



Professional

C# 4 and .NET 4

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

PROFESSIONAL C# 4 and .NET 4 in C# and VB

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:



```

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`:



```

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 `remainder` 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 remainder = dividend % divisor;

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

```

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}, remainder: {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

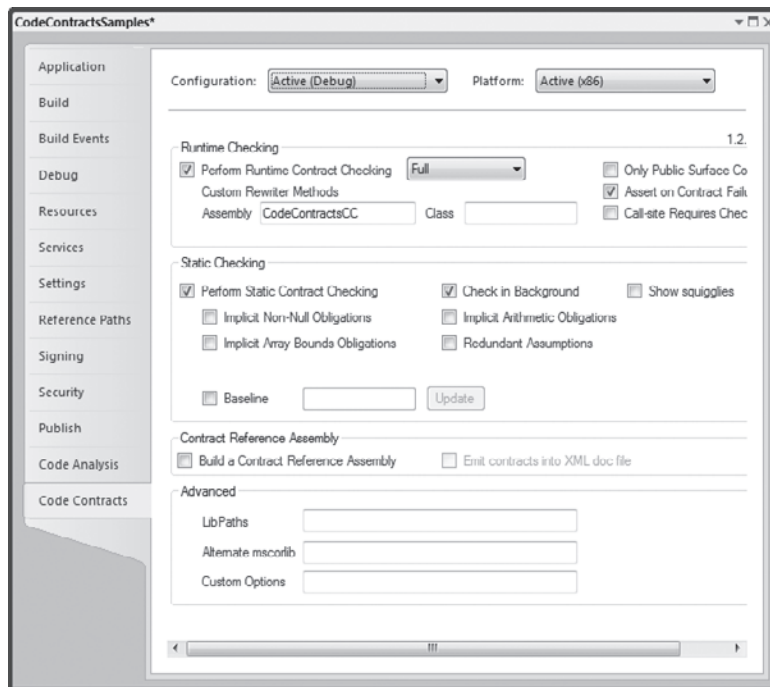
Design-by-contracts is an idea from the Eiffel programming language. Now .NET 4 includes classes for static and runtime checks of code within the namespace `System.Diagnostics.Contracts` that can be used by all .NET languages.

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.



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 PreconditionsWithLegacyCode(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.



```
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.



```
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.



```
[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.



```

[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.



```
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 remainder = div.Item1 % div.Item2;
    Console.WriteLine("task creates a result...");

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

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` 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.



```

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.



```

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.



```
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`.



```

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

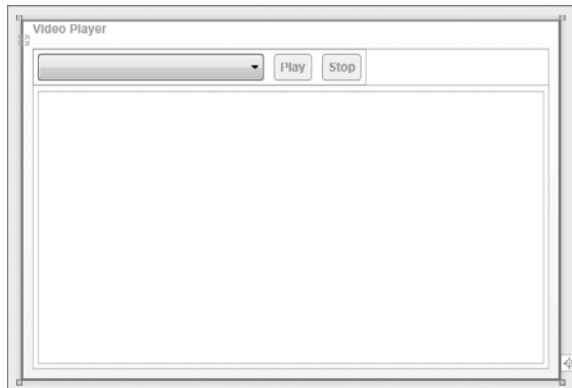
```

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 play 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 `ThumbButtonInfos` 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>
</Window.Resources>
```

```

        </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.



```

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`).

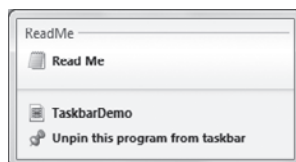
```
var jumpItems = new List<JumpTask>();
var workingDirectory = Environment.CurrentDirectory;
var windowsDirectory = Environment.GetFolderPath(
    Environment.SpecialFolder.Windows);
var notepadPath = System.IO.Path.Combine(windowsDirectory,
    "Notepad.exe");

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.