What's New in C# 6

The 6.0 release of C# contained many features that improve productivity for developers. Features in this release include:

- Read-only Auto-properties:
  - You can create read-only auto-properties that can be set only in constructors.
- Auto-Property Initializers:
  - You can write initialization expressions to set the initial value of an auto-property.
- Expression-bodied function members:
  - You can author one-line methods using lambda expressions.
- using static:
  - You can import all the methods of a single class into the current namespace.
- Null - conditional operators:
  - You can concisely and safely access members of an object while still checking for null with the null conditional operator.
- String Interpolation:
  - You can write string formatting expressions using inline expressions instead of positional arguments.
- Exception filters:
  - You can catch expressions based on properties of the exception or other program state.
- nameof Expressions:
  - You can let the compiler generate string representations of symbols.
- await in catch and finally blocks:
  - You can use `await` expressions in locations that previously disallowed them.
- index initializers:
  - You can author initialization expressions for associative containers as well as sequence containers.
- Extension methods for collection initializers:
  - Collection initializers can rely on accessible extension methods, in addition to member methods.
- Improved overload resolution:
  - Some constructs that previously generated ambiguous method calls now resolve correctly.
- `deterministic` compiler option:
  - The deterministic compiler option ensures that subsequent compilations of the same source generate the same binary output.

The overall effect of these features is that you write more concise code that is also more readable. The syntax contains less ceremony for many common practices. It's easier to see the design intent with less ceremony. Learn these features well, and you'll be more productive, write more readable code, and concentrate more on your core features than on the constructs of the language.

The remainder of this topic provides details on each of these features.

Auto-Property enhancements

The syntax for automatically implemented properties (usually referred to as 'auto-properties') made it very easy to create properties that had simple get and set accessors:

```
public string FirstName { get; set; }
public string LastName { get; set; }
```

However, this simple syntax limited the kinds of designs you could support using auto-properties. C# 6 improves the auto-properties capabilities so that you can use them in more scenarios. You won't need to fall back on the more verbose syntax of declaring and manipulating the backing field by hand so often.

The new syntax addresses scenarios for read-only properties, and for initializing the variable storage behind an auto-property.

Read-only auto-properties

Read-only auto-properties provide a more concise syntax to create immutable types. The closest you could get to immutable types in earlier versions of C# was to declare private setters:

```
public string FirstName { get; private set; }
public string LastName { get; private set; }
```

Using this syntax, the compiler doesn't ensure that the type really is immutable. It only enforces that the `FirstName` and `LastName` properties are not modified from any code outside the class.

Read-only auto-properties enable true read-only behavior. You declare the auto-property with only a get accessor:

```
public string FirstName { get; }
public string LastName { get;  }
```

The `FirstName` and `LastName` properties can be set only in the body of a constructor:

```
public Student(string firstName, string lastName)
{
    if (IsNullOrWhiteSpace(lastName))
        throw new ArgumentException(message: "Cannot be blank", paramName:
nameof(lastName));
    FirstName = firstName;
    LastName = lastName;
}
```

Trying to set `LastName` in another method generates a `CS0200` compilation error:

```
public class Student
{
    public string LastName { get;  }

    public void ChangeName(string newLastName)
    {
        // Generates CS0200: Property or indexer cannot be assigned to -- it
is read only
        LastName = newLastName;
    }
}
```

This feature enables true language support for creating immutable types and using the more concise and convenient auto-property syntax.

Auto-Property Initializers

Auto-Property Initializers let you declare the initial value for an auto-property as part of the property declaration. In earlier versions, these properties would need to have setters and you would need to use that setter to initialize the data storage used by the backing field. Consider this class for a student that contains the name and a list of the student's grades:

```
public Student(string firstName, string lastName)
{
    FirstName = firstName;
    LastName = lastName;
}
```

As this class grows, you may include other constructors. Each constructor needs to initialize this field, or you'll introduce errors.

C# 6 enables you to assign an initial value for the storage used by an auto-property in the auto-property declaration:

```
public ICollection<double> Grades { get; } = new List<double>();
```

The `Grades` member is initialized where it is declared. That makes it easier to perform the initialization exactly once. The initialization is part of the property declaration, making it easier to equate the storage allocation with public interface for `Student` objects.

Property Initializers can be used with read/write properties as well as read-only properties, as shown here.

```
public Standing YearInSchool { get; set; } = Standing.Freshman;
```

Expression-bodied function members

The body of a lot of members that we write consist of only one statement that can be represented as an expression. You can reduce that syntax by writing an expression-bodied member instead. It works for methods and read-only properties. For example, an override of `ToString()` is often a great candidate:

```
public override string ToString() => $"{LastName}, {FirstName}";
```

You can also use expression-bodied members in read-only properties as well:

```
public string FullName => $"{FirstName} {LastName}";
```

using static

The *using static* enhancement enables you to import the static methods of a single class. Previously, the `using` statement imported all types in a namespace.

Often we use a class' static methods throughout our code. Repeatedly typing the class name can obscure the meaning of your code. A common example is when you write classes that perform many numeric calculations. Your code will be littered with Sin, Sqrt and other calls to different methods in the Math class. The new `using static` syntax can make these classes much cleaner to read. You specify the class you're using:

```
using static System.Math;
```

And now, you can use any static method in the Math class without qualifying the Math class. The Math class is a great use case for this feature because it does not contain any instance methods. You can also use `using static` to import a class' static methods for a class that has both static and instance methods. One of the most useful examples is String:

```
using static System.String;
```

Note

You must use the fully qualified class name, `System.String` in a static using statement. You cannot use the `string` keyword instead.

You can now call static methods defined in the String class without qualifying those methods as members of that class:

```
if (IsNullOrWhiteSpace(lastName))
    throw new ArgumentException(message: "Cannot be blank", paramName:
nameof(lastName));
```

The `static using` feature and extension methods interact in interesting ways, and the language design included some rules that specifically address those interactions. The goal is to minimize any chances of breaking changes in existing codebases, including yours.

Extension methods are only in scope when called using the extension method invocation syntax, not when called as a static method. You'll often see this in LINQ queries. You can import the LINQ pattern by importing Enumerable.

```
using static System.Linq.Enumerable;
```

This imports all the methods in the Enumerable class. However, the extension methods are only in scope when called as extension methods. They are not in scope if they are called using the static method syntax:

```
public bool MakesDeansList()
{
    return Grades.All(g => g > 3.5) && Grades.Any();
    // Code below generates CS0103:
    // The name 'All' does not exist in the current context.
    //return All(Grades, g => g > 3.5) && Grades.Any();
}
```

This decision is because extension methods are typically called using extension method invocation expressions. In the rare case where they are called using the static method call syntax it is to resolve ambiguity. Requiring the class name as part of the invocation seems wise.

There's one last feature of `static using`. The `static using` directive also imports any nested types. That enables you to reference any nested types without qualification.

Null-conditional operators

Null values complicate code. You need to check every access of variables to ensure you are not dereferencing `null`. The *null conditional operator* makes those checks much easier and fluid.

Simply replace the member access `.` with `?.`:

```
var first = person?.FirstName;
```

In the preceding example, the variable `first` is assigned `null` if the person object is `null`. Otherwise, it gets assigned the value of the `FirstName` property. Most importantly, the `?.` means that this line of code does not generate a `NullReferenceException` when the `person` variable is `null`. Instead, it short-circuits and produces `null`.

Also, note that this expression returns a `string`, regardless of the value of `person`. In the case of short circuiting, the `null` value returned is typed to match the full expression.

You can often use this construct with the *null coalescing* operator to assign default values when one of the properties are `null`:

```
first = person?.FirstName ?? "Unspecified";
```

The right hand side operand of the `?.` operator is not limited to properties or fields. You can also use it to conditionally invoke methods. The most common use of member functions with the null conditional operator is to safely invoke delegates (or event handlers) that may be `null`. You'll do this by calling the delegate's `Invoke` method using the `?.` operator to access the member. You can see an example in the delegate patterns topic.

The rules of the `?.` operator ensure that the left-hand side of the operator is evaluated only once. This is important and enables many idioms, including the example using event handlers. Let's start with the event handler usage. In previous versions of C#, you were encouraged to write code like this:

```
var handler = this.SomethingHappened;
if (handler != null)
    handler(this, eventArgs);
```

This was preferred over a simpler syntax:

```
// Not recommended
if (this.SomethingHappened != null)
    this.SomethingHappened(this, eventArgs);
```

Important

The preceding example introduces a race condition. The `SomethingHappened` event may have subscribers when checked against `null`, and those subscribers may have been removed before the event is raised. That would cause a **NullReferenceException** to be thrown.

In this second version, the `SomethingHappened` event handler might be non-null when tested, but if other code removes a handler, it could still be null when the event handler was called.

The compiler generates code for the `?.` operator that ensures the left side (`this.SomethingHappened`) of the `?.` expression is evaluated once, and the result is cached:

```
// preferred in C# 6:
this.SomethingHappened?.Invoke(this, eventArgs);
```

Ensuring that the left side is evaluated only once also enables you to use any expression, including method calls, on the left side of the `?.` Even if these have side-effects, they are evaluated once, so the side effects occur only once. You can see an example in our content on events.

String Interpolation

C# 6 contains new syntax for composing strings from a format string and expressions that are evaluated to produce other string values.

Traditionally, you needed to use positional parameters in a method like `string.Format`:

```
public string FullName
{
    get
    {
```

```
        return string.Format("{0} {1}", FirstName, LastName);
    }
}
```

With C# 6, the new string interpolation feature enables you to embed the expressions in the format string. Simply preface the string with `$`:

```
public string FullName => $"{FirstName} {LastName}";
```

This initial example uses property expressions for the substituted expressions. You can expand on this syntax to use any expression. For example, you could compute a student's grade point average as part of the interpolation:

```
public string GetFormattedGradePoint() =>
    $"Name: {LastName}, {FirstName}. G.P.A: {Grades.Average()}";
```

Running the preceding example, you would find that the output for `Grades.Average()` might have more decimal places than you would like. The string interpolation syntax supports all the format strings available using earlier formatting methods. You add the format strings inside the braces. Add a `:` following the expression to format:

```
public string GetGradePointPercentage() =>
    $"Name: {LastName}, {FirstName}. G.P.A: {Grades.Average():F2}";
```

The preceding line of code formats the value for `Grades.Average()` as a floating-point number with two decimal places.

The `:` is always interpreted as the separator between the expression being formatted and the format string. This can introduce problems when your expression uses a `:` in another way, such as a conditional operator:

```
public string GetGradePointPercentages() =>
```

```
    $"Name: {LastName}, {FirstName}. G.P.A: {Grades.Any() ? Grades.Average()
: double.NaN:F2}";
```

In the preceding example, the `:` is parsed as the beginning of the format string, not part of the conditional operator. In all cases where this happens, you can surround the expression with parentheses to force the compiler to interpret the expression as you intend:

```
public string GetGradePointPercentages() =>
    $"Name: {LastName}, {FirstName}. G.P.A: {(Grades.Any() ? Grades.Average()
: double.NaN):F2}";
```

There aren't any limitations on the expressions you can place between the braces. You can execute a complex LINQ query inside an interpolated string to perform computations and display the result:

```
public string GetAllGrades() =>
    $@"All Grades: {Grades.OrderByDescending(g => g)
    .Select(s => s.ToString("F2")).Aggregate((partial, element) =>
$"{partial}, {element}")}";
```

You can see from this sample that you can even nest a string interpolation expression inside another string interpolation expression. This example is very likely more complex than you would want in production code. Rather, it is illustrative of the breadth of the feature. Any C# expression can be placed between the curly braces of an interpolated string.

String interpolation and specific cultures

All the examples shown in the preceding section format the strings using the current culture and language on the machine where the code executes. Often you may need to format the string produced using a specific culture. To do that use the fact that the object produced by a string interpolation can be implicitly converted to FormattableString.

The FormattableString instance contains the format string, and the results of evaluating the expressions before converting them to strings. You can use public methods of FormattableString to specify the culture when formatting a string. For example, the

following example produces a string using German culture. (It uses the ',' character for the decimal separator, and the '.' character as the thousands separator.)

```
FormattableString str = $"Average grade is {s.Grades.Average()}";
var gradeStr = str.ToString(new System.Globalization.CultureInfo("de-DE"));
```

For more information, see the String interpolation topic.

Exception Filters

Another new feature in C# 6 is *exception filters*. Exception Filters are clauses that determine when a given catch clause should be applied. If the expression used for an exception filter evaluates to `true`, the catch clause performs its normal processing on an exception. If the expression evaluates to `false`, then the `catch` clause is skipped.

One use is to examine information about an exception to determine if a `catch` clause can process the exception:

```
public static async Task<string> MakeRequest()
{
    WebRequestHandler webRequestHandler = new WebRequestHandler();
    webRequestHandler.AllowAutoRedirect = false;
    using (HttpClient client = new HttpClient(webRequestHandler))
    {
        var stringTask =
client.GetStringAsync("https://docs.microsoft.com/en-us/dotnet/about/");
        try
        {
            var responseText = await stringTask;
            return responseText;
        }
        catch (System.Net.Http.HttpRequestException e) when
(e.Message.Contains("301"))
        {
            return "Site Moved";
        }
    }
}
```

The code generated by exception filters provides better information about an exception that is thrown and not processed. Before exception filters were added to the language, you would need to create code like the following:

```
public static async Task<string> MakeRequest()
{
    var client = new System.Net.Http.HttpClient();
    var streamTask = client.GetStringAsync("https://localHost:10000");
    try {
        var responseText = await streamTask;
        return responseText;
    } catch (System.Net.Http.HttpRequestException e)
    {
        if (e.Message.Contains("301"))
            return "Site Moved";
        else
            throw;
    }
}
```

The point where the exception is thrown changes between these two examples. In the previous code, where a `throw` clause is used, any stack trace analysis or examination of crash dumps will show that the exception was thrown from the `throw` statement in your catch clause. The actual exception object will contain the original call stack, but all other information about any variables in the call stack between this throw point and the location of the original throw point has been lost.

Contrast that with how the code using an exception filter is processed: the exception filter expression evaluates to `false`. Therefore, execution never enters the `catch` clause. Because the `catch` clause does not execute, no stack unwinding takes place. That means the original throw location is preserved for any debugging activities that would take place later.

Whenever you need to evaluate fields or properties of an exception, instead of relying solely on the exception type, use an exception filter to preserve more debugging information.

Another recommended pattern with exception filters is to use them for logging routines. This usage also leverages the manner in which the exception throw point is preserved when an exception filter evaluates to `false`.

A logging method would be a method whose argument is the exception that unconditionally returns `false`:

```
public static bool LogException(this Exception e)
{
    Console.Error.WriteLine($"Exceptions happen: {e}");
    return false;
}
```

Whenever you want to log an exception, you can add a catch clause, and use this method as the exception filter:

```
public void MethodThatFailsSometimes()
{
    try {
        PerformFailingOperation();
    } catch (Exception e) when (e.LogException())
    {
        // This is never reached!
    }
}
```

The exceptions are never caught, because the `LogException` method always returns `false`. That always false exception filter means that you can place this logging handler before any other exception handlers:

```
public void MethodThatFailsButHasRecoveryPath()
{
    try {
        PerformFailingOperation();
    } catch (Exception e) when (e.LogException())
    {
        // This is never reached!
    }
    catch (RecoverableException ex)
    {
        Console.WriteLine(ex.ToString());
        // This can still catch the more specific
        // exception because the exception filter
```

```
        // above always returns false.
        // Perform recovery here
    }
}
```

The preceding example highlights a very important facet of exception filters. The exception filters enable scenarios where a more general exception catch clause may appear before a more specific one. It's also possible to have the same exception type appear in multiple catch clauses:

```
public static async Task<string> MakeRequestWithNotModifiedSupport()
{
    var client = new System.Net.Http.HttpClient();
    var streamTask = client.GetStringAsync("https://localHost:10000");
    try {
        var responseText = await streamTask;
        return responseText;
    } catch (System.Net.Http.HttpRequestException e) when
(e.Message.Contains("301"))
    {
        return "Site Moved";
    } catch (System.Net.Http.HttpRequestException e) when
(e.Message.Contains("304"))
    {
        return "Use the Cache";
    }
}
```

Another recommended pattern helps prevent catch clauses from processing exceptions when a debugger is attached. This technique enables you to run an application with the debugger, and stop execution when an exception is thrown.

In your code, add an exception filter so that any recovery code executes only when a debugger is not attached:

```
public void MethodThatFailsWhenDebuggerIsNotAttached()
{
    try {
        PerformFailingOperation();
    } catch (Exception e) when (e.LogException())
```

```
    {
        // This is never reached!
    }
    catch (RecoverableException ex) when
(!System.Diagnostics.Debugger.IsAttached)
    {
        Console.WriteLine(ex.ToString());
        // Only catch exceptions when a debugger is not attached.
        // Otherwise, this should stop in the debugger.
    }
}
```

After adding this in code, you set your debugger to break on all unhandled exceptions. Run the program under the debugger, and the debugger breaks whenever `PerformFailingOperation()` throws a `RecoverableException`. The debugger breaks your program, because the catch clause won't be executed due to the false-returning exception filter.

`nameof` Expressions

The `nameof` expression evaluates to the name of a symbol. It's a great way to get tools working whenever you need the name of a variable, a property, or a member field.

One of the most common uses for `nameof` is to provide the name of a symbol that caused an exception:

```
if (IsNullOrWhiteSpace(lastName))
    throw new ArgumentException(message: "Cannot be blank", paramName:
nameof(lastName));
```

Another use is with XAML based applications that implement the `INotifyPropertyChanged` interface:

```
public string LastName
{
    get { return lastName; }
    set
    {
        if (value != lastName)
```

```
        {
            lastName = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(LastName)));
        }
    }
}
private string lastName;
```

The advantage of using the `nameof` operator over a constant string is that tools can understand the symbol. If you use refactoring tools to rename the symbol, it will rename it in the `nameof` expression. Constant strings don't have that advantage. Try it yourself in your favorite editor: rename a variable, and any `nameof` expressions will update as well.

The `nameof` expression produces the unqualified name of its argument (`LastName` in the previous examples) even if you use the fully qualified name for the argument:

```
public string FirstName
{
    get { return firstName; }
    set
    {
        if (value != firstName)
        {
            firstName = value;
            PropertyChanged?.Invoke(this,
                new
PropertyChangedEventArgs(nameof(UXComponents.ViewModel.FirstName)));
        }
    }
}
private string firstName;
```

This `nameof` expression produces `FirstName`, not `UXComponents.ViewModel.FirstName`.

Await in Catch and Finally blocks

C# 5 had several limitations around where you could place `await` expressions. One of those has been removed in C# 6. You can now use `await` in `catch` or `finally` expressions.

The addition of await expressions in catch and finally blocks may appear to complicate how those are processed. Let's add an example to discuss how this appears. In any async method, you can use an await expression in a finally clause.

With C# 6, you can also await in catch expressions. This is most often used with logging scenarios:

```
public static async Task<string> MakeRequestAndLogFailures()
{
    await logMethodEntrance();
    var client = new System.Net.Http.HttpClient();
    var streamTask = client.GetStringAsync("https://localHost:10000");
    try {
        var responseText = await streamTask;
        return responseText;
    } catch (System.Net.Http.HttpRequestException e) when
(e.Message.Contains("301"))
    {
        await logError("Recovered from redirect", e);
        return "Site Moved";
    }
    finally
    {
        await logMethodExit();
        client.Dispose();
    }
}
```

The implementation details for adding `await` support inside `catch` and `finally` clauses ensures that the behavior is consistent with the behavior for synchronous code. When code executed in a `catch` or `finally` clause throws, execution looks for a suitable `catch` clause in the next surrounding block. If there was a current exception, that exception is lost. The same happens with awaited expressions in `catch` and `finally` clauses: a suitable `catch` is searched for, and the current exception, if any, is lost.

Note

This behavior is the reason it's recommended to write `catch` and `finally` clauses carefully, to avoid introducing new exceptions.

Index Initializers

Index Initializers is one of two features that make collection initializers more consistent with index usage. In earlier releases of C#, you could use *collection initializers* only with sequence style collections, including Dictionary<TKey,TValue> by adding braces around key and value pairs:

```
private List<string> messages = new List<string>
{
    "Page not Found",
    "Page moved, but left a forwarding address.",
    "The web server can't come out to play today."
};
```

Now, you can use them with Dictionary<TKey,TValue> collections and similar types. The new syntax supports assignment using an index into the collection:

```
private Dictionary<int, string> webErrors = new Dictionary<int, string>
{
    [404] = "Page not Found",
    [302] = "Page moved, but left a forwarding address.",
    [500] = "The web server can't come out to play today."
};
```

This feature means that associative containers can be initialized using syntax similar to what's been in place for sequence containers for several versions.

Extension `Add` methods in collection initializers

Another feature that makes collection initialization easier is the ability to use an *extension method* for the `Add` method. This feature was added for parity with Visual Basic.

The feature is most useful when you have a custom collection class that has a method with a different name to semantically add new items.

For example, consider a collection of students like this:

```
public class Enrollment : IEnumerable<Student>
{
    private List<Student> allStudents = new List<Student>();

    public void Enroll(Student s)
    {
        allStudents.Add(s);
    }

    public IEnumerator<Student> GetEnumerator()
    {
        return ((IEnumerable<Student>)allStudents).GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return ((IEnumerable<Student>)allStudents).GetEnumerator();
    }
}
```

The `Enroll` method adds a student. But it doesn't follow the `Add` pattern. In previous versions of C#, you could not use collection initializers with an `Enrollment` object:

```
var classList = new Enrollment()
{
    new Student("Lessie", "Crosby"),
    new Student("Vicki", "Petty"),
    new Student("Ofelia", "Hobbs"),
    new Student("Leah", "Kinney"),
    new Student("Alton", "Stoker"),
    new Student("Luella", "Ferrell"),
    new Student("Marcy", "Riggs"),
    new Student("Ida", "Bean"),
    new Student("Ollie", "Cottle"),
    new Student("Tommy", "Broadnax"),
    new Student("Jody", "Yates"),
    new Student("Marguerite", "Dawson"),
    new Student("Francisca", "Barnett"),
    new Student("Arlene", "Velasquez"),
    new Student("Jodi", "Green"),
```

```
    new Student("Fran", "Mosley"),
    new Student("Taylor", "Nesmith"),
    new Student("Ernesto", "Greathouse"),
    new Student("Margret", "Albert"),
    new Student("Pansy", "House"),
    new Student("Sharon", "Byrd"),
    new Student("Keith", "Roldan"),
    new Student("Martha", "Miranda"),
    new Student("Kari", "Campos"),
    new Student("Muriel", "Middleton"),
    new Student("Georgette", "Jarvis"),
    new Student("Pam", "Boyle"),
    new Student("Deena", "Travis"),
    new Student("Cary", "Totten"),
    new Student("Althea", "Goodwin")
};
```

Now you can, but only if you create an extension method that maps `Add` to `Enroll`:

```
public static class StudentExtensions
{
    public static void Add(this Enrollment e, Student s) => e.Enroll(s);
}
```

What you are doing with this feature is to map whatever method adds items to a collection to a method named `Add` by creating an extension method.

Improved overload resolution

This last feature is one you probably won't notice. There were constructs where the previous version of the C# compiler may have found some method calls involving lambda expressions ambiguous. Consider this method:

```
static Task DoThings()
{
     return Task.FromResult(0);
}
```

In earlier versions of C#, calling that method using the method group syntax would fail:

```
Task.Run(DoThings);
```

The earlier compiler could not distinguish correctly between `Task.Run(Action)` and `Task.Run(Func<Task>())`. In previous versions, you'd need to use a lambda expression as an argument:

```
Task.Run(() => DoThings());
```

The C# 6 compiler correctly determines that `Task.Run(Func<Task>())` is a better choice.

Deterministic compiler output

The `-deterministic` option instructs the compiler to produce a byte-for-byte identical output assembly for successive compilations of the same source files.

By default, every compilation produces unique output on each compilation. The compiler adds a timestamp, and a GUID generated from random numbers. You use this option if you want to compare the byte-for-byte output to ensure consistency across builds.