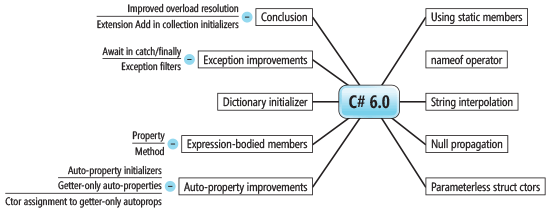
30 NOVEMBER 2015

# C# : How C# 6.0 Simplifies, Clarifies and Condenses Your Code

[Mark Michaelis](https://msdn.microsoft.com/en-us/magazine/mt149362?author=mark+michaelis) | Special connect(); issue 2014

C# 6.0 isn’t a radical revolution in C# programming. Unlike the introduction of generics in C# 2.0, C# 3.0 and its groundbreaking way to program collections with LlNQ, or the simplification of asynchronous programming patterns in C# 5.0, C# 6.0 isn’t going to transform development. That said, C# 6.0 will change the way you write C# code in specific scenarios, due to features that are so much more efficient you’ll likely forget there was another way to code them. It introduces new syntax shortcuts, reduces the amount of ceremony on occasion, and ultimately makes writing C# code leaner. In this article I’m going to delve into the details of the new C# 6.0 feature set that make all this possible. Specifically, I’ll focus on the items outlined in the Mind Map shown in **Figure 1**.

   
**Figure 1 C# 6.0 Mind Map**

Note that many of the examples herein come from the next edition of my book, “Essential C# 6.0” (Addison-Wesley Professional).

## Using Static

Many of the C# 6.0 features can be leveraged in the most basic of Console programs. For example, using is now supported on specific classes in a feature called the using static directive, which renders static methods available in global scope, without a type prefix of any kind, as shown in **Figure 2**. Because System.Console is a static class, it provides a great example of how this feature can be leveraged.

**Figure 2 The Using Static Directive Reduces Noise Within Your Code**

using System;

using static System.ConsoleColor;

using static System.IO.Directory;

using static System.Threading.Interlocked;

using static System.Threading.Tasks.Parallel;

public static class Program

{

  // ...

  public static void Main(string[] args)

  {

    // Parameter checking eliminated for elucidation.

    EncryptFiles(args[0], args[1]);

  }

  public static int EncryptFiles(

    string directoryPath, string searchPattern = "\*")

  {

    ConsoleColor color = ForegroundColor;

    int fileCount = 0;

    try

    {

      ForegroundColor = Yellow

      WriteLine("Start encryption...");

      string[] files = GetFiles(directoryPath, searchPattern,

        System.IO.SearchOption.AllDirectories);

      ForegroundColor = White

      ForEach(files, (filename) =>

      {

        Encrypt(filename);

        WriteLine("\t'{0}' encrypted", filename);

        Increment(ref fileCount);

      });

      ForegroundColor = Yellow

      WriteLine("Encryption completed");

    }

    finally

    {

      ForegroundColor = color;

    }

    return fileCount;

  }

}

In this example, there are using static directives for System.ConsoleColor, System.IO.Directory, System.Threading.Interlocked and System.Threading.Tasks.Parallel. These enable the invocation of numerous methods, properties and enums directly: ForegroundColor, WriteLine, GetFiles, Increment, Yellow, White and ForEach. In each case, this eliminates the need to qualify the static member with its type. (For those of you using Visual Studio 2015 Preview or earlier, the syntax doesn’t include adding the “static” keyword after using, so it’s only “using System.Console,” for example. In addition, not until after Visual Studio 2015 Preview does the using static directive work for enums and structs in addition to static classes.)

For the most part, eliminating the type qualifier doesn’t significantly reduce the clarity of the code, even though there is less code. WriteLine in a console program is fairly obvious as is the call to GetFiles. And, because the addition of the using static directive on System.Threading.Tasks.Parallel was obviously intentional, ForEach is a way of defining a parallel foreach loop that, with each version of C#, looks (if you squint just right) more and more like a C# foreach statement.

The obvious caution with the using static directive is to take care that clarity isn’t sacrificed. For example, consider the Encrypt function defined in **Figure 3**.

**Figure 3 Ambiguous Exists Invocation (with the nameof Operator)**

private static void Encrypt(string filename)

  {

    if (!Exists(filename)) // LOGIC ERROR: Using Directory rather than File

    {

      throw new ArgumentException("The file does not exist.",

nameof(filename));

    }

    // ...

  }

It would seem that the call to Exists is exactly what’s needed. However, more explicitly, the call is Directory.Exists when, in fact, File.Exists is what’s needed here. In other words, although the code is certainly readable, it’s incorrect and, at least in this case, avoiding the using static syntax is probably better.

Note that if using static directives are specified for both System.IO.Directory and System.IO.File, the compiler issues an error when calling Exists, forcing the code to be modified with a type disambiguation prefix in order to resolve the ambiguity.

An additional feature of the using static directive is its behavior with extension methods. Extension methods aren’t moved into global scope, as would happen with static methods normally. For example, a using static ParallelEnumerable directive would not put the Select method into global scope and you couldn’t call Select(files, (filename) => { ... }). This restriction is by design. First, extension methods are intended to be rendered as instance methods on an object (files.Select((filename)=>{ ... }) for example) and it’s not a normal pattern to call them as static methods directly off of the type. Second, there are libraries such as System.Linq, with types such as Enumerable and ParallelEnumerable that have overlapping method names like Select. To have all these types added to the global scope forces unnecessary clutter into IntelliSense and possibly introduces an ambiguous invocation (although not in the case of System.Linq-based classes).

Although extension methods won’t get placed into global scope, C# 6.0 still allows classes with extension methods in using static directives. The using static directive achieves the same as a using (namespace) directive does except only for the specific class targeted by the using static directive. In other words, using static allows the developer to narrow what extensions methods are available down to the particular class identified, rather than the entire namespace. For example, consider the snippet in **Figure 4**.

**Figure 4 Only Extension Methods from ParallelEnumerable Are in Scope**

using static System.Linq.ParallelEnumerable;

using static System.IO.Console;

using static System.Threading.Interlocked;

// ...

    string[] files = GetFiles(directoryPath, searchPattern,

      System.IO.SearchOption.AllDirectories);

    files.AsParallel().ForAll( (filename) =>

    {

      Encrypt(filename);

      WriteLine($"\t'{filename}' encrypted");

      Increment(ref fileCount);

    });

// ...

Notice there’s no using System.Linq statement. Instead, there is the using static System.Linq.ParallelEnumerable directive, causing only extension methods from ParallelEnumerable to be in scope as extension methods. All the extension methods on classes like System.Linq.Enumerable will not be available as extension methods. For example, files.Select(...) will fail compilation because Select isn’t in scope on a string array (or even IEnumerable<string>). In contrast, AsParallel is in scope via System.Linq.ParallelEnumerable. In summary, the using static directive on a class with extension methods will bring that class’ extension methods into scope as extension methods. (Non-extension methods on the same class will be brought into global scope normally.)

In general, the best practice is to limit usage of the using static directive to a few classes that are used repeatedly throughout the scope (unlike Parallel) such as System.Console or System.Math. Similarly, when using static for enums, be sure the enum items are self-explanatory without their type identifier. For example, and perhaps my favorite, specify using Microsoft.VisualStudio.TestTools.UnitTesting.Assert in unit tests files to enable test assertion invocations such as IsTrue, AreEqual<T>, Fail and IsNotNull.

## The nameof Operator

**Figure 3** includes another feature new in C# 6.0, the nameof operator. This is a new contextual keyword to identify a string literal that extracts a constant for (at compile time) the unqualified name of whatever identifier is specified as an argument. In **Figure 3**, nameof(filename) returns “filename,” the name of the Encrypt method’s parameter. However, nameof works with any programmatic identifier. For example, **Figure 5** leverages nameof to pass the property name to INotifyPropertyChanged.PropertyChanged. (By the way, using the CallerMemberName attribute to retrieve a property name for the PropertyChanged invocation is still a valid approach for retrieving the property name; see [intellitect.com/?p=11661](http://intellitect.com/?p=11661).)

**Figure 5 Using the nameof Operator for INotifyPropertyChanged.PropertyChanged**

public class Person : INotifyPropertyChanged

{

  public event PropertyChangedEventHandler PropertyChanged;

  public Person(string name)

  {

    Name = name;

  }

  private string \_Name;

  public string Name

  {

    get{ return \_Name; }

    set

    {

      if (\_Name != value)

      {

        \_Name = value;

        PropertyChangedEventHandler propertyChanged = PropertyChanged;

        if (propertyChanged != null)

        {

          propertyChanged(this,

            new PropertyChangedEventArgs(nameof(Name)));

        }

      }

    }

  }

  // ...

}

[TestClass]

public class PersonTests

{

  [TestMethod]

  public void PropertyName()

  {

    bool called = false;

    Person person = new Person("Inigo Montoya");

    person.PropertyChanged += (sender, eventArgs) =>

    {

      AreEqual(nameof(CSharp6.Person.Name), eventArgs.PropertyName);

      called = true;

    };

    person.Name = "Princess Buttercup";

    IsTrue(called);

  }

}

Notice that whether only the unqualified “Name” is provided (because it’s in scope) or the fully qualified CSharp6.Person.Name identifier is used as in the test, the result is only the final identifier (the last element in a dotted name).

By leveraging the nameof operator, it’s possible to eliminate the vast majority of “magic” strings that refer to code identifiers as long as they’re in scope. This not only eliminates runtime errors due to misspellings within the magic strings, which are never verified by the complier, but also enables refactoring tools like Rename to update all references to the name change identifier. And, if a name changes without a refactoring tool, the compiler will issue an error indicating that the identifier no longer exists.

## String Interpolation

**Figure 3** could be improved by not only specifying the exception message indicating the file wasn’t found, but also by displaying the file name itself. Prior to C# 6.0, you’d do this using string.Format in order to embed the filename into the string literal. However, composite formatting wasn’t the most readable. Formatting a person’s name, for example, required substituting placeholders based on the order of parameters as shown in the **Figure 6** assignment of message.

**Figure 6 Composite String Formatting Versus String Interpolation**

[TestMethod]

public void InterpolateString()

{

  Person person = new Person("Inigo", "Montoya") { Age = 42 };

  string message =

    string.Format("Hello!  My name is {0} {1} and I am {2} years old.",

    person.FirstName, person.LastName, person.Age);

  AreEqual<string>

    ("Hello!  My name is Inigo Montoya and I am 42 years old.", message);

  string messageInterpolated =

    $"Hello!  My name is {person.FirstName} {person.LastName} and I am

    {person.Age} years old.";

  AreEqual<string>(message, messageInterpolated);

}

Notice the alternate approach to composite formatting with the assignment to messageInterpolated. In this example, the expression assigned to messageInterpolated is a string literal prefixed with a “$” and curly brackets identify code that is embedded inline within the string. In this case, the properties of person are used to make this string significantly easier to read than a composite string. Furthermore, the string interpolation syntax reduces errors caused by arguments following the format string that are in improper order, or missing altogether and causing an exception. (In Visual Studio 2015 Preview there’s no $ character and, instead, each left curly bracket requires a slash before it. Releases following Visual Studio 2015 Preview are updated to use the $ in front of the string literal syntax instead.)

String interpolation is transformed at compile time to invoke an equivalent string.Format call. This leaves in place support for localization as before (though still with traditional format strings) and doesn’t introduce any post compile injection of code via strings.

**Figure 7** shows two more examples of string interpolation.

**Figure 7 Using String Interpolation in Place of string.Format**

public Person(string firstName, string lastName, int? age=null)

{

  Name = $"{firstName} {lastName}";

  Age = age;

}

private static void Encrypt(string filename)

{

  if (!System.IO.File.Exists(filename))

  {

    throw new ArgumentException(

      $"The file, '{filename}', does not exist.", nameof(filename));

  }

  // ...

}

Notice that in the second case, the throw statement, both string interpolation and the nameof operator are leveraged. String interpolation is what causes the ArgumentException message to include the file name (that is, “The file, ‘c:\data\missingfile.txt’ does not exist”). The use of the nameof operator is to identify the name of the Encrypt parameter (“filename”), the second argument of the ArgumentException constructor. Visual Studio 2015 is fully aware of the string interpolation syntax, providing both color coding and IntelliSense for the code blocks embedded within the interpolated string.

## Null-Conditional Operator

Although eliminated in **Figure 2** for clarity, virtually every Main method that accepts arguments requires checking the parameter for null prior to invoking the Length member to determine how many parameters were passed in. More generally, it’s a very common pattern to check for null before invoking a member in order to avoid a System.NullReferenceException (which almost always indicates an error in the programming logic). Because of the frequency of this pattern, C# 6.0 introduces the “?.” operator known as the null-conditional operator:

public static void Main(string[] args)

{

  switch (args?.Length)

  {

  // ...

  }

}

The null-conditional operator translates to checking whether the operand is null prior to invoking the method or property (Length in this case). The logically equivalent explicit code would be (although in the C# 6.0 syntax the value of args is only evaluated once):

(args != null) ? (int?)args.Length : null

What makes the null-conditional operator especially convenient is that it can be chained. If, for example, you invoke string[] names = person?.Name?.Split(' '), Split will only be invoked if both person and person.Name are not null. When chained, if the first operand is null, the expression evaluation is short-circuited, and no further invocation within the expression call chain will occur. Beware, however, you don’t unintentionally neglect additional null-conditional operators. Consider, for example, names = person?.Name.Split(' '). If there’s a person instance but Name is null, a NullReferenceException will occur upon invocation of Split. This doesn’t mean you must use a chain of null-conditional operators, but rather that you should be intentional about the logic. In the Person case, for example, if Name is validated and can never be null, no additional null-conditional operator is necessary.

An important thing to note about the null-conditional operator is that, when utilized with a member that returns a value type, it always returns a nullable version of that type. For example, args?.Length returns an int?, not simply an int. Although perhaps a little peculiar (in comparison to other operator behavior), the return of a nullable value type occurs only at the end of the call chain. The result is that calling the dot (“.”) operator on Length only allows invocation of int (not int?) members. However, encapsulating args?.Length in parenthesis—forcing the int? result via parentheses operator precedence—will invoke the int? return and make the Nullable<T> specific members (HasValue and Value) available.

The null-conditional operator is a great feature on its own. However, using it in combination with a delegate invocation resolves a C# pain point that has existed since C# 1.0. Note how in **Figure 5** I assign the PropertyChange event handler to a local copy (propertyChanged) before checking the value for null and then finally firing the event. This is the easiest thread safe way to invoke events without risking that an event unsubscribe will occur between the time the check for null occurs and when the event is fired. Unfortunately, this is non-intuitive and I frequently encounter code that doesn’t follow this pattern—with the result of inconsistent NullReferenceExceptions. Fortunately, with the introduction of the null-conditional operator in C# 6.0, this issue is resolved.

With C# 6.0, the code snippet changes from:

PropertyChangedEventHandler propertyChanged = PropertyChanged;

if (propertyChanged != null)

{

  propertyChanged(this, new PropertyChangedEventArgs(nameof(Name)));

}

to simply:

PropertyChanged?.Invoke(propertyChanged(

  this, new PropertyChangedEventArgs(nameof(Name)));

And, because an event is just a delegate, the same pattern of invoking a delegate via the null-conditional operator and an Invoke is always possible. This feature, perhaps more than any other in C# 6.0, is sure to change the way you write C# code in the future. Once you leverage null-conditional operators on delegates, chances are you’ll never go back and code the old way (unless, of course, you’re stuck in a pre-C# 6.0 world.)

Null-conditional operators can also be used in combination with an index operator. For example, when you use them in combination with a Newtonsoft.JObject, you can traverse a JSON object to retrieve specific elements, as shown in **Figure 8**.

**Figure 8 A Console Color Configuration Example**

string jsonText =

    @"{

      'ForegroundColor':  {

        'Error':  'Red',

        'Warning':  'Red',

        'Normal':  'Yellow',

        'Verbose':  'White'

      }

    }";

  JObject consoleColorConfiguration = JObject.Parse(jsonText);

  string colorText = consoleColorConfiguration[

    "ForegroundColor"]?["Normal"]?.Value<string>();

  ConsoleColor color;

  if (Enum.TryParse<ConsoleColor>(colorText, out color))

  {

    Console.ForegroundColor = colorText;

  }

It’s important to note that, unlike most collections within MSCORLIB, JObject doesn’t throw an exception if an index is invalid. If, for example, ForegroundColordoesn’t exist, JObject returns null rather than throwing an exception. This is significant because using the null-conditional operator on collections that throw an IndexOutOfRangeException is almost always unnecessary and may imply safety when no such safety exists. Returning to the snippet showing the Main and args example, consider the following:

public static void Main(string[] args)

{

  string directoryPath = args?[0];

  string searchPattern = args?[1];

  // ...

}

What makes this example dangerous is that the null-conditional operator gives a false sense of security, implying that if args isn’t null then the element must exist. Of course, this isn’t the case because the element may not exist even if args isn’t null. Because checking for the element count with args?.Length already verifies that args isn’t null, you never really need to also use the null-conditional operator when indexing the collection after checking length. In conclusion, avoid using the null-conditional operator in combination with the index operator if the index operator throws an IndexOutOfRangeException for non-existent indexes. Doing so leads to a false sense of code validity.

## Default Constructors in Structs

Another C# 6.0 feature to be cognizant of is support for a default (parameterless) constructor on a value type. This was previously disallowed because the constructor wouldn’t be called when initializing arrays, defaulting a field of type struct, or initializing an instance with the default operator. In C# 6.0, however, default constructors are now allowed with the caveat that they’re only invoked when the value type is instantiated with the new operator. Both array initialization and explicit assignment of the default value (or the implicit initialization of a struct field type) will circumvent the default constructor.

To understand how the default constructor is leveraged, consider the example of the ConsoleConfiguration class shown in **Figure 9**. Given a constructor, and its invocation via the new operator as shown in the CreateUsingNewIsInitialized method, structs will be fully initialized. As you’d expect, and as is demonstrated in **Figure 9**, constructor chaining is fully supported, whereby one constructor can call another using the “this” keyword following the constructor declaration.

**Figure 9 Declaring a Default Constructor on a Value Type**

using Newtonsoft.Json.Linq;

using System;

using System.Collections.Generic;

using System.IO;

public struct ConsoleConfiguration

{

  public ConsoleConfiguration() :

    this(ConsoleColor.Red, ConsoleColor.Yellow, ConsoleColor.White)

  {

    Initialize(this);

  }

  public ConsoleConfiguration(ConsoleColor foregroundColorError,

    ConsoleColor foregroundColorInformation,

    ConsoleColor foregroundColorVerbose)

  {

    // All auto-properties and fields must be set before

    // accessing expression bodied members

    ForegroundColorError = foregroundColorError;

    ForegroundColorInformation = foregroundColorInformation;

    ForegroundColorVerbose = foregroundColorVerbose;

  }

   private static void Initialize(ConsoleConfiguration configuration)

  {

    // Load configuration from App.json.config file ...

  }

  public ConsoleColor ForegroundColorVerbose { get; }

  public ConsoleColor ForegroundColorInformation { get; }

  public ConsoleColor ForegroundColorError { get; }

  // ...

  // Equality implementation excluded for elucidation

}

[TestClass]

public class ConsoleConfigurationTests

{

  [TestMethod]

  public void DefaultObjectIsNotInitialized()

  {

    ConsoleConfiguration configuration = default(ConsoleConfiguration);

    AreEqual<ConsoleColor>(0, configuration.ForegroundColorError);

    ConsoleConfiguration[] configurations = new ConsoleConfiguration[42];

    foreach(ConsoleConfiguration item in configurations)

    {

      AreEqual<ConsoleColor>(default(ConsoleColor),

        configuration.ForegroundColorError);

      AreEqual<ConsoleColor>(default(ConsoleColor),

        configuration.ForegroundColorInformation);

      AreEqual<ConsoleColor>(default(ConsoleColor),

        configuration.ForegroundColorVerbose);

    }

  }

  [TestMethod]

  public void CreateUsingNewIsInitialized()

  {

    ConsoleConfiguration configuration = new ConsoleConfiguration();

    AreEqual<ConsoleColor>(ConsoleColor.Red,

      configuration.ForegroundColorError);

    AreEqual<ConsoleColor>(ConsoleColor.Yellow,

      configuration.ForegroundColorInformation);

    AreEqual<ConsoleColor>(ConsoleColor.White,

      configuration.ForegroundColorVerbose);

  }

}

There’s one key point to remember about structs: all instance fields and auto-properties (because they have backing fields) must be fully initialized prior to invoking any other instance members. As a result, in the example in **Figure 9**, the constructor can’t call the Initialize method until all fields and auto-properties have been assigned. Fortunately, if a chained constructor handles all requisite initialization and is invoked via a “this” invocation, the compiler is clever enough to detect that it isn’t necessary to initialize data again from the body of the non-this invoked constructor, as shown in **Figure 9**.

## Auto-Property Improvements

Notice also in **Figure 9** that the three properties (for which there are no explicit fields) are all declared as auto-properties (with no body) and only a getter. These getter-only auto-properties are a C# 6.0 feature for declaring read-only properties that are backed (internally) by a read-only field. As such, these properties can only be modified from within the constructor.

Getter-only auto-properties are available in both structs and class declarations, but they’re especially important to structs because of the best practice guideline that structs be immutable. Rather than the six or so lines needed to declare a read-only property and initialize it prior to C# 6.0, now a single-line declaration and the assignment from within the constructor are all that’s needed. Thus, declaration of immutable structs is now not only the correct programming pattern for structs, but also the simpler pattern—a much appreciated change from prior syntax where coding correctly required more effort.

A second auto-property feature introduced in C# 6.0 is support for initializers. For example, I can add a static DefaultConfig auto-property with initializer to ConsoleConfiguration:

// Instance property initialization not allowed on structs.

static private Lazy<ConsoleConfiguration> DefaultConfig{ get; } =

  new Lazy<ConsoleConfiguration>(() => new ConsoleConfiguration());

Such a property would provide a single instance factory pattern for accessing a default ConsoleConfigurtion instance. Notice that rather than assigning the getter only auto-property from within the constructor, this example leverages System.Lazy<T> and instantiates it as an initializer during declaration. The result is that once the constructor completes, the instance of Lazy<ConsoleConfiguration> will be immutable and an invocation of DefaultConfig will always return the same instance of ConsoleConfiguration.

Note that auto-property initializers aren’t allowed on instance members of structs (although they’re certainly allowed on classes).

## Expression Bodied Methods and Auto-Properties

Another feature introduced in C# 6.0 is expression bodied members. This feature exists for both properties and methods and allows the use of the arrow operator (=>) to assign an expression to either a property or method in place of a statement body. For example, because the DefaultConfig property in the previous example is both private and of type Lazy<T>, retrieving the actual default instance of ConsoleConfiguration requires a GetDefault method:

static public ConsoleConfiguration GetDefault() => DefaultConfig.Value;

However, in this snippet, notice there’s no statement block type method body. Rather, the method is implemented with only an expression (not a statement) prefixed with the lambda arrow operator. The intent is to provide a simple one-line implementation without all the ceremony, and one that’s functional with or without parameters in the method signature:

private static void LogExceptions(ReadOnlyCollection<Exception> innerExceptions) =>

  LogExceptionsAsync(innerExceptions).Wait();

In regard to properties, note that the expression bodies work only for read-only (getter-only) properties. In fact, the syntax is virtually identical to that of the expression bodied methods except that there are no parentheses following the identifier. Returning to the earlier Person example, I could implement read-only FirstName and LastName properties using expression bodies, as shown in **Figure 10**.

**Figure 10 Expression Bodied Auto-Properties**

public class Person

{

  public Person(string name)

  {

    Name = name;

  }

  public Person(string firstName, string lastName)

  {

    Name = $"{firstName} {lastName}";

    Age = age;

  }

  // Validation ommitted for elucidation

  public string Name {get; set; }

  public string FirstName => Name.Split(' ')[0];

  public string LastName => Name.Split(' ')[1];

  public override string ToString() => "\{Name}(\{Age}";

}

Furthermore, expression-bodied properties can also be used on index members, returning an item from an internal collection, for example.

## Dictionary Initializer

Dictionary type collections are a great means for defining a name value pair. Unfortunately, the syntax for initialization is somewhat suboptimal:

{ {"First", "Value1"}, {"Second", "Value2"}, {"Third", "Value3"} }

To improve this, C# 6.0 includes a new dictionary assignment type syntax:

Dictionary<string, ConsoleColor> colorMap =

new Dictionary<string, ConsoleColor>

{

  ["Error"] =               ConsoleColor.Red,

  ["Information"] =        ConsoleColor.Yellow,

  ["Verbose"] =            ConsoleColor.White

};

To improve the syntax, the language team introduced the assignment operator as a means of associating a pair of items that make a lookup (name) value pair or map. The lookup is whatever the index value (and data type) the dictionary is declared to be.

## Exception Improvements

Not to be outdone, exceptions also had a couple of minor language tweaks in C# 6.0. First, it’s now possible to use await clauses within both catch and finally blocks, as shown in **Figure 11**.

**Figure 11 Using Await Within Catch and Finally Blocks**

public static async Task<int> EncryptFilesAsync(string directoryPath, string searchPattern = "\*")

{

  ConsoleColor color = Console.ForegroundColor;

  try

  {

  // ...

  }

  catch (System.ComponentModel.Win32Exception exception)

    if (exception.NativeErrorCode == 0x00042)

  {

    // ...

  }

  catch (AggregateException exception)

  {

    await LogExceptionsAsync(exception.InnerExceptions);

  }

  finally

  {

    Console.ForegroundColor = color;

    await RemoveTemporaryFilesAsync();

  }

}

Since the introduction of await in C# 5.0, support for await in catch and finally blocks turned out to be far more sought after than originally expected. For example, the pattern of invoking asynchronous methods from within a catch or finally block is fairly common, especially when it comes to cleaning up or logging during those times. Now, in C# 6.0, it’s finally possible.

The second exception-related feature (which has been available in Visual Basic since 1.0) is support for exception filters such that on top of filtering by a particular exception type, it’s now possible to specify an if clause to further restrict whether the exception will be caught by the catch block or not. (On occasion this feature has also been leveraged for side effects such as logging exceptions as an exception “flies by” without actually performing any exception processing.)   One caution to note about this feature is that, if there’s any chance your application might be localized, avoid catch conditional expressions that operate via exception messages because they’ll no longer work without changes following localization.

## Wrapping Up

One final point to mention about all the C# 6.0 features is that although they obviously require the C# 6.0 compiler, included with Visual Studio 2015 or later, they do not require an updated version of the Microsoft .NET Framework. Therefore, you can use C# 6.0 features even if you’re compiling against the .NET Framework 4, for example. The reason this is possible is that all features are implemented in the compiler, and don’t have any .NET Framework dependencies.

With that whirlwind tour, I wrap up my look at C# 6.0. The only two remaining features not discussed are support for defining a custom Add extension method to help with collection initializers, and some minor but improved overload resolution. In summary, C# 6.0 doesn’t radically change your code, at least not in the way that generics or LINQ did. What it does do, however, is make the right coding patterns simpler. The null-conditional operator of a delegate is probably the best example of this, but so, too, are many of the other features including string interpolation, the nameof operator and auto-property improvements (especially for read-only properties).

**What’s New in C# 6**

The C# language itself has changed little in version 6, the main importance of the release being the introduction of the Roslyn .NET Compiler Platform. However the New features and improvements that have been made to C# are welcome because they are aimed at aiding productivity. Paulo Morgado explains what they are, and how to use them.

[Paulo Morgado](https://www.simple-talk.com/author/paulo-morgado/) [Paulo orgado](https://www.simple-talk.com/author/paulo-morgado/)

Version 6 of the .NET C# programming language was introduced when Visual Studio 2015 was recently released. What were the most significant improvements in the language?

The main focus of the release was the new compiler platform (known as ‘Roslyn’): There weren’t many obvious improvements or additions to the language as such, but the changes that I’ll mention in this article will definitely be appreciated by anyone who uses the C# programming language to develop applications.

Improvements in Auto-Properties

C# has ‘auto-implemented properties’ (or, more concisely, ‘auto-properties’) that make property-declaration for simple properties more concise. Behind the scenes, a private, anonymous backing field is created that can be read or updated only via the property’s get and set accessors. They are declared with the get and set keyword followed by a semicolon.

Initializers for Auto-Properties

It is now possible to declare the auto-properties just as you would a field:

|  |  |
| --- | --- |
|  | public class Person  {      public string First { get; set; } = "Jane";      public string Last { get; set; } = "Doe";  } |

With this syntax, the initializer directly initializes the field that supports the property without actually using the set accessor of the property.

The property initializers are executed in the order they are declared, just as and along with field initializers. After all, they are, in fact, field initializers.

As with the field initializers, property initializers cannot refer to this because like field initializers, they run before the object is properly initialized.

The implementation of this new functionality is actually syntactic sugar, in that it generates code that is compatible with previous versions of the .NET platform. In fact, the code I’ve used to illustrate the new syntax is translated by the compiler to the following C# code 1:

|  |  |
| --- | --- |
|  | public class Person  {      [CompilerGenerated]      [DebuggerBrowsable(DebuggerBrowsableState.Never)]      public string <First>k\_\_BackingField = "Jane";      [CompilerGenerated]      [DebuggerBrowsable(DebuggerBrowsableState.Never)]      public string <Last>k\_\_BackingField = "Doe";        public string First      {          [CompilerGenerated]          get { return <First>k\_\_BackingField }          [CompilerGenerated]          set { <First>k\_\_BackingField = value; }      }      public string Last      {          [CompilerGenerated]          get { return <Last>k\_\_BackingField }          [CompilerGenerated]          set { <Last>k\_\_BackingField = value; }      }  } |

Note that the <First>k\_\_BackingField and <Last>k\_\_BackingField have names that are not valid C# names. This is done to ensure that there is no chance of collision between a name that is given by the programmer and a name given by the compiler.

Read-Only Auto-Properties

If you only declare a get accessor, then the property is immutable everywhere except in its initializer and the constructor, so that it becomes read-only:

|  |  |
| --- | --- |
|  | public class Person  {      public string First { get; } = "Jane";      public string Last { get; } = "Doe";  } |

In this case, the field is declared generated as readonly (although this has importance only for the purpose of reflection).

As in the previous case, the generated code will be:

|  |  |
| --- | --- |
|  | public class Person  {      [CompilerGenerated]      [DebuggerBrowsable(DebuggerBrowsableState.Never)]      private readonly string <First>k\_\_BackingField = "Jane";      [CompilerGenerated]      [DebuggerBrowsable(DebuggerBrowsableState.Never)]      private readonly string <Last>k\_\_BackingField = "Doe";        public string First      {          [CompilerGenerated]          get { return <First>k\_\_BackingField; }      }      public string Last      {          [CompilerGenerated]          get { return <Last>k\_\_BackingField; }      }  } |

As with the read-only fields, you can also initialize a read-only auto-property in the constructor.

|  |  |
| --- | --- |
|  | public class Person  {      // ...        public Person(string first, string last)      {          First = first;          Last = last;      }  } |

And, once again, the equivalent compiler generates code C# 1:

|  |  |
| --- | --- |
|  | public class Person  {      // ...        public Person(string first, string last)      {          <First>k\_\_BackingField = first;          <Last>k\_\_BackingField = last;      }  } |

Expression Bodied Function Members

We can now use the same convenient fat arrow expression syntax (=>) to define lambda expressions to define function bodies.

Expression Bodied Method-Like Members

We use the lambda fat arrow (=>) to define the body of methods as well as user-defined operators, type conversions and indexers. The expression to the right of the lambda arrow represents the body of the method.

|  |  |
| --- | --- |
|  | 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 have only a return statement. The above examples are translated by the compiler to:

|  |  |
| --- | --- |
|  | public Point Move(int dx, int dy)  {      return new Point(x + dx, y + dy);  }  public static Complex operator +(Complex a, Complex b)  {      return a.Add(b);  }  public static implicit operator string (Person p)  {      return p.First + " " + p.Last;  } |

For methods whose return type is void (or Task for asynchronous methods), the lambda arrow (=>) syntax still applies but the subsequent expression must be a statement (this is similar to what already happens with lambdas):

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

…will be translated into…

|  |  |
| --- | --- |
|  | public void Print()  {      Console.WriteLine(First + " " + Last);  } |

Expression Bodied Property-Like Function Members

Expression bodies can also be used to define the body of properties and an indexers:

|  |  |
| --- | --- |
|  | public string Name => First + " " + Last;  public Person this[long id] => store.LookupPerson(id); |

Note the absence of the get keyword, which instead is implied by the syntax of the expression.

The previous examples are translated by the compiler to:

|  |  |
| --- | --- |
|  | public string Name  {      get      {          return First + " " + Last;      }  }  public Person this[long id]  {      get      {          return store.LookupPerson(id);      }  } |

The ‘using static’ Directive

We can now import the static or enumerated members of a class into our namespace. We can then use the static members of a class directly without qualifying them with their namespace or type name!

|  |  |
| --- | --- |
|  | 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);      }  } |

The previous code is translated by the compiler to:

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

This feature is great when you have a set of related functions with a particular domain that is often used, such as System.Math. We can reference the methods of the Math class as though they were members of the class invoking them. It also allows you to specify the names of the members of an enumerated class as members of System.DayOfWeek in the example above.

Extension Methods

Extension methods are invoked like regular static methods, (See: <https://msdn.microsoft.com/en-gb/magazine/dn879355.aspx>) but they are called as if they were instance methods on the extended type. Instead of bringing these methods to the current scope, the static import functionality makes these methods available as extension methods without the need to import all extension methods in a namespace like before:

|  |  |
| --- | --- |
|  | 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 means that it will now be a breaking change to change a normal static method into an extension method, which was not the case before. Extension methods are usually only invoked as static methods when there’s an ambiguity. And, in those cases, it seems legit to require full qualification anyway.

Null-Conditional Operator

It is often necessary to scatter null checks around code. Operators with null checks allow access to members and elements only when the receiver is not null, otherwise returning a null result:

|  |  |
| --- | --- |
|  | int? length = people?.Length; // null if people is null  Person first = people?[0];    // null if people is null |

The above code is translated into:

|  |  |
| --- | --- |
|  | int? nullable = (people != null) ? new int?(people.Length) : null;  Person person = (people != null) ? people[0] : null; |

The null-conditional operators can be very convenient when used with the coalesce operator (??):

|  |  |
| --- | --- |
|  | int length = people?.Length ?? 0; // 0 se people é null |

The null-conditional operators have a short-circuit behaviour. In the chain of access to members, elements or invocations immediately adjacent will only run if the original recipient is not null:

|  |  |
| --- | --- |
|  | int? first = people?[0].Orders.Count(); |

This example is, in essence, equivalent to:

|  |  |
| --- | --- |
|  | int? first = (people != null) ? people[0].Orders.Count() : (int?)null; |

…except that people is evaluated only once. None of the access to members or elements and invocations that follow the ? operator will run if the value of people is null.

Nothing prevents null-conditional operators from being chained, if null verification is required more than once in the chain:

|  |  |
| --- | --- |
|  | int? first = people?[0]?.Orders.Count(); |

In an invocation, a list of arguments in parentheses cannot be immediately preceded by the ‘?‘ operator – It would lead to too many ambiguities. Therefore, the invoking of a delegate that one might expect if it is not null is not allowed. However, the delegate can always be invoked via its Invoke method:

|  |  |
| --- | --- |
|  | if (predicate?.Invoke(e) ?? false) { ... } |

A common use of this feature is the event trigger:

|  |  |
| --- | --- |
|  | PropertyChanged?.Invoke(this, args); |

…which is translated to…

|  |  |
| --- | --- |
|  | var handler = PropertyChanged;  if (handler != null)  {      handler.Invoke(this, args);  } |

This is a thread-safe way to see if the event has subscribers, because it only evaluates the left side of the invocation once, and keeps its value in a temporary variable. Without this feature, you used to have to write to this pattern yourself.

String Interpolation

The String.Format method, and other similar methods, is very versatile and useful, but its use is a bit awkward and error-prone due to numerical markers ({0}) that must match the position of the arguments supplied separately:

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

The interpolation syntax allows strings to directly replace the literal string indexes by “holes” with the expressions that correspond to the values:

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

As with the String.format method, it is possible to specify shapes and alignments:

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

The content of the holes can be any expression, including strings:

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

Note that the conditional expression is in brackets, so that: “s” is not confused with the format specifier.

Formattable Strings

When not otherwise specified, a formatting provider uses the current culture of the current thread when invoking the String.Formatmethod, but this is not always what is wanted. That is why, as happens with lambda expressions, the compiler translates the interpolated string differently depending on the type of receptor expression.

If the receiver of expression is the IFormattable type…

|  |  |
| --- | --- |
|  | IFormattable christmas = $"{new DateTime(2015, 12, 25):f}"; |

…the compiler generates the following code:

|  |  |
| --- | --- |
|  | IFormattable christmas = FormattableStringFactory.Create("{0:f}", new DateTime(2015, 12, 25)); |

This can be used as follows:

|  |  |
| --- | --- |
|  | var christamasText = christmas.ToString(null, new CultureInfo("pt-PT")); |

FormattableString

The concrete type returned by FormattableStringFactory.Create is derived from:

|  |  |
| --- | --- |
|  | namespace System  {      public abstract class FormattableString : IFormattable      {          protected FormattableString();          public abstract int ArgumentCount { get; }          public abstract string Format { get; }          public static string Invariant(FormattableString formattable);          public abstract object GetArgument(int index);          public abstract object[] GetArguments();          public override string ToString();          public abstract string ToString(IFormatProvider formatProvider);      }  } |

This provides access not only the format but also to the format string arguments.

Backward Compatibility

The features introduced by the C# 6 are compatible with the previous .NET platforms. However, this particular feature requires System.Runtime.CompilerServices.FormattableStringFactory and System.FormattableString types that have been introduced only in version 4.6 of the platform. The good news is that the compiler is not tied to the location of these types in a particular assembly and, if we are to use this functionality in an earlier version of the platform, we just need to add the implementation of these types.

‘nameof’ Expressions

Occasionally it is necessary to provide a string with the name of some program elements.

* This can happen when throwing a System.ArgumentNullException;
* Firing a PropertyChanged event;
* And many other occasions.

We can use string literals for this, but they are error prone and are not changed when a name is changed by a code refactoring.

The nameof expressions are a sort of literal of type string in which the compiler validates the existence of something with that name. As it becomes a reference to the artefact, Visual Studio knows what it refers to and code navigation and refactoring work.

In essence, code such as the following:

|  |  |
| --- | --- |
|  | if (x == null) throw new ArgumentNullException(nameof(x));  var s = nameof(person.Address.ZipCode); |

is converted to:

|  |  |
| --- | --- |
|  | if (x == null) throw new ArgumentNullException("x");  var s = "ZipCode"; |

Source Code vs. Metadata

The names used by the compiler are the source names and not the metadata names of the artifacts, and so the following code…

|  |  |
| --- | --- |
|  | using S = System.String;  class C  {      void M<T>(S s)      {          var s1 = nameof(T);          var s2 = nameof(S);      }  } |

…is converted to…

|  |  |
| --- | --- |
|  | using S = System.String;  class C  {      void M<T>(S s)      {          var s1 = "T";          var s2 = "S";      }  } |

Primitive Types

It is not allowed to use primitive types (int, long, char, bool, string, etc.) in nameof expressions because they are not expressions and the argument of nameof is an expression.

Add Extension Methods in Collection Initializers

When collection initializers were introduced in C#, the Add method calls could not be an extension method invocation. Visual Basic allowed its use but it seemed to have been forgotten for C#.

In this version the fault has been corrected and it is now possible to use Add extension methods in collection initializers.

It is a small but useful feature and, in terms of the implementation of the compiler, this was just the removal of the check of the condition that prevented it.

Index Initializers

Object and collection initializers are useful to initialize declaratively the fields and properties of objects or, in the case of collections, an initial set of elements.

Initializing dictionaries, on the other hand, were not so elegant, requiring the existence of an Add method that received as an argument the key and the value corresponding to that key. If a dictionary implementation in particular did not have an Add method with the aforementioned characteristics, it would not be possible to use an initializer.

From now on, it becomes possible to use initializers that use indexes:

|  |  |
| --- | --- |
|  | var numbers = new Dictionary<int, string>  {      [7] = "sete",      [9] = "nove",      [13] = "treze"  }; |

which will be translated into:

|  |  |
| --- | --- |
|  | var dictionary = new Dictionary<int, string>();  dictionary[7] = "sete";  dictionary[9] = "nove";  dictionary[13] = "treze";  var numbers = dictionary; |

Exception Filters

Exception filters a CLR feature already provided by Visual Basic and F# and will now also be available in C#:

|  |  |
| --- | --- |
|  | try  {      ...  }  catch (Exception ex) when (SomeFilter(ex))  {      ...  } |

If the evaluation of the expression in parentheses after the when keyword evaluates to true, the exception is caught. Otherwise, the catch block is ignored.

This allows them to be laid over a catch block for the same type of exception:

|  |  |
| --- | --- |
|  | try  {      //...  }  catch (SqlException ex) when (ex.Number == 2)  {      // ...  }  catch (SqlException ex)  {      // ...  } |

In the example above, the first catch block is only executed if an exception of type SqlException in the value of the Number property is 2. Otherwise you run the next block.

It is considered acceptable and commonplace “abuse” of exceptions filters with side effects such as logging.

WARNING: Exception filters are executed in the context of the throw and not in the context of the catch because the stack hasn’t been unwound yet.

‘await’ in ‘catch’ and ‘finally’ blocks

C# 5 was not allowed to use the await keyword in catch and finally blocks because, at the time of implementation of the async-functionality await, the team thought that this would not be possible to implement. But later, they found out that, after all, it was not impossible, although it can lead to some [interesting semantics](https://clivetong.wordpress.com/2015/07/22/c-await-inside-catch-and-finally-leads-to-some-interesting-semantics/).

It becomes possible to write code like this:

|  |  |
| --- | --- |
|  | Resource res = null;  try  {      res = await Resource.OpenAsync();    }  catch (ResourceException e)  {      await Resource.LogAsync(res, e);  }  finally  {      if (res != null) await res.CloseAsync();  } |

Improvements in Overload Resolution Methods

They were introduced some improvements in overload method resolution (See: <http://bc-programming.com/blogs/2015/06/c-6-features-improved-overload-resolution/>) in order to make it more intuitive to determine how the compiler decides which method overloading to use.

Where it is more noticeable is the choice of method overloading when receiving nullable value types, or when moving a group of methods (rather than a lambda) methods for overload receiving delegates.