, and it will display the content.

For example, the following code shows an extremely simple `Label` template:

Available for download on Wrox.com

```
<Window.Resources>
    <ControlTemplate x:Key="temSimpleLabel" TargetType="Label">
        <Border BorderBrush="Red" BorderThickness="1">
            <ContentPresenter/>
        </Border>
    </ControlTemplate>
</Window.Resources>
```

The SimpleLabelTemplate example program shown in the next figure displays two `Labels`. The one on the left uses no template, while the one on the right uses the `temSimpleLabel` template.



Template Binding

If you compare the two `Labels` in the previous figure, you'll see that even this simple example has some potential problems. Because the template's `Border` control includes explicit values for its `BorderBrush` and `BorderThickness` properties, it overrides any values set in the code that creates the `Label`. The `Border` control also doesn't specify a `Background`, so it uses its default transparent background.

This means the templated control doesn't display the correct background or border. It also doesn't honor the requested `HorizontalContentAlignment` and `VerticalContentAlignment` values.

Fortunately, a template can learn about some of the properties set on the client control by using a *template binding*. For example, the following code fragment sets the `Background` property for a piece of the template to the value set for the control's `Background` property:

```
Background="{TemplateBinding Background}"
```

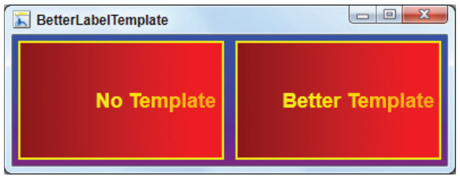
The following code shows a better version of the `Label` template that honors several of the control's background and foreground properties:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temBetterLabel" TargetType="Label">
    <Border
     Background="{TemplateBinding Background}"
     BorderBrush="{TemplateBinding BorderBrush}"
     BorderThickness="{TemplateBinding BorderThickness}">
        <ContentPresenter Margin="4"
         HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
         VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
    </Border>
</ControlTemplate>
```

WPF Programmer's Reference 9780470477229 *BetterLabelTemplate*

The BetterLabelTemplate example program shown here uses this template to display a `Label` that looks much more like one that has no template.

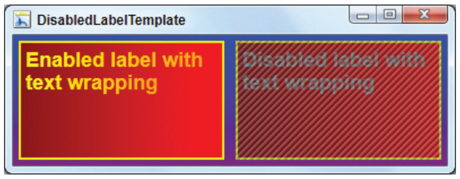


Template Events

To make a template control respond to events, you can add property and event triggers to the template much as you previously added them to styles.

In addition to events caused by user actions such as moving or pressing the mouse, controls must respond to changes in state. For example, although a `Label` mostly just sits around doing nothing, it should also change its appearance when it is disabled.

The DisabledLabelTemplate example program shown here uses a template that gives a disabled `Label` a distinctive appearance.


DisabledLabelTemplate
Enabled label with text wrapping
Disabled label with text wrapping

The following code shows the template that gives the disabled `Label` its appearance:

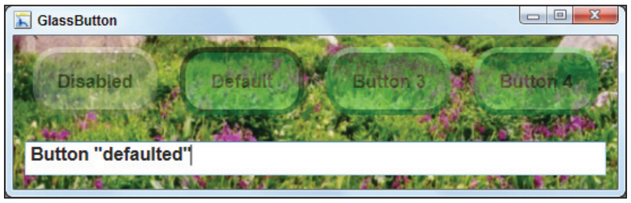
Available for download on Wrox.com

```
<ControlTemplate x:Key="temWrappedLabel" TargetType="Label">
    <Grid>
        <Border Name="brdMain"
         Background="{TemplateBinding Background}"
         BorderBrush="{TemplateBinding BorderBrush}"
         BorderThickness="{TemplateBinding BorderThickness}">
            <TextBlock Name="txtbContent"
            Margin="4"
            TextWrapping="Wrap"
            Text="{TemplateBinding ContentPresenter.Content}"/>
        </Border>
        <Canvas Name="canDisabled" Opacity="0">
            <Canvas.Background>
                <LinearGradientBrush StartPoint="0,0" EndPoint="3,3"
                MappingMode="Absolute"
                SpreadMethod="Repeat">
                    <GradientStop Color="LightGray" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
                </LinearGradientBrush>
            </Canvas.Background>
        </Canvas>
    </Grid>
    <ControlTemplate.Triggers>
        <Trigger Property="IsEnabled" Value="False">
            <Setter TargetName="canDisabled"
            Property="Opacity" Value="0.5"/>
            <Setter TargetName="txtbContent"
            Property="Foreground" Value="Gray"/>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

WPF Programmer's Reference 9780470477229 DisabledLabelTemplate

Glass Button

The GlassButton example program shown here uses a template to give its buttons a glassy appearance.

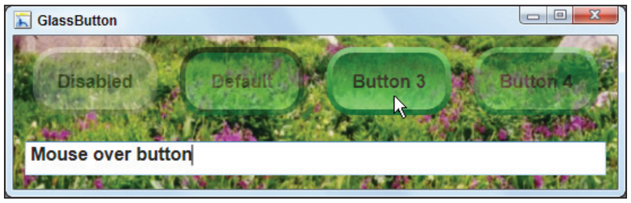


Button "defaulted"

The disabled button on the left looks washed-out and doesn't respond to the user.

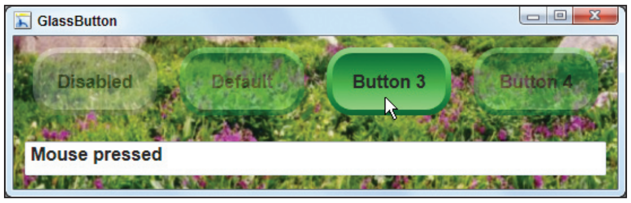
The second button labeled *Default* has a different border from that of the other buttons. If no other button has the focus when the user presses the [Enter] key, that `Button` fires its `Click` event. In the previous figure, the `TextBox` has the focus, so pressing the [Enter] key will fire the default `Button`.

The next figure shows the program when the mouse is over Button 3. The button under the mouse becomes less transparent. Notice that the focus is still in the `TextBox` (you can see the caret in this figure), so the default button still shows its distinctive border.



Mouse over button

The next figure shows the program when the user presses the mouse down on Button 3.



Mouse pressed

At this point, the pressed button is opaque. Pressing the button also moves focus to that `button`. Because focus is now on Button 3, the default button is no longer defaulted. In fact, no button is defaulted right now. Button 3 has the focus so it will fire if the user presses [Enter] but it is not defaulted so it won't display the default border even after the user releases the mouse.

If you drag the mouse off the button while it is still pressed, the button returns to its focused "mouse over" appearance. If you then release the mouse, no mouse click occurs.

You can download the example program from the WPF Programmer's Reference (9780470477229) web site to see the details.

Researching Control Templates

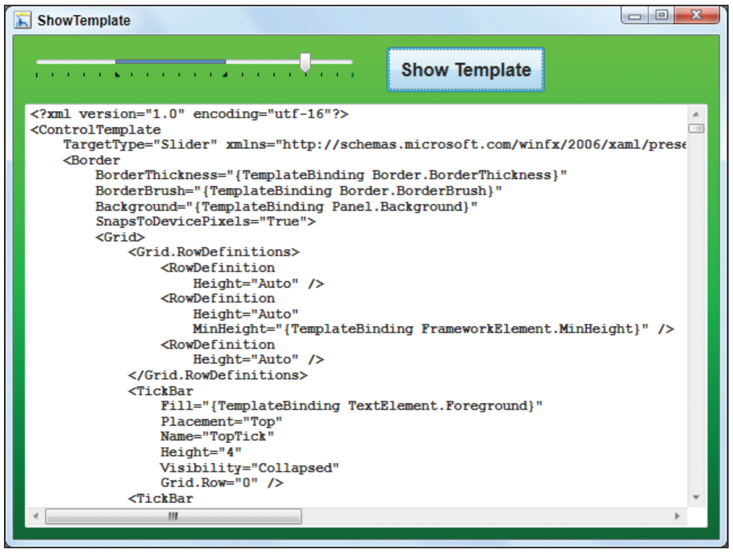
To effectively build templates, you need to learn what behaviors the control provides for you and what behaviors you need to provide for it. You also need to determine what events the control provides so that you know when you have a chance to make the control take action.

So, how do you learn what properties and template bindings are available?

One good source of information is Microsoft's "Control Styles and Templates" web page at `msdn.microsoft.com/cc278075(VS.95).aspx`. That page provides links to other pages that describe the features available to different control templates.

These web pages also show the default templates used by the controls. The `Button` control's default template is 84 lines long and fairly complicated. Some are much longer and much more complex.

In addition to using Microsoft's web pages, you can make a control tell you about its template. The ShowTemplate example program displays the default template for a control. In the following figure, that control is the `Slider` in the upper-left corner. To see the template used by a different kind of control, replace the `Slider` with a different control, name it `Target`, and run the program.



```
<?xml version="1.0" encoding="utf-16"?>
<ControlTemplate
    TargetType="Slider" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/prese
    <Border
        BorderThickness="{TemplateBinding Border.BorderThickness}"
        BorderBrush="{TemplateBinding Border.BorderBrush}"
        Background="{TemplateBinding Panel.Background}"
        SnapsToDevicePixels="True">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition
                    Height="Auto" />
                <RowDefinition
                    Height="Auto"
                    MinHeight="{TemplateBinding FrameworkElement.MinHeight}" />
                <RowDefinition
                    Height="Auto" />
            </Grid.RowDefinitions>
            <TickBar
                Fill="{TemplateBinding TextElement.Foreground}"
                Placement="Top"
                Name="TopTick"
                Height="4"
                Visibility="Collapsed"
                Grid.Row="0" />
            <TickBar
```

The ShowTemplate program uses the following code to display the `Target` control's template:

Available for download on Wrox.com

```
private void btnShowTemplate_Click(object sender, RoutedEventArgs e)
{
    XmlWriterSettings writer_settings = new XmlWriterSettings();
    writer_settings.Indent = true;
    writer_settings.IndentChars = "    ";
    writer_settings.NewLineOnAttributes = true;

    StringBuilder sb = new StringBuilder();
    XmlWriter xml_writer = XmlWriter.Create(sb, writer_settings);

    XamlWriter.Save(Target.Template, xml_writer);

    txtResult.Text = sb.ToString();
}
```

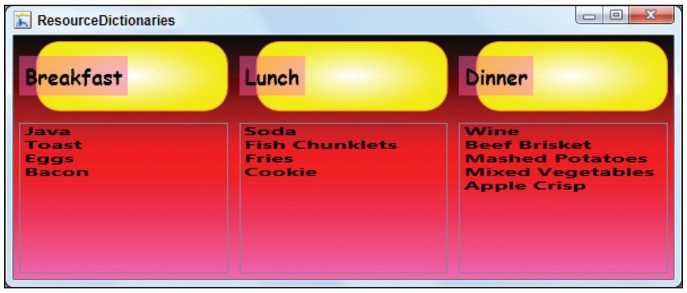
WPF Programmer's Reference 9780470477229 ShowTemplate

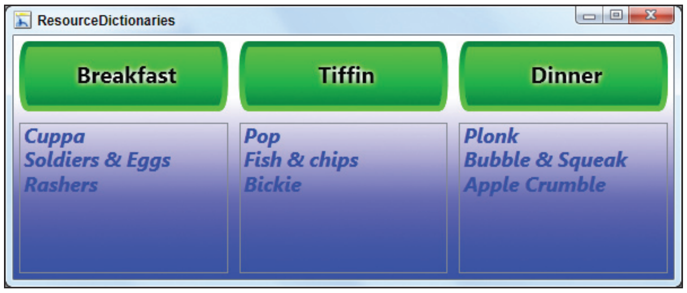
SKINS

Themes let a program automatically change to match the rest of the system's appearance.

Skins are much more interesting. A *skin* is a packaged set of appearances and behaviors that can give an application (or part of an application) a distinctive appearance while still allowing it to provide its key features.

For example, the next two figures show the ResourceDictionaries example program (see the "Merged Resource Dictionaries" section) displaying two very different appearances. It's the same program in both figures and it contains the same controls — just rearranged slightly and with different colors, fonts, and so forth.





Resource Skins

The program ResourceDictionaries shown in the previous two figures uses two different sets of resources to change its appearance at design time.

The following code shows the `Window`'s resource dictionary. The inner `ResourceDictionary` elements load two different resource dictionaries. Because dictionaries loaded later override those loaded earlier, you can change the application's appearance by changing the order of these two elements. (As shown here, the RedRed.xaml dictionary is loaded second, so the program uses its red interface.)

Available for download on Wrox.com

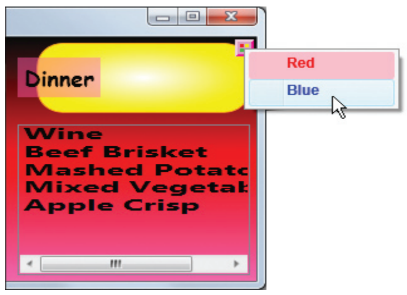
```
<Window.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="ResBlue.xaml"/>
            <ResourceDictionary Source="ResRed.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Window.Resources>
```

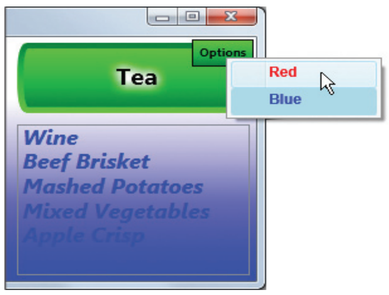
WPF Programmer's Reference 9780470477229 ResourceDictionary

While the program ResourceDictionaries can display two different appearances, you need to modify the program at design time to pick the skin you want.

The Skins example program is very similar to the program ResourceDictionaries except that it can change skins at run time. To make that possible, most of its resources are dynamic rather than static. The program also contains two new user interface elements: an `Image` and a `Label`.

When it displays its red interface, this program adds a small `Image` in its upper-right corner. This `Image` has a context menu that displays the choices Red and Blue (shown on the left in the next figure), which let you pick between the red and blue skins.





The program's blue interface displays a label in its upper-right corner (on the right in the previous figure) that displays the same context menu.

The following code shows how the program displays its Options textbox on the blue interface:

Available for download on Wrox.com

```
<Label MouseDown="Options_MouseDown"
 Grid.Row="0" Grid.Column="2" Margin="2"
 Content="Options" FontSize="10"
 HorizontalAlignment="Right" VerticalAlignment="Top"
 Foreground="Black" BorderBrush="Black"
 BorderThickness="1"
 Visibility="{DynamicResource visBlue}"
>
    <Label.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Color="Lime" Offset="0"/>
            <GradientStop Color="Green" Offset="1"/>
        </LinearGradientBrush>
    </Label.Background>
    <Label.ContextMenu>
        <ContextMenu Name="ctxOptions">
            <MenuItem Header="Red" Background="Pink"
             Foreground="Red"
            Click="ctxSkin_Click" Tag="ResRed.xaml"/>
            <MenuItem Header="Blue" Background="LightBlue"
             Foreground="Blue"
            Click="ctxSkin_Click" Tag="ResBlue.xaml"/>
        </ContextMenu>
    </Label.ContextMenu>
</Label>
```

WPF Programmer's Reference 9780470477229 Skins

This code contains four real points of interest:

1. First, the `Label`'s `MouseDown` event triggers the `Options_MouseDown` event handler. This routine, which is shown in the following code, displays the context menu by setting the `ContextMenu`'s `IsOpen` property to `True`:

Available for download on Wrox.com

```
// Display the options context menu.
private void Options_MouseDown(object sender, RoutedEventArgs e)
{
    ctxOptions.IsOpen = true;
}
```

WPF Programmer's Reference 9780470477229 Skins

2. Second, the `Label`'s `Visibility` property is set to the value of the `visBlue` static resource. This resource has the value `True` in the Blue resource dictionary and `False` in the Red resource dictionary. This means that the `Label` is visible only in the blue interface. The red interface's skin-changing `Image` uses a similar `visRed` resource that is only `True` in the red interface.
3. Third, the context menu's items have a `Tag` property that names the XAML resource file that they load. For example, the Blue menu item has its `Tag` set to `ResBlue.xaml`. The program uses the `Tag` property to figure out which file to load when the user picks a menu item.
4. Finally, both of the context menu's items fire the `ctxSkin_Click` event handler shown in the following code to load the appropriate skin:

Available for download on Wrox.com

```
// Use the selected skin.
private void ctxSkin_Click(object sender, RoutedEventArgs e)
{
    // Get the context menu item that was clicked.
    MenuItem menu_item = (MenuItem)sender;

    // Create a new resource dictionary, using the
    // menu item's Tag property as the dictionary URI.
    ResourceDictionary dict = new ResourceDictionary();
    dict.Source = new Uri((String)menu_item.Tag, UriKind.Relative);

    // Remove all but the first dictionary.
    while (App.Current.Resources.MergedDictionaries.Count > 1)
    {
        App.Current.Resources.MergedDictionaries.RemoveAt(1);
    }

    // Install the new dictionary.
    App.Current.Resources.MergedDictionaries.Add(dict);
}
```

WPF Programmer's Reference 9780470477229 Skins

This code gets the menu item that triggered the event and looks at the item's `Tag` property to see which resource file to load. It creates a `ResourceDictionary` object loaded from the file, removes old resource dictionaries from the application's MergedDictionaries collection, and adds the new dictionary.

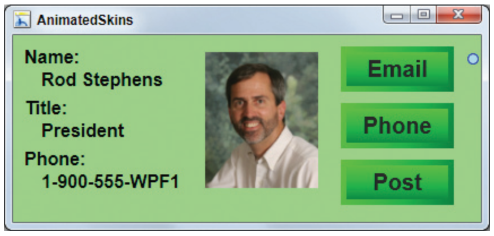
Animated Skins

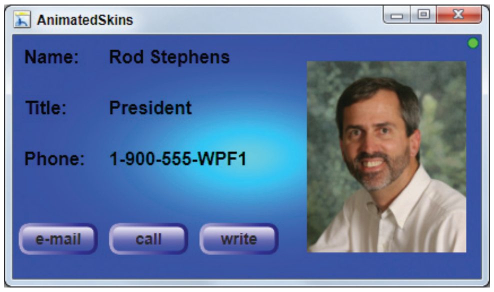
The skins described in the previous section use separate resource dictionaries to provide different appearances. The program's XAML file sets its control properties to resource values so that when you change the resource values, the interface changes accordingly.

Another way to change property values is to use property animation. This section explains briefly how to use animation to provide skinning.

XAML files allow you to define triggers that launch storyboards that represent property animations. For example, when the user presses the mouse down over a rectangle, the XAML code can run a storyboard that varies the `Rectangle`'s `Width` property smoothly from 100 to 200 over a 1-second period.

The AnimatedSkins example program uses this technique to provide skinning. The next two figures show the program displaying its green skin and its blue skin.





When you click the appropriate control, a trigger launches a storyboard that:

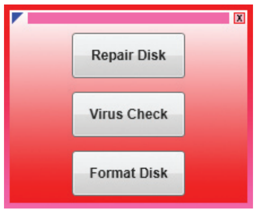
- Resizes the main window and changes its `Background` brush.
- Hides and displays the small blue or green ellipses in the upper-right corner that you click to switch skins.
- Moves `Labels`.
- Resizes, moves, and changes the corner radii of the rectangles that act as buttons.
- Changes the `Fill` and `Stroke` brushes for the rectangles pretending to be buttons.
- Changes the text displayed in the `Labels`.
- Moves and resizes the `Image`.
- Changes the background and foreground colors.

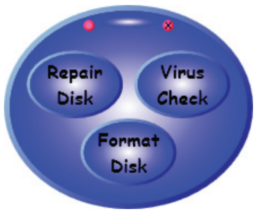
In addition to displaying very different appearances, animated skins let the user watch as one interface morphs into another. The effect is extremely cool.

Dynamically Loaded Skins

One of the drawbacks of the previous two skinning techniques is that they only modify existing objects. They can display an `Ellipse`, `Button`, or `Label` with different properties, but they are still the same `Ellipse`, `Button`, or `Label`. For example, you cannot provide one skin that launches tasks with `Buttons`, another that uses `Menus`, and a third that uses `Labels`.

The SkinInterfaces example program displays new skins at run time by loading XAML files and wiring up their event handlers. The next two figures show the program displaying its two skins.





These skins not only provide radically different appearances, but they also use different types of controls that generate different kinds of events.

When the program loads a XAML file, it looks through the new controls and attaches event handlers to those that need them.

To provide some separation between the XAML files and the code-behind, the program uses a separate group of routines to do the real work. Event handlers catch the control events and call the work routines to do all the interesting stuff.

The function `LoadSkin` uses the following code to load a XAML skin file. To save space, the code only shows a few of the statements that connect controls to their event handlers.

Available for download on Wrox.com

```
// Load the skin file and wire up event handlers.
private void LoadSkin(string skin_file)
{
    // Load the controls.
    FrameworkElement element =
        (FrameworkElement)Application.LoadComponent(
            new Uri(skin_file, UriKind.Relative));
    this.Content = element;

    // Wire up the event handlers.
```

```
        Button btn;
        Polygon pgn;
        Rectangle rect;
        Grid grd;
        Ellipse ell;

        switch (element.Tag.ToString())
        {
            case "Red":
                btn = (Button)element.FindName("btnRepairDisk");
                btn.Click += new RoutedEventHandler(btnRepairDisk_Click);

                    Code for other controls omitted
                break;

            case "Blue":
                    Lots of code omitted

                // Uses the same event handler as rectMove.
                ell = (Ellipse)element.FindName("ellMove");
                ell.MouseDown +=
                    new System.Windows.Input.MouseButtonEventHandler(
                        rectMove_MouseDown);

                grd = (Grid)element.FindName("grdExit");
                grd.MouseDown +=
                    new System.Windows.Input.MouseButtonEventHandler(
                        grdExit_MouseDown);
                break;
        }
    }
```

WPF Programmer's Reference 9780470477229 SkinInterfaces

The code starts by using the WPF `LoadComponent` method to load the desired XAML skin file. It sets the `Window`'s main content element to the root loaded from the file so the new controls are displayed.

Next, the code checks the newly loaded root element's `Tag` property to see whether it is now displaying the red or the blue skin. Depending on which skin is loaded, the code looks for specific controls in the skin and connects their event handlers.

PRINTING VISUAL OBJECTS

One of the central classes for printing in WPF is `PrintDialog`. This class provides methods for letting the user pick a printer, interacting with print queues, and sending output to the printer.

Normally there are three steps to printing:

1. Create a new `PrintDialog` object and use its `ShowDialog` method to display it to the user.
2. Check `ShowDialog`'s return result to see if the user clicked OK.
3. Use the `PrintDialog`'s methods to generate a printout.

The SimplePrintWindow example program uses the following code to perform these three steps:

Available for download on Wrox.com

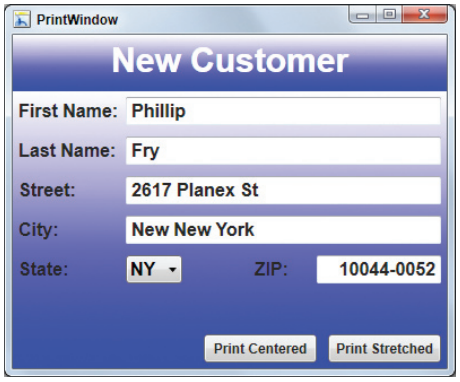
```
// Display the print dialog.
PrintDialog pd = new PrintDialog();

// See if the user clicked OK.
if (pd.ShowDialog() == true)
{
    // Print.
    pd.PrintVisual(this, "New Customer");
}
```

WPF Programmer's Reference 9780470477229 SimplePrintWindow

This is remarkably simple, but it has an unfortunate drawback: WPF tries to draw the visual in the upper-left corner of the paper. Since printers generally cannot print all the way to the edges of the paper, the result is chopped off.

Rather than passing the program's window into the `PrintVisual` method, you can make a hierarchy of controls that contains an image of the window. That hierarchy can use `Grids`, `Viewboxes`, `Images`, `Rectangles`, and any other control that you want to produce the results. It can even include extra controls to produce such items as a page header and footer.

The PrintWindow example program shown in the next figure uses this approach to print its window centered at either normal or enlarged scale. Click on the "Print Centered" button to print the window at its normal scale. Click on the "Print Stretched" button to print the window as large as possible on the page.



First Name: Phillip
Last Name: Fry
Street: 2617 Planex St
City: New New York
State: NY ZIP: 10044-0052

The PrintWindow program requires references to the ReachFramework and System.Printing libraries. In Visual Studio, open the Project menu and select "Add Reference." Select these two libraries and click OK.

To make working with the libraries easier, the program includes the following `using` statements:

```
using System.Printing;
using System.Windows.Media.Effects;
```

When you click its buttons, the PrintWindow program uses the following code to start the printing process. Both buttons display the `PrintDialog` and, if the user selects a printer and clicks OK, call `PrintWindowCentered` to do all of the interesting work.

Available for download on Wrox.com

```
// Print the window centered.
private void btnPrintCentered_Click(object sender, RoutedEventArgs e)
{
    PrintDialog pd = new PrintDialog();
    if (pd.ShowDialog() == true)
    {
        PrintWindowCentered(pd, this, "New Customer", null);
    }
}

// Print the window stretched to fit.
private void btnPrintStretched_Click(object sender, RoutedEventArgs e)
{
    PrintDialog pd = new PrintDialog();
    if (pd.ShowDialog() == true)
    {
        PrintWindowCentered(pd, this, "New Customer", new Thickness(50));
    }
}
```

WPF Programmer's Reference 9780470477229 PrintWindow

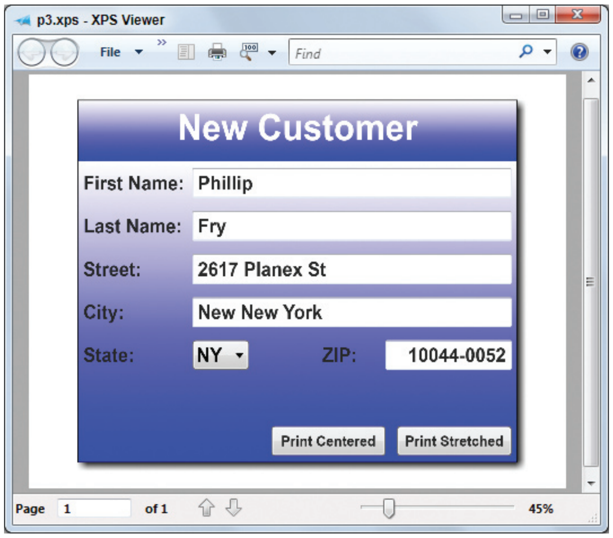
The following code shows the `PrintWindowCentered` function:

Available for download on Wrox.com

```
// Print a Window centered on the printer.
private void PrintWindowCentered(PrintDialog pd, Window win,
 String title, Thickness? margin)
{
    // Make a Grid to hold the contents.
    Grid drawing_area = new Grid();
    drawing_area.Width = pd.PrintableAreaWidth;
    drawing_area.Height = pd.PrintableAreaHeight;

    // Make a Viewbox to stretch the result if necessary.
    Viewbox view_box = new Viewbox();
    drawing_area.Children.Add(view_box);
    view_box.HorizontalAlignment = HorizontalAlignment.Center;
    view_box.VerticalAlignment = VerticalAlignment.Center;

    if (margin == null)
    {
        // Center without resizing.
        view_box.Stretch = Stretch.None;
    }
    else
    {
        // Resize to fit the margin.
        view_box.Margin = margin.Value;
        view_box.Stretch = Stretch.Uniform;
        }

        // Make a VisualBrush holding an image of the Window's contents.
        VisualBrush vis_br = new VisualBrush(win);

        // Make a rectangle the size of the Window.
        Rectangle win_rect = new Rectangle();
        view_box.Child = win_rect;
        win_rect.Width = win.Width;
        win_rect.Height = win.Height;
        win_rect.Fill = vis_br;
        win_rect.Stroke = Brushes.Black;
        win_rect.BitmapEffect = new DropShadowBitmapEffect();

        // Arrange to produce output.
        Rect rect = new Rect(0, 0,
            pd.PrintableAreaWidth, pd.PrintableAreaHeight);
        drawing_area.Arrange(rect);

        // Print it.
        pd.PrintVisual(drawing_area, title);
    }
```

WPF Programmer's Reference 9780470477229 PrintWindow

The next figure shows the result when the program's window is stretched and printed using the Microsoft XPS Document Writer with preferences set to landscape orientation and displayed in XPS Viewer.



First Name: Phillip
Last Name: Fry
Street: 2617 Planex St
City: New New York
State: NY ZIP: 10044-0052

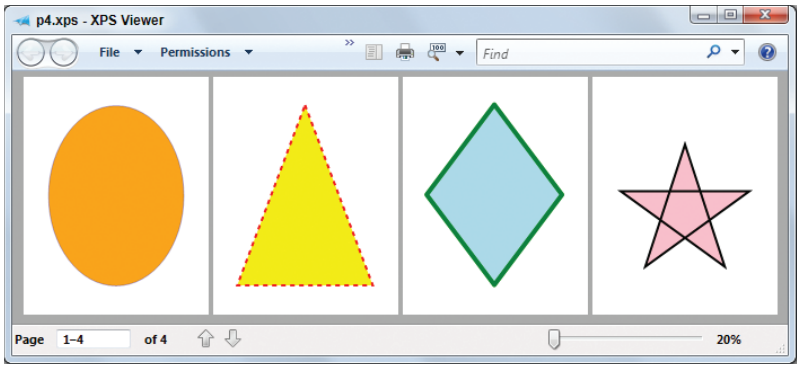
PRINTING CODE-GENERATED OUTPUT

The PrintDialog object's PrintVisual method is easy to use and, with a little extra work, can produce nicely scaled and centered results. It still assumes that you are only printing one page at a time, however. If you want to print a longer document, you'll need to call PrintVisual once for each page, and it will produce one print job for each page. This isn't an ideal solution. Fortunately there's a better way to produce multi-page printouts.

The PrintDialog object's PrintDocument method takes a DocumentPaginator object as a parameter. That object generates the pages of a printout, and the PrintDocument places them in a single print job.

DocumentPaginator is an abstract class that defines methods for you to implement in a subclass. Those methods give the PrintDocument method information such as the printout's total number of pages and the objects to print.

The PrintShapes example program uses a ShapesPaginator class that inherits from DocumentPaginator to print four pages of shapes. The next figure shows the output saved by the Microsoft XPS Document Writer printer and displayed in XPS Viewer.



The following code shows how the program responds when you click on its "Print Shapes" button. Like the previous code that uses PrintVisual, it displays a PrintDialog and sees whether the user selected a printer and clicked OK. It then calls the PrintDocument method, passing it a paginator object and a print job title.

Available for download on Wrox.com

```
// Print the shapes.
private void btnPrintShapes_Click(object sender, RoutedEventArgs e)
{
    PrintDialog pd = new PrintDialog();
    if (pd.ShowDialog() == true)
    {
        pd.PrintDocument(
            new ShapesPaginator(
                new Size(pd.PrintableAreaWidth, pd.PrintableAreaHeight)),
            "Shapes");
    }
}
```

WPF Programmer's Reference 9780470477229 PrintShapes

The following code fragment shows the key pieces of the `ShapesPaginator` class:

Available for download on Wrox.com

```
class ShapesPaginator : DocumentPaginator
{
    private Size m_PageSize;

    // Save the page size.
    public ShapesPaginator(Size page_size)
    {
        m_PageSize = page_size;
    }

    // Return the needed page.
    public override DocumentPage GetPage(int pageNumber)
    {
        const double WID = 600;
        const double HGT = 800;

        Grid drawing_grid = new Grid();
        drawing_grid.Width = m_PageSize.Width;
        drawing_grid.Height = m_PageSize.Height;

        switch (pageNumber)
        {
            case 0: // Ellipse
                Ellipse ell = new Ellipse();
                ell.Fill = Brushes.Orange;
                ell.Stroke = Brushes.Blue;
                ell.StrokeThickness = 1;
                ell.HorizontalAlignment = HorizontalAlignment.Center;
                ell.VerticalAlignment = VerticalAlignment.Center;
                ell.Width = WID;
                ell.Height = HGT;
                drawing_grid.Children.Add(ell);
                break;

            ... Code for other pages omitted ...
        }

        // Arrange to make the controls draw themselves.
        Rect rect = new Rect(new Point(0, 0), m_PageSize);
        drawing_grid.Arrange(rect);

        // Return a DocumentPage wrapping the grid.
        return new DocumentPage(drawing_grid);
    }

    // If pagination is in progress and PageCount is not final, return False.
    // If pagination is complete and PageCount is final, return True.
    // In this example, there is no pagination to do.
    public override bool IsPageCountValid
    {
        get { return true; }
        }

        // The number of pages paginated so far.
        // This example has exactly 4 pages.
        public override int PageCount
        {
            get { return 4; }
        }

        // The suggested page size.
        public override Size PageSize
        {
            get { return m_PageSize; }
            set { m_PageSize = value; }
        }

        // The element currently being paginated.
        public override IDocumentPaginatorSource Source
        {
            get { return null; }
        }
    }
```

WPF Programmer's Reference 9780470477229 PrintShapes

DATA BINDING

WPF data binding lets you bind a target to a data source so the target automatically displays the value in the data source. For example, this lets you:

- Make a `ListBox` display an array of values defined in XAML code.
- Make a `ListBox` display a list of objects created in code-behind.
- Make a `TreeView` build a hierarchical display of objects created in code-behind.
- Make `TextBoxes`, `Labels`, and other controls display additional detail about the currently selected item in a `ListBox`, `ComboBox`, or `TreeView`.

Additional WPF data-binding features let you sort, filter, and group data; let the user modify a control to update a data source; and validate changes to data.

Binding Basics

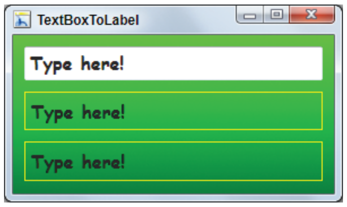
Data bindings have these four basic pieces:

- **Target** — The object that will use the result of the binding
- **Target Property** — The target object's property that will use the result
- **Source** — The object that provides a value for the target object to use
- **Path** — A path that locates the value within the source object

As a trivial example, suppose you want to bind a `Label` control's `Content` property so that the `Label` displays whatever you type in a `TextBox`. If the `Label` control is named `lblResult` and the `TextBox` is named `txtTypeHere`, then you would need to create a binding where:

- The target is `lblResult` (the control where you want the binding's result to go).
- The target property is `Content` (you want the `Label` to display the result in its `Content` property).
- The source is `txtTypeHere` (the object providing the value).
- The path is `Text` (the path to the data in the source object, in this case, the `TextBox`'s `Text` property).

The TextBoxToLabel example program shown here demonstrates this kind of simple binding. When you type in the textbox, the two labels below it echo whatever you type.



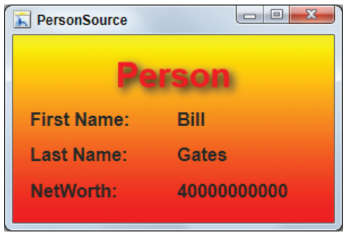
The following XAML code shows how the program works:

Available for download on Wrox.com

```
<StackPanel Margin="5">
    <TextBox Name="txtTypeHere" Margin="5"
Height="30"
     VerticalAlignment="Top" Text="Type here!"/>

    <Label Margin="5" BorderBrush="Yellow" BorderThickness="1">
        <Binding ElementName="txtTypeHere" Path="Text"/>
    </Label>

    <Label Margin="5" BorderBrush="Yellow" BorderThickness="1"
     Content="{Binding ElementName=txtTypeHere, Path=Text}"/>
</StackPanel>
```

WPF Programmer's Reference 9780470477229 TextBoxToLabel

The PersonSource example program shown in the next figure uses a binding with its `Source` property set to display information about a `Person` object.



First Name:	Bill
Last Name:	Gates
NetWorth:	40000000000

The PersonSource example program defines a `Person` class with the properties `FirstName`, `LastName`, and `NetWorth`. The program's XAML code includes the following namespace definition so it can use the class:

```
xmlns:local="clr-namespace:PersonSource"
```

The following code shows how the program's XAML code defines a static resource named `a_person` that is a `Person` object:

Available for download on Wrox.com

```
<Window.Resources>
    <local:Person x:Key="a_person" FirstName="Bill"
     LastName="Gates" NetWorth="40000000000"/>
</Window.Resources>
```

WPF Programmer's Reference 9780470477229 PersonSource

The following code shows how the program uses bindings to display the `Person` object's `FirstName`, `LastName`, and `NetWorth` values. It uses binding `Source` properties to identify the `Person` object that provides the values.

Available for download on Wrox.com

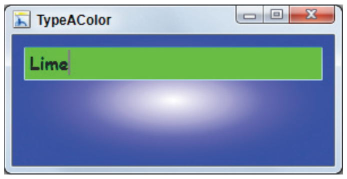
```
<Label Grid.Row="1" Grid.Column="0" Content="First Name:"/>
<Label Grid.Row="1" Grid.Column="1"
 Content="{Binding Source={StaticResource a_person}, Path=FirstName}"/>

<Label Grid.Row="2" Grid.Column="0" Content="Last Name:"/>
<Label Grid.Row="2" Grid.Column="1"
 Content="{Binding Source={StaticResource a_person}, Path=LastName}"/>

<Label Grid.Row="3" Grid.Column="0" Content="NetWorth:"/>
<Label Grid.Row="3" Grid.Column="1"
 Content="{Binding Source={StaticResource a_person}, Path=NetWorth}"/>
```

WPF Programmer's Reference 9780470477229 PersonSource

RelativeSource

The binding's `RelativeSource` property lets you specify a source object by its relationship to the target control. One situation in which this is useful is when you want to bind two properties on the same control.

The TypeAColor example program shown in the next figure binds the `TextBox`'s `Background` property to its `Text` property, so when you type the name of a color, the control uses that color as its background.



The following code shows how the TypeAColor program binds the `TextBox`'s `Background` property to its `Text` property:

Available for download on Wrox.com

```
<TextBox Margin="10" Height="30" VerticalAlignment="Top"
 Background="{Binding RelativeSource={RelativeSource Self}, Path=Text}"/>
```

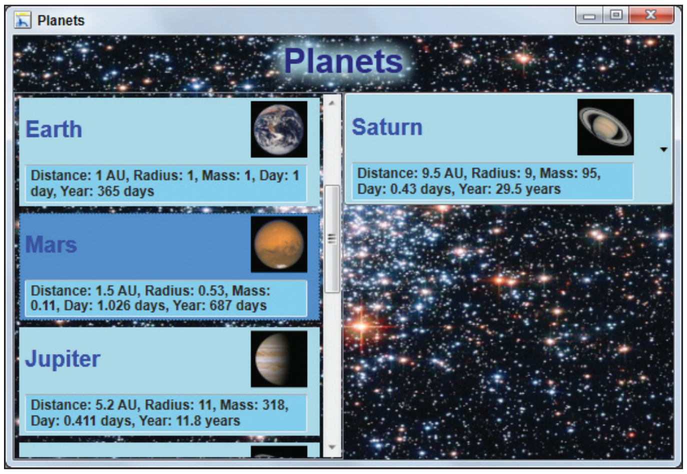
WPF Programmer's Reference 9780470477229 TypeAColor

ListBox and ComboBox Templates

The `ListBox` and `ComboBox` controls have an `ItemTemplate` property that determines how each item is displayed. These templates are somewhat similar to the templates described in the "Templates" section.

The template can hold whatever controls you like to represent the control's items. Usually many of these controls are bound to the properties of whatever item the control is currently displaying.

The Planets example program shown in the next figure displays an array of `Planet` objects in a `ListBox` on the left and in a `ComboBox` on the right.



Earth
Distance: 1 AU, Radius: 1, Mass: 1, Day: 1 day, Year: 365 days

Mars
Distance: 1.5 AU, Radius: 0.53, Mass: 0.11, Day: 1.026 days, Year: 687 days

Jupiter
Distance: 5.2 AU, Radius: 11, Mass: 318, Day: 0.411 days, Year: 11.8 years

Saturn
Distance: 9.5 AU, Radius: 9, Mass: 95, Day: 0.43 days, Year: 29.5 years

The following code shows the `ItemTemplate` used by the program's `ListBox`:

Available for download on Wrox.com

```
<ListBox.ItemTemplate>
    <DataTemplate>
        <DataTemplate.Resources>
            <!--  The ListBoxItem style must be set
                  in ListBox.Resources.  -->
            <Style TargetType="TextBlock">
                <Setter Property="Margin" Value="3"/>
                <Setter Property="HorizontalAlignment" Value="Left"/>
                <Setter Property="VerticalAlignment" Value="Center"/>
                <Setter Property="FontSize" Value="20"/>
                <Setter Property="FontWeight" Value="Bold"/>
                <Setter Property="Foreground" Value="Blue"/>
            </Style>
            <Style TargetType="Image">
                <Setter Property="Height" Value="50"/>
                <Setter Property="Margin" Value="3"/>
                <Setter Property="Stretch" Value="Uniform"/>
                <Setter Property="HorizontalAlignment" Value="Right"/>
            </Style>
            <Style TargetType="TextBox">
                <Setter Property="Margin" Value="3"/>
                <Setter Property="Width" Value="250"/>
                <Setter Property="Background" Value="SkyBlue"/>
                <Setter Property="TextWrapping" Value="Wrap"/>
                <Setter Property="IsReadOnly" Value="True"/>
            </Style>
```

```
                </DataTemplate.Resources>
                <StackPanel>
                    <Grid>
                        <TextBlock Text="{Binding Name}"/>
                        <Image Source="{Binding Picture}"
                        Height="50"/>
                    </Grid>
                    <TextBox Text="{Binding Stats}"/>
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
```

WPF Programmer's Reference 9780470477229 Planets

Most of this code uses `Styles` to define properties for the controls that will display the item's data. The `StackPanel` at the end contains those controls. The controls that actually display the data are:

- A `TextBlock` bound to the `Planet` item's `Name` property
- An `Image` bound to the `Planet`'s `Picture` property
- A `TextBox` bound to the `Planet`'s `Stats` property

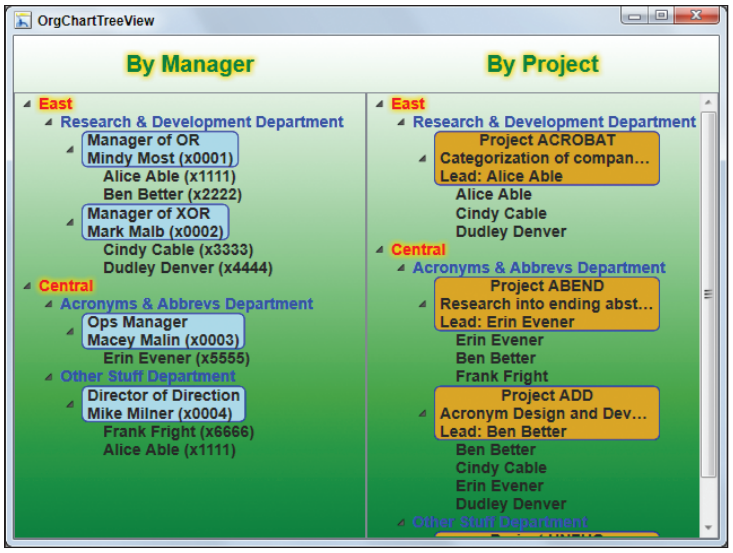
TreeView Templates

The `TreeView` control works a bit differently than the `ListBox` and `ComboBox` because it displays hierarchical data instead of items in a simple list.

In a `TreeView`, you must specify two things at each node in the tree — the data to display for that node and the path to follow deeper into the tree.

The OrgChartTreeView example program shown in the next figure displays a company organizational chart with departments displayed either by managers or by projects. To make understanding the items easier, region names are red, department names are blue and end in the word *department*, manager names are shown in blue boxes, project names are shown in goldenrod boxes, and employee names are plain black text.

The `TreeView` on the left shows regions, each region's departments, each department's managers, and each manager's direct report employees. The `TreeView` on the right shows regions, each region's departments, each department's projects, and each project's team members.



The TreeView controls that produce these two displays include HierarchicalDataTemplate objects in their Resources sections to describe what the control should do at each node. For example, the following code shows how the controls handle Region objects:

Available for download on Wrox.com

```
<HierarchicalDataTemplate
 DataType="{x:Type local:Region}"
 ItemsSource="{Binding Path=Departments}">
    <TextBlock Text="{Binding Path=RegionName}" Foreground="Red">
        <TextBlock.BitmapEffect>
            <OuterGlowBitmapEffect/>
        </TextBlock.BitmapEffect>
    </TextBlock>
</HierarchicalDataTemplate>
```

WPF Programmer's Reference 9780470477229 OrgChartTreeView

The TreeView on the left in the previous figure uses the following HierarchicalDataTemplate to display data for a Department object:

Available for download on Wrox.com

```
<HierarchicalDataTemplate
 DataType="{x:Type local:Department}"
 ItemsSource="{Binding Path=Managers}">
    <TextBlock Text="{Binding Path=Name}" Foreground="Blue"/>
</HierarchicalDataTemplate>
```

WPF Programmer's Reference 9780470477229 OrgChartTreeView

Contrast that code with the following code used by the TreeView on the right in the previous figure:

Available for download on Wrox.com

```
<HierarchicalDataTemplate
 DataType="{x:Type local:Department}"
 ItemsSource="{Binding Path=Projects}">
    <TextBlock Text="{Binding Path=Name}" Foreground="Blue"/>
</HierarchicalDataTemplate>
```

WPF Programmer's Reference 9780470477229 OrgChartTreeView

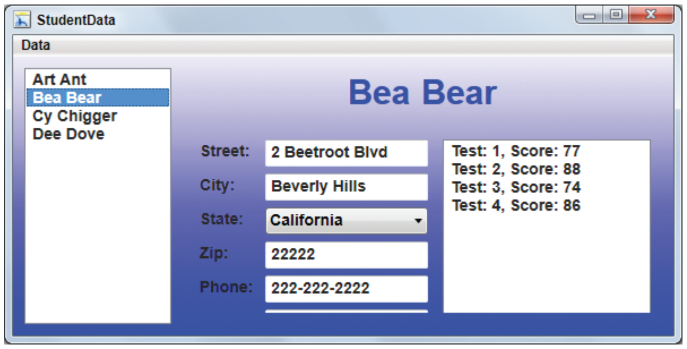
Binding Database Objects

ADO.NET provides objects that interact with relational databases. Because they are objects, you can bind them to controls much as you can bind the objects described earlier in this section. Because these objects are more complicated than those you would typically build yourself, binding them is a little more complicated.

The basic strategy is to open the database in code-behind as you would for any other ADO.NET application. Load data into DataSet, DataTable, and other ADO.NET objects, and then bind them to the application's controls.

You can think of a DataTable as similar to a collection of DataRow objects and a DataSet as similar to a collection of DataTables, so the binding syntax isn't totally unfamiliar.

The StudentData example program shown in the next figure demonstrates ADO.NET binding. This program loads its data from the StudentData.mdb Microsoft Access database.



Art Ant
Bea Bear
Cy Chigger
Dee Dove

Bea Bear

Street: 2 Beetroot Blvd
City: Beverly Hills
State: California
Zip: 22222
Phone: 222-222-2222

Test: 1, Score: 77
Test: 2, Score: 88
Test: 3, Score: 74
Test: 4, Score: 86

The `ListBox` on the left lists the students in the `Students` table. When you click on a student, the controls in the middle display the student's address and phone number, and the `ListBox` on the right displays the corresponding test score data from the `TestScores` table.

If you modify any of the student values in the middle, the controls automatically update the corresponding ADO.NET objects. If you use the Data menu's "Save Changes" command, the program writes those changes back into the database. (This example doesn't provide a way to add or delete students, or to add or modify a student's test scores.)

The database has a third table named `States` that lists the U.S. states (plus Washington, DC) and their abbreviations. The program uses this table as a lookup table. The `Students` table contains each student's state as an abbreviation. When it displays that value, the `ComboBox` in the middle of the window uses the `States` table to convert the abbreviation into the state's full name.

TRANSFORMATIONS

A *transformation* alters an object's geometry before it is drawn. Different kinds of transformations stretch, rotate, squash, skew, and move an object.

WPF provides four basic kinds of transformations represented by the following XAML elements:

- `RotateTransform` — This transformation rotates an object. The `Angle` property determines the number of degrees by which the object is rotated clockwise.
- `ScaleTransform` — This transformation scales the object vertically and horizontally. The `ScaleX` and `ScaleY` properties determine the horizontal and vertical scale factors respectively.
- `SkewTransform` — This transformation skews the object by rotating its X and Y axes through an angle given by the `AngleX` and `AngleY` properties.
- `TranslateTransform` — This transformation moves the object. The `X` and `Y` properties determine how far the object is moved horizontally and vertically.

To apply a transformation to an object, give that object a `LayoutTransform` or `RenderTransform` property element that contains one of the four basic transformations. When you use a `LayoutTransform`, WPF modifies the control before it arranges the controls. When you use a `RenderTransform`, WPF arranges the controls first and then modifies the control.

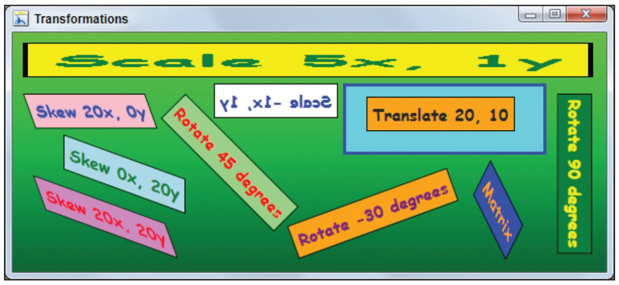
For example, the following code creates a `Label`. The `Label.LayoutTransform` element contains a `RotateTransform` that rotates the `Label` by 45 degrees.

Available for download on Wrox.com

```
<Label Canvas.Left="140" Canvas.Top="50"
 Background="LightGreen" Foreground="Red"
 Content="Rotate 45 degrees">
    <Label.LayoutTransform>
        <RotateTransform Angle="45"/>
    </Label.LayoutTransform>
</Label>
```

WPF Programmer's Reference 9780470477229 Transformations

The Transformations example program shown here demonstrates an assortment of transformations.



EFFECTS

Much as transformations modify an object's geometry, *bitmap effects* modify the way in which an object is drawn.

To add an effect to an object, give it a `BitmapEffect` property element that contains an effect object. For example, the following code defines a `Canvas` that holds an `Ellipse` and a `Path`. The `Canvas.BitmapEffect` element holds a `DropShadowBitmapEffect` object to give the result a drop shadow.

Available for download on Wrox.com

```
<Canvas Width="100" Height="100">
    <Canvas.BitmapEffect>
        <DropShadowBitmapEffect/>
    </Canvas.BitmapEffect>
    <Ellipse Stroke="Blue" Fill="Yellow" Width="100" Height="100"/>
    <Path Data="M 20,50 A 30,30 180,1,0 80,50"
     Stroke="Blue" StrokeThickness="2"/>
</Canvas>
```

WPF Programmer's Reference 9780470477229 Effects

The following list summarizes WPF's bitmap effect classes:

- `BevelBitmapEffect` — Adds beveled edges to the object.
- `BlurBitmapEffect` — Blurs the object. `Radius` determines how large the blurring is.
- `DropShadowBitmapEffect` — Adds a drop shadow behind the object.

- `EmbossBitmapEffect` — Embosses the object.
- `OuterGlowBitmapEffect` — Adds a glowing aura around the object.

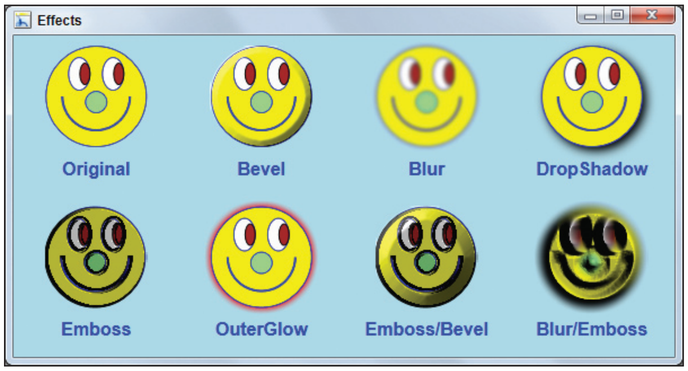
If you want to use more than one effect at the same time, use a `BitmapEffect` element that contains a `BitmapEffectGroup`. Inside the `BitmapEffectGroup`, place the effects that you want to use. For example, the following code makes a `Canvas` use an `EmbossBitmapEffect` followed by a `BevelBitmapEffect`.

Available for download on Wrox.com

```
<Canvas.BitmapEffect>
    <BitmapEffectGroup>
        <EmbossBitmapEffect/>
        <BevelBitmapEffect BevelWidth="10"/>
    </BitmapEffectGroup>
</Canvas.BitmapEffect>
```

WPF Programmer's Reference 9780470477229 Effects

The Effects example program shown in the next figure displays an unmodified object and the same object using the five basic bitmap effects. The final version showe the object using combinations of effects.



DOCUMENTS

WPF documents can contain a wide variety of objects such as:

- Paragraphs
- Tables
- Lists
- Floaters
- Figures
- User interface elements such as `Buttons` and `TextBoxes`
- Three-dimensional objects

WPF has two different kinds of documents: fixed documents and flow documents.

Fixed Documents

A *fixed document* displays its contents in exactly the same size and position whenever you view it. If you resize the control that holds the document, parts of the document may not fit, but they won't be moved. In that sense, this kind of document is similar to a PDF (Portable Document Format) file.

Building XPS Documents

When it defined fixed documents, Microsoft also defined *XML Paper Specification* (XPS) files. An *XPS file* is a fixed document saved in a special format that can be read and displayed by certain programs such as recent versions of Internet Explorer.

One of the most flexible ways you can make an XPS document is to use Microsoft Word. Simply create a Word document containing any text, graphics, lists, or other content that you want, and then export it as an XPS document.

Microsoft Word 2007 can export files in XPS (or PDF) format, but you need to install an add-in first. As of this writing, you can install the add-in by following the instructions on Microsoft's "2007 Microsoft Office Add-in: Microsoft Save as PDF or XPS" download page at `r.office.microsoft.com/r/rlidMSAddinPDFXPS`. (If the page has moved so you can't find it, go to Microsoft's Download Center at `www.microsoft.com/downloads` and search for "Save as PDF or XPS.")

After you install the add-in, simply click on the Windows icon in Word's upper-left corner to expand the main file menu. Click on the arrow next to the "Save As" command, and select the "PDF or XPS" item.

If you don't have Microsoft Word but are running the Microsoft Windows 7 operating system, you can still create XPS documents relatively easily. Use Notepad, WordPad, or some other editor to create the document. Next print it, selecting Microsoft XPS Document Writer as the printer. The writer will prompt you for the file where you want to save the result.

You can even display a web page in a browser such as Firefox or Internet Explorer, print it, and select Microsoft XPS Document Writer as the printer.

Displaying XPS Documents

After you have created an XPS file, you have several options for displaying it.

The Microsoft XPS Viewer, which is installed by default in Windows 7 and integrated into Internet Explorer 6.0 and higher, can display XPS files. By default, .xps files are associated with Internet Explorer, so you can probably just double-click on the file to view it.

You can also download Microsoft XPS Viewer or Microsoft XPS Essentials Pack (which contains a stand-alone viewer) at `www.microsoft.com/whdc/xps/viewxps.mspx`.

Finally, you can make a WPF application display an XPS file inside a `DocumentViewer` control. To do that, first add a reference to the ReachFramework library.

In Visual Studio, open the Project menu and select "Add Reference." On the Add Reference dialog, select the .NET tab, select the ReachFramework entry, and click OK.

Now you can write code-behind to load an XPS document into a `DocumentViewer` control. To make using the necessary classes easier, you can add the following `using` statement to the program:

```
using System.Windows.Xps.Packaging;
```

The following code loads the file "Fixed Document.xps" when the program's window loads:

Available for download on Wrox.com

```
// Load the XPS document.
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Note: Mark the file as "Copy if newer"
    // to make a copy go in the output directory.
    XpsDocument xps_doc = new XpsDocument("Fixed Document.xps",
        System.IO.FileAccess.Read);
    docViewer.Document = xps_doc.GetFixedDocumentSequence();
}
```

WPF Programmer's Reference 9780470477229 ViewXpsDocument

Building Fixed Documents in XAML

Building an XPS document is easy with Microsoft Word, WordPad, or some other external program, but sometimes it's more convenient to build a fixed document in XAML or code-behind. For example, you might want a program to generate a report at run time and display it as a fixed document.

Building a fixed document in XAML code isn't too complicated, although it is a lot of work because it's a fairly verbose format.

Start by creating a `DocumentViewer` to hold the document. Give it any properties and resources it needs (such as a background), and place a `FixedDocument` object inside it.

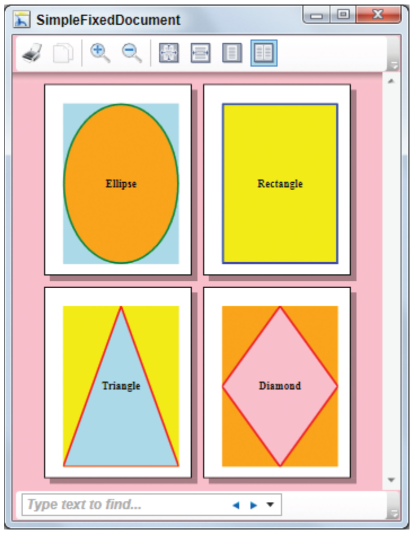
The `FixedDocument` object should contain one or more `PageContent` objects that define its pages.

You can set each `PageContent` object's `Width` and `Height` properties to determine the size of the page.

Inside the `PageContent`, place a `FixedPage` object to define the page's contents.

Inside the `FixedPage`, you can place controls such as `StackPanel`, `Grid`, `TextBlock`, and `Border` to create the page's contents.

The SimpleFixedDocument example program shown in the next figure draws four fixed pages that contain simple shapes and labels.



Saving XPS Files

Having gone to the trouble of building a fixed document in XAML or code-behind, you might want to save it as an XPS file.

To save a fixed document into an XPS file, open a project in Visual Studio and add references to the ReachFramework and System.Printing libraries. To make using the libraries easier, you can add the following `using` statements to the program:

```
using System.Windows.Xps;
using System.Windows.Xps.Packaging;
```

Next, create an `XpsDocument` object to write into the file that you want to create. Make an `XpsDocumentWriter` associated with the document object, and use its `Write` method to write the `FixedDocument` into the file.

The SaveFixedDocument example program uses the following code to save its `FixedDocument` object named `fdContents` into a file.

Available for download on Wrox.com

```
// Save as an XPS file.
private void mnuFileSave_Click(System.Object sender, System.Windows.RoutedEventArgs e)
{
    // Get the file name.
    Microsoft.Win32.SaveFileDialog dlg =
        new Microsoft.Win32.SaveFileDialog();
    dlg.FileName = "Shapes";
    dlg.DefaultExt = ".xps";
    dlg.Filter = "XPS Documents (.xps)|*.xps|All Files (*.*)|*.*";
    if (dlg.ShowDialog() == true)
    {
        // Save the document.
        // Make an XPS document.
        XpsDocument xps_doc = new XpsDocument(dlg.FileName,
            System.IO.FileAccess.Write);

        // Make an XPS document writer.
        XpsDocumentWriter doc_writer =
            XpsDocument.CreateXpsDocumentWriter(xps_doc);
        doc_writer.Write(fdContents);
        xps_doc.Close();
    }
}
```

WPF Programmer's Reference 9780470477229 SavedFixedDocument

Flow Documents

A *flow document* rearranges its contents as necessary to fit the container that is holding it. If you make the viewing control tall and thin, the document reorganizes its contents to fit. In that sense, this kind of document is similar to a simple web page that rearranges its contents when you resize the browser.

A WPF program displays flow documents inside one of three kinds of viewers: FlowDocumentPageViewer, FlowDocumentReader, or FlowDocumentScrollViewer. S

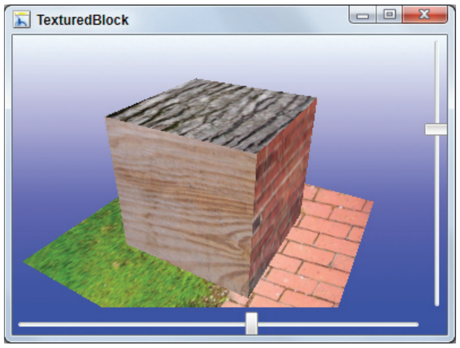
A FlowDocument object represents the flow document itself. The FlowDocument's children must be objects that are derived from the Block class. These include BlockUIContainer, List, Paragraph, Section, and Table.

THREE-DIMENSIONAL DRAWING

WPF relies on Microsoft's DirectX technologies for high-quality multimedia support.

One of my favorite parts of DirectX is *Direct3D*, the three-dimensional (3D) drawing library.

The TexturedBlock example program shown in the next figure displays a simple cube with sides made of various materials sitting on a brick and grass surface. You can use the sliders to rotate the scene in real time.



Unfortunately, writing Direct3D code to display these types of objects is fairly involved. One of the first and most annoying tasks is figuring out what graphics hardware is available on the user's computer.

Fortunately, WPF's 3D drawing objects handle this setup for you.

That doesn't mean that the rest of the process is easy. Producing a 3D scene requires a fair amount of work because there are many complicated factors to consider, but at least WPF takes care of initialization and redrawing.

The following section explains the basic structure and geometry of a simple 3D scene. The sections after that describe cameras, lighting, and materials in greater detail.

Basic Structure

The Viewport3D control displays a 3D scene. You can think of it as a window leading into 3D space.

The Viewport3D object's Camera property defines the camera used to view the scene.

The Viewport3D should contain one or more ModelVisual3D objects that define the items in the scene. The ModelVisual3D's Content property should contain the visual objects.

Typically, the `Content` property holds either a single `GeometryModel3D` object that defines the entire scene or a `Model3DGroup` object that holds a collection of `GeometryModel3D` objects.

Each `GeometryModel3D` object can define any number of triangles, so a single `GeometryModel3D` object can produce a very complicated result. Since the triangles don't even need to be connected to each other, a `GeometryModel3D` could define several separate physical objects.

The catch is that a single `GeometryModel3D` can only have one material, so any separate objects would have a similar appearance.

The "Materials" section later describes materials in greater detail.

The `GeometryModel3D` object's two most important properties are `Material` (which was just described) and `Geometry`. The `Geometry` property should contain a single `MeshGeometry3D` object that defines the triangles that make up the object.

`MeshGeometry3D` has four key properties that define its triangles: `Positions`, `TriangleIndices`, `Normals`, and `TextureCoordinates`. The following sections describe these properties.

Positions

The `MeshGeometry3D`'s `Positions` property is a list of 3D point coordinate values. You can separate the coordinates in the list by spaces or commas.

For example, the value "`1,0,1 -1,0,1 1,0,-1 -1,0,-1`" defines the four points in the Y = 0 plane where X and Z are 1 or –1. This defines a square two units wide centered at the origin.

TriangleIndices

The `TriangleIndices` property gives a list of indexes into the `Positions` array that give the points that make up the object's triangles. Note that the triangles are free to share the points defined by the `Positions` property.

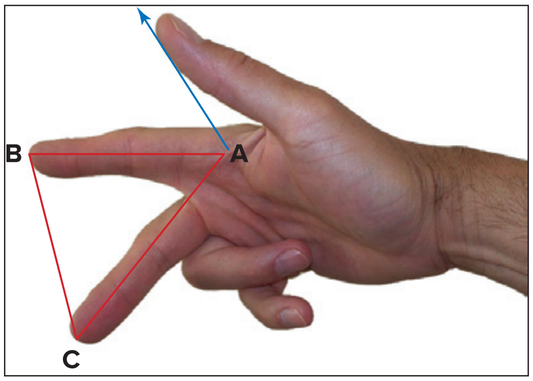
Outward Orientation

A vector that points perpendicularly to a triangle is called a *normal* or *surface normal* for the triangle (or any other surface for that matter). Sometimes people require that the normal have length 1. Alternatively, they may call a length 1 normal a *unit normal*.

Note that a triangle has two normals that point in opposite directions. If the triangle is lying flat on a table, then one normal points toward the ceiling, and the other points toward the floor.

The right-hand rule lets you use the order of the points that define the triangle to determine which normal is which.

To use the right-hand rule, picture the triangle ABC that you are building in 3D space. Place your right index finger along the triangle's first segment AB as shown in the next figure. Then bend your middle finger inward so it lies along the segment AC. If you've done it right, then your thumb (the blue arrow) points toward the triangle's "outside." The *outer normal* is the normal that lies on the same side of the triangle as your thumb.



Why should you care about the right-hand rule and outwardly-oriented normals? Direct3D uses the triangle's orientation to decide whether it should draw the triangle. If the outwardly-oriented normal points *toward* the camera's viewing position, then Direct3D draws the triangle. If the outwardly-oriented normal points *away from* the camera's viewing position, then Direct3D doesn't draw the triangle.

Normals

As you'll see in the section "Lighting" later, a triangle's normal not only determines whether it's visible, but it also helps determine the triangle's color. That means more accurate normals give more accurate colors.

Left to its own devices, Direct3D finds a triangle's normal by performing some calculations using the points that define the triangle. (It lines up its virtual fingers to apply the right-hand rule.)

For objects defined by flat surfaces such as cubes, octahedrons, and other polyhedrons, that works well. For smooth surfaces such as spheres, cylinders, and torii (donuts), it doesn't work as well because the normal at one part of a triangle on the object's surface points in a slightly different direction from the normals at other points.

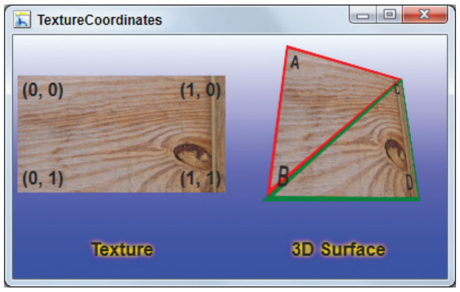
The `MeshGeometry3D` object's `Normals` property lets you tell the drawing engine what normals to use for the points defined by the object's `Positions` property. The engine still uses the calculated normal to decide whether to draw a triangle, but it uses the normals you supply to color the triangle.

TextureCoordinates

The `TextureCoordinates` property is a collection that determines how points are mapped to positions on a material's surface. You specify a point's position on the surface by giving the coordinates of the point on the brush used by the material to draw the surface.

The coordinates on the brush begin with (0, 0) in the upper left with the first coordinate extending to the right and the second extending downward.

The next figure shows the idea graphically.



The picture on the left shows the texture material with its corners labeled in the U–V coordinate system.

The picture on the right shows a `MeshGeometry3D` object that defines four points (labeled A, B, C, and D) and two triangles (outlined in red and green).

The following code shows the `MeshGeometry3D`'s definition. The corresponding `Positions` and `TextureCoordinates` entries map the points A, B, C, and D to the U–V coordinates (0, 0), (0, 1), (1, 0), and (1, 1), respectively.

Available for download on Wrox.com

```
<MeshGeometry3D
    Positions="-1,1,1 -1,-1,1 1,1,1 1,-1,1"
    TriangleIndices="0,1,2 2,1,3"
    TextureCoordinates="0,0 0,1 1,0 1,1"
 />
```

WPF Programmer's Reference 9780470477229 TextureCoordinates

The TexturedBlock program shown in the previous figure uses similar code to map textures to points for all of its surfaces.

Cameras

The camera determines the location and direction from which a 3D scene is viewed. You can think of the camera as if it were a motion picture camera pointed at a scene. The camera is in some position (possibly on a boom, a crane, or in a cameraperson's hand) pointed toward some part of the scene.

The following camera properties let you specify the camera's location and orientation:

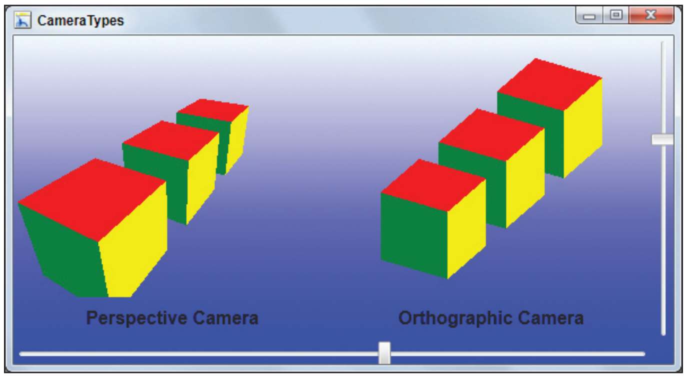
- `Position` — Gives the camera's coordinates in 3D space.
- `LookDirection` — Gives the direction in which the camera should be pointed relative to its current position. For example, if the camera's `Position` is `"1, 1, 1"` and `LookDirection` is `"1, 2, 3"`, then the camera is pointed at the point (1 + 1, 1 + 2, 1 + 3) = (2, 3, 4).
- `UpDirection` — Determines the camera's *roll* or *tilt*. For example, you might tilt the camera sideways or at an angle.

The two most useful kinds of cameras in WPF are perspective and orthographic.

In a *perspective view*, parallel lines seem to merge toward a vanishing point and objects farther away from the camera appear smaller. Since this is similar to the way you see things in real life, the result of a parallel camera is more realistic.

In an *orthographic view*, parallel lines remain parallel and objects that have the same size appear to have the same size even if one is farther from the camera than another. While this is less *realistic*, orthographic views can be useful for engineering diagrams and other drawings where you might want to perform measurements.

The CameraTypes example program shown in the next figure displays images of the same scene with both kinds of cameras.



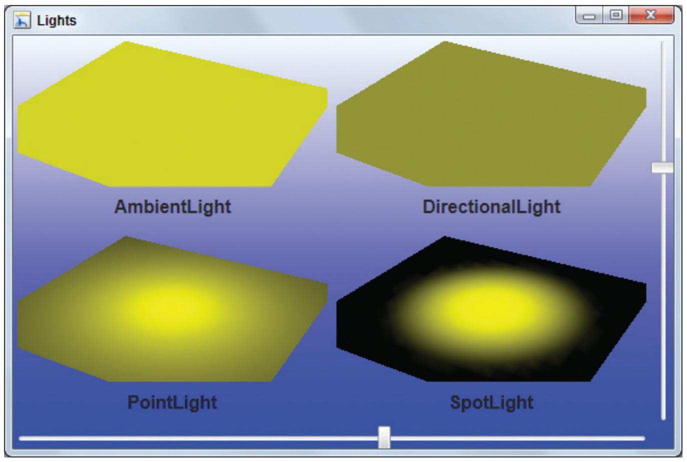
Lighting

The color that you see in a scene depends on both the lights in the scene and on the materials used by the objects. The next section discusses materials in detail. This section considers only lights.

WPF provides several kinds of lights that provide different effects. The following list summarizes these kinds of lights:

- **Ambient Light** — This is light that comes from all directions and hits every surface equally. It's the reason you can see what's under your desk even if no light is shining directly there. Most scenes need at least some ambient light.
- **Directional Light** — This light shines in a particular direction as if the light is infinitely far away. Light from the Sun is a close approximation of directional light, at least on a local scale, because the Sun is practically infinitely far away compared to the objects near you.
- **Point Light** — This light originates at a point in space and shines radially on the objects around it. Note that the light itself is invisible, so you won't see a bright spot as you would if you had a real lightbulb in the scene.
- **Spot Light** — This light shines a cone into the scene. Objects directly in front of the cone receive the full light, with objects farther to the sides receiving less.

The Lights example program shown in the next figure demonstrates the different kinds of lights. Each scene shows a yellow square (the corners are cropped by the edges of the viewport) made up of lots of little triangles.



Materials

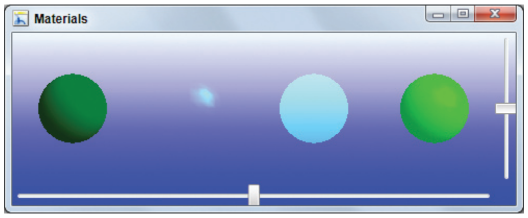
As previous sections have explained, the exact color given to a triangle in a scene depends on the light and the angle at which the light hits the triangle. The result also depends on the triangle's type of material. WPF provides three kinds of materials: diffuse, specular, and emissive.

A *diffuse* material's brightness depends on the angle at which light hits it, but the brightness does not depend on the angle at which you view it.

A *specular* material is somewhat shiny. In that case, an object's apparent brightness depends on how closely the angle between you, the object, and the light sources matches the object's *mirror angle*. The *mirror angle* is the angle at which most of the light would bounce off the object if it were perfectly shiny.

The final type of material, an *emissive* material, glows. An emissive material glows but only on itself. In other words, it makes its own object brighter, but it doesn't contribute to the brightness of other nearby objects as a light would.

The Materials example program shown in the next figure shows four identical spheres made of different materials. From left to right, the materials are diffuse, specular, emissive, and a `MaterialGroup` combining all three types of materials.



The following code shows how the program creates its spheres:

Available for download on Wrox.com

```
MakeSingleMeshSphere(Sphere00, new DiffuseMaterial(Brushes.Green), 1, 20, 30);
MakeSingleMeshSphere(Sphere01, new SpecularMaterial(Brushes.Green, 50), 1, 30, 30);
MakeSingleMeshSphere(Sphere02, new EmissiveMaterial(Brushes.DarkGreen), 1, 20, 30);

MaterialGroup combined_material = new MaterialGroup();
combined_material.Children.Add(new DiffuseMaterial(Brushes.Green));
combined_material.Children.Add(new SpecularMaterial(Brushes.Green, 50));
combined_material.Children.Add(new EmissiveMaterial(Brushes.DarkGreen));
MakeSingleMeshSphere(Sphere03, combined_material, 1, 20, 30);
```

WPF Programmer's Reference 9780470477229 Materials

Building Complex Scenes

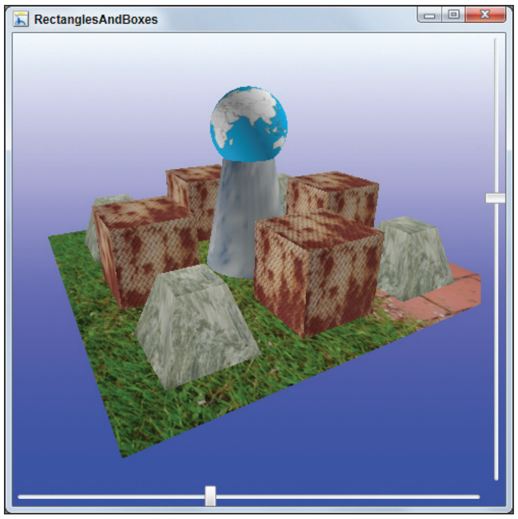
Throughout this book, I've tried to use XAML code as much as possible.

XAML code, however, will only get you so far when you're building 3D scenes. XAML is just fine for simple scenes containing a dozen or so triangles that display cubes, tetrahedrons, and other objects with large polygonal faces, but building anything really complex in XAML can be difficult.

For example, the spheres shown in the previous figure each use 1,140 triangles. While in principle you could define all of those triangles in XAML code by hand, in practice that would be extremely difficult and time-consuming.

To make building complex scenes easier, it's helpful to have a library of code-behind routines that you can use to build these more complex shapes.

The RectanglesAndBoxes example program shown in the next figure demonstrates routines that draw textured rectangles (the ground), boxes (the truncated pyramids and cubes), cylinders and cones (the truncated cone under the globe), and spheres (the globe).



Unfortunately, some of these routines are fairly long and complicated (drawing a sphere and mapping a texture onto it takes some work), so they are not shown here. Download the example program from the book's web page to see the details.

ABOUT THE AUTHORS

CHRISTIAN NAGEL is a Microsoft Regional Director and Microsoft MVP, an associate of thinktecture, and owner of CN innovation. He is a software architect and developer who offers training and consulting on how to develop Microsoft .NET solutions. He looks back on more than 25 years of software development experience. Christian started his computing career with PDP 11 and VAX/VMS systems, covering a variety of languages and platforms. Since 2000, when .NET was just a technology preview, he has been working with various .NET technologies to build numerous .NET solutions. With his profound knowledge of Microsoft technologies, he has written numerous .NET books, and is certified as a Microsoft Certified Trainer and Professional Developer. Christian speaks at international conferences such as TechEd and Tech Days, and started INETA Europe to support .NET user groups. You can contact Christian via his web sites `www. cninnovation.com` and `www.thinktecture.com` and follow his tweets on `www.twitter.com/christiannagel`.

BILL EVJEN is an active proponent of .NET technologies and community-based learning initiatives for .NET. He has been actively involved with .NET since the first bits were released in 2000. In the same year, Bill founded the St. Louis .NET User Group (`www.stlnet.org`), one of the world's first such groups. Bill is also the founder and former executive director of the International .NET Association (`www.ineta.org`), which represents more than 500,000 members worldwide.

Based in St. Louis, Missouri, Bill is an acclaimed author and speaker on ASP.NET and Services. He has authored or coauthored more than 20 books including *Professional ASP.NET 4*, *Professional VB 2008*, *ASP.NET Professional Secrets*, *XML Web Services for ASP.NET*, and *Web Services Enhancements: Understanding the WSE for Enterprise Applications* (all published by Wiley). In addition to writing, Bill is a speaker at numerous conferences, including DevConnections, VSLive, and TechEd. Along with these activities, Bill works closely with Microsoft as a Microsoft Regional Director and an MVP.

Bill is the Global Head of Platform Architecture for Thomson Reuters, Lipper, the international news and financial services company (`www.thomsonreuters.com`). He graduated from Western Washington University in Bellingham, Washington, with a Russian language degree. When he isn't tinkering on the computer, he can usually be found at his summer house in Toivakka, Finland. You can reach Bill on Twitter at `@billevjen`.

JAY GLYNN is the Principle Architect at PureSafety, a leading provider of results-driven software and information solutions for workforce safety and health. Jay has been developing software for over 25 years and has worked with a variety of languages and technologies including PICK Basic, C, C++, Visual Basic, C# and Java. Jay currently lives in Franklin TN with his wife and son.

KARLI WATSON is consultant at Infusion Development (`www.infusion.com`), a technology architect at Boost.net (`www.boost.net`), and a freelance IT specialist, author, and developer. For the most part, he immerses himself in .NET (in particular C# and lately WPF) and has written numerous books in the field for several publishers. He specializes in communicating complex ideas in a way that is accessible to anyone with a passion to learn, and spends much of his time playing with new technology to find new things to teach people about.

During those (seemingly few) times where he isn't doing the above, Karli will probably be wishing he was hurtling down a mountain on a snowboard. Or possibly trying to get his novel published. Either way, you'll know him by his brightly colored clothes. You can also find him tweeting online at `www.twitter.com/karlequin`, and maybe one day he'll get round to making himself a website.

MORGAN SKINNER began his computing career at a young age on the Sinclair ZX80 at school, where he was underwhelmed by some code a teacher had written and so began programming in assembly language. Since then he's used all sorts of languages and platforms, including VAX Macro Assembler, Pascal, Modula2, Smalltalk, X86 assembly language, PowerBuilder, C/C++, VB, and currently C# (of course). He's been programming in .NET since the PDC release in 2000, and liked it so much he joined Microsoft in 2001. He now works in premier support for developers and spends most of his time assisting customers with C#. You can reach Morgan at `www.morganskinner.com`.

SCOTT HANSELMAN works for Microsoft as a Principal Program Manager Lead in the Server and Tools Online Division, aiming to spread the good word about developing software, most often on the Microsoft stack. Before this, Scott was the Chief Architect at Corillian, an eFinance enabler, for 6+ years; and before Corillian, he was a Principal Consultant at Microsoft Gold Partner for 7 years. He was also involved in a few things like the MVP and RD programs and will speak about computers (and other passions) whenever someone will listen to him. He blogs at `www.hanselman.com` and podcasts at `www.hanselminutes.com` and contributes to `www.asp.net`, `www.windowsclient.net`, and `www.silverlight.net`.

DEVIN RADER is a Product Manager on the Infragistics Web Client team, responsible for leading the creation of Infragistics ASP.NET and Silverlight products. Devin is also an active proponent and member of the .NET developer community, being a co-founder of the St. Louis .NET User Group, an active member of the New Jersey .NET User Group, a former board member of the International .NET Association (INETA), and a regular speaker at user groups. He is also a contributing author on the Wrox titles Silverlight 3 Programmer's Reference and Silverlight 1.0 and a technical editor for several other Wrox publications and has written columns for ASP.NET Pro magazine, as well as .NET technology articles for MSDN Online. You can find more of Devin's musings at `http://blogs.infragistics.com/blogs/devin_rader/`.

ROD STEPHENS started out as a mathematician, but while studying at MIT, discovered the joys of programming and has been programming professionally ever since. During his career, he has worked on an eclectic assortment of applications in such fields as telephone switching, billing, repair dispatching, tax processing, wastewater treatment, concert ticket sales, cartography, and training for professional football players.

Rod is a Microsoft Visual Basic Most Valuable Professional (MVP) and ITT adjunct instructor. He has written more than 20 books that have been translated into languages from all over the world, and more than 250 magazine articles covering Visual Basic, C#, Visual Basic for Applications, Delphi, and Java. He is currently a regular contributor to DevX (`www.DevX.com`).

Rod's popular VB Helper web site `www.vb-helper.com` receives several million hits per month and contains thousands of pages of tips, tricks, and example code for Visual Basic programmers, as well as example code for this book.

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

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
`www.wiley.com`

Copyright © 2010 by Wiley Publishing, Inc., Indianapolis, Indiana

ISBN: 978-0-470-77034-4

Code samples are available on Wrox.com under the following ISBNs:

Professional C# 4 and .NET 4   9780470502259

Professional ASP.NET 4 in C# and VB   9780470502204

WPF Programmer's Reference   9780470477229