New Language Features in C# 6

Kevin Pilch-Bisson edited this page on Aug 13, 2015 · 4 revisions

This document describes the new language features in C# 6, the next version of C#. All of these are implemented and available in VS 2015.

Auto-property enhancements

Initializers for auto-properties

You can now add an initializer to an auto-property, just as you can in a field:

```
public class Customer
{
    public string First { get; set; } = "Jane";
    public string Last { get; set; } = "Doe";
}
```

The initializer directly initializes the backing field; it doesn't work through the setter of the auto-property. The initializers are executed in order as written, just as – and along with – field initializers.

Just like field initializers, auto-property initializers cannot reference `this` – after all they are executed before the object is properly initialized.

Getter-only auto-properties

Auto-properties can now be declared without a setter.

```
public class Customer
{
    public string First { get; } = "Jane";
    public string Last { get; } = "Doe";
}
```

The backing field of a getter-only auto-property is implicitly declared as `readonly` (though this matters only for reflection purposes). It can be initialized through an initializer on the property as in the example above. Also, a getter-only property can be assigned to in the declaring type's constructor body, which causes the value to be assigned directly to the underlying field:

```
public class Customer
{
    public string Name { get; }
    public Customer(string first, string last)
    {
        Name = first + " " + last;
    }
}
```

Pages 38

Home
Project Overview
Project Roadmap
Labels used for Issues
Performance considerations for large solutions

Contributors

Building, Testing, and Debugging
Contributing Code

Tool Authors

Getting Started
Samples and Walkthroughs
FAQ
API Changelist

Status

Language feature status
Edit-and-Continue Supported Edits

Interactive Docs

Interactive Window Documentation
Scripting API Samples
C# Interactive Walkthrough

Clone this wiki locally

https://github.com/dotnet/r

Clone in Desktop

This is about expressing types more concisely, but note that it also removes an important difference in the language between mutable and immutable types: auto-properties were a shorthand available only if you were willing to make your class mutable, and so the temptation to default to that was great. Now, with getter-only auto-properties, the playing field has been leveled between mutable and immutable.

Expression-bodied function members

Lambda expressions can be declared with an expression body as well as a conventional function body consisting of a block. This feature brings the same convenience to function members of types.

Expression bodies on method-like members

Methods as well as user-defined operators and conversions can be given an expression body by use of the "lambda arrow":

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public static implicit operator string(Person p) => p.First + " " + p.Last;
```

The effect is exactly the same as if the methods had had a block body with a single return statement.

For void-returning methods – and `Task` -returning async methods – the arrow syntax still applies, but the expression following the arrow must be a statement expression (just as is the rule for lambdas):

```
public void Print() => Console.WriteLine(First + " " + Last);
```

Expression bodies on property-like function members

Properties and indexers can have getters and setters. Expression bodies can be used to write getter-only properties and indexers where the body of the getter is given by the expression body:

```
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Note that there is no `get` keyword: It is implied by the use of the expression body syntax.

Using static

The feature allows all the accessible static members of a type to be imported, making them available without qualification in subsequent code:

```
using static System.Console;
using static System.Math;
using static System.DayOfWeek;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
        WriteLine(Friday - Monday);
```

```
    }
}
```

This is great for when you have a set of functions related to a certain domain that you use all the time. `System.Math` would be a common example of that. It also lets you directly specify the individual named values of an enum type, like the `System.DayOfWeek` members above.

Extension methods

Extension methods are static methods, but are intended to be used as instance methods. Instead of bringing extension methods into the global scope, the using static feature makes the extension methods of the type available as extension methods:

```
using static System.Linq.Enumerable; // The type, not the namespace
class Program
{
    static void Main()
    {
        var range = Range(5, 17);                // Ok: not extension
        var odd = Where(range, i => i % 2 == 1); // Error, not in scope
        var even = range.Where(i => i % 2 == 0); // Ok
    }
}
```

This does mean that it can now be a breaking change to turn an ordinary static method into an extension method, which was not the case before. But extension methods are generally only called as static methods in the rare cases where there is an ambiguity. In those cases, it seems right to require full qualification of the method anyway.

Null-conditional operators

Sometimes code tends to drown a bit in null-checking. The null-conditional operator lets you access members and elements only when the receiver is not-null, providing a null result otherwise:

```
int? length = customers?.Length; // null if customers is null
Customer first = customers?[0];  // null if customers is null
```

The null-conditional operator is conveniently used together with the null coalescing operator `??`:

```
int length = customers?.Length ?? 0; // 0 if customers is null
```

The null-conditional operator exhibits short-circuiting behavior, where an immediately following chain of member accesses, element accesses and invocations will only be executed if the original receiver was not null:

```
int? first = customers?[0].Orders.Count();
```

This example is essentially equivalent to:

```
int? first = (customers != null) ? customers[0].Orders.Count() : null;
```

Except that `customers` is only evaluated once. None of the member accesses, element accesses and invocations immediately following the `?` are executed unless `customers` has a non-null value.

Of course null-conditional operators can themselves be chained, in case there is a need to check for null more than once in a chain:

```
int? first = customers?[0].Orders?.Count();
```

Note that an invocation (a parenthesized argument list) cannot immediately follow the `?` operator – that would lead to too many syntactic ambiguities. Thus, the straightforward way of calling a delegate *only* if it's there does not work. However, you can do it via the `Invoke` method on the delegate:

```
if (predicate?.Invoke(e) ?? false) { … }
```

We expect that a very common use of this pattern will be for triggering events:

```
PropertyChanged?.Invoke(this, args);
```

This is an easy and thread-safe way to check for null before you trigger an event. The reason it's thread-safe is that the feature evaluates the left-hand side only once, and keeps it in a temporary variable.

String interpolation

`String.Format` and its cousins are very versatile and useful, but their use is a little clunky and error prone. Particularly unfortunate is the use of `{0}` etc. placeholders in the format string, which must line up with arguments supplied separately:

```
var s = String.Format("{0} is {1} year{{s}} old", p.Name, p.Age);
```

String interpolation lets you put the expressions right in their place, by having "holes" directly in the string literal:

```
var s = $"{p.Name} is {p.Age} year{{s}} old";
```

Just as with `String.Format`, optional alignment and format specifiers can be given:

```
var s = $"{p.Name,20} is {p.Age:D3} year{{s}} old";
```

The contents of the holes can be pretty much any expression, including even other strings:

```
var s = $"{p.Name} is {p.Age} year{(p.Age == 1 ? "" : "s")} old";
```

Notice that the conditional expression is parenthesized, so that the `: "s"` doesn't get confused with a format specifier.

nameof expressions

Occasionally you need to provide a string that names some program element: when throwing an `ArgumentNullException` you want to name the guilty argument; when raising a `PropertyChanged` event you want to name the property that changed, etc.

Using string literals for this purpose is simple, but error prone. You may spell it wrong, or a refactoring may leave it stale. nameof expressions are essentially a fancy kind of string literal where the compiler checks that you have something of the given name, and Visual Studio knows what it refers to, so navigation and refactoring will work:

```
if (x == null) throw new ArgumentNullException(nameof(x));
```

You can put more elaborate dotted names in a nameof expression, but that's just to tell the compiler where to look: only the final identifier will be used:

```
WriteLine(nameof(person.Address.ZipCode)); // prints "ZipCode"
```

Index initializers

Object and collection initializers are useful for declaratively initializing fields and properties of objects, or giving a collection an initial set of elements. Initializing dictionaries and other objects with indexers is less elegant. We are adding a new syntax to object initializers allowing you to set values to keys through any indexer that the new object has:

```
var numbers = new Dictionary<int, string> {
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

Exception filters

VB has them. F# has them. Now C# has them too. This is what they look like:

```
try { … }
catch (MyException e) when (myfilter(e))
{
    …
}
```

If the parenthesized expression evaluates to true, the catch block is run, otherwise the exception keeps going.

Exception filters are preferable to catching and rethrowing because they leave the stack unharmed. If the exception later causes the stack to be dumped, you can see where it originally came from, rather than just the last place it was rethrown.

It is also a common and accepted form of “abuse” to use exception filters for side effects; e.g. logging. They can inspect an exception “flying by” without intercepting its course. In those cases, the filter will often be a call to a false-returning helper function which executes the side effects:

```
private static bool Log(Exception e) { /* log it */ ; return false; }
…
try { … } catch (Exception e) when (Log(e)) {}
```

Await in catch and finally blocks

In C# 5 we don't allow the `await` keyword in catch and finally blocks, because we'd somehow convinced ourselves that it wasn't possible to implement. Now we've figured it out, so apparently it wasn't impossible after all.

This has actually been a significant limitation, and people have had to employ unsightly workarounds to compensate. That is no longer necessary:

```
Resource res = null;
try
{
    res = await Resource.OpenAsync(…);       // You could do this.
    …
}
catch(ResourceException e)
{
    await Resource.LogAsync(res, e);         // Now you can do this …
}
finally
{
    if (res != null) await res.CloseAsync(); // … and this.
}
```

The implementation is quite complicated, but you don't have to worry about that. That's the whole point of having async in the language.

Extension Add methods in collection initializers

When we first implemented collection initializers in C#, the `Add` methods that get called couldn't be extension methods. VB got it right from the start, but it seems we forgot about it in C#. This has been fixed: the code generated from a collection initializer will now happily call an extension method called `Add`. It's not much of a feature, but it's occasionally useful, and it turned out implementing it in the new compiler amounted to removing a check that prevented it.

Improved overload resolution

There are a number of small improvements to overload resolution, which will likely result in more things just working the way you'd expect them to. The improvements all relate to "betterness" – the way the compiler decides which of two overloads is better for a given argument.

One place where you might notice this (or rather stop noticing a problem!) is when choosing between overloads taking nullable value types. Another is when passing method groups (as opposed to lambdas) to overloads expecting delegates. The details aren't worth expanding on here – just wanted to let you know!