C# 6.0

What's New Quick Start

Jason Roberts

C# 6.0: What's New Quick Start

Jason Roberts

This book is for sale at http://leanpub.com/cs6

This version was published on 2016-05-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

Welcome to C# 6.0: What's New Quick Start

C# 6.0 adds a number of smaller language features with the design goal of allowing developers to write cleaner, more expressive code by making the intent of the programmer clearer and by reducing some of the repetitive boilerplate code that previous versions of the language may have required.

This book is a useful guide for quickly getting up to speed on the new features that have been added in C# 6.0. The code examples show what the code would have looked like in C# 5.0 and how the C# 6.0 code differs.

About The Author

With over 15 years experience, Jason Roberts is a Microsoft .NET MVP, freelance developer, writer and Pluralsight course author[1]. He is the author of multiple books including Clean C#, and C# Tips and writes at his blog DontCodeTired.com[2]. He is an open source contributor and the creator of FeatureToggle. In addition to enterprise software development, he has designed and developed both Windows Phone and Windows Store apps. He holds a Bachelor of Science degree in computing and is an amateur music producer and landscape photographer.

You can contact him on Twitter as @robertsjason[3], at his blog DontCodeTired.com[4] or check out his Pluralsight courses by Jason Roberts[5].

[1] http://bit.ly/psjasonroberts
[2] http://dontcodetired.com
[3] https://twitter.com/robertsjason
[4] http://dontcodetired.com
[5] http://bit.ly/psjasonroberts

Other Books by Jason Roberts

- Keeping Software Soft[6] (also available on Kindle).
- Clean C#[7]
- C# Tips[8]

---

[6]http://keepingsoftwaresoft.com
[7]http://cleancsharp.com/
[8]http://bit.ly/sharpbook

Using Static Type Directive

Before C# 6.0 the `using` directive allowed the "importing" of a namespace so that accessing types within that namespace can be done without having to use the fully qualified version.

In C# 6.0 the `static` modifier can be used. When `using` is followed by `static`, rather than specifying a namespace, an actual type is specified.

Once the type has been specified, its static members can be used without first having to reference the name of the static type itself.

The following code shows an example of calling static methods on the `Console` class.

C# 5 code accessing static members of the Console class

```
using System;

namespace CSharp5
{
    public class ConsoleWriter
    {
        public void WriteSomething(string message)
        {
            Console.ForegroundColor = ConsoleColor.Black;
            Console.BackgroundColor = ConsoleColor.White;
            Console.WriteLine(message);
        }
    }
}
```

In the preceding code, the static `ForegroundColor` and `BackgroundColor` properties of the `Console` class are being referenced. The values of the `ConsoleColor` enumeration are also being accessed, e.g. `ConsoleColor.Black`.

The following code shows an example of the same class using the new using static type directive to allow the static members of the `Console` class to be accessed without needing the preceding `Console` reference.

C# 6 code accessing static members of the Console class

```
using System;
using static System.Console;

namespace CSharp6
{
    public class ConsoleWriter
    {
        public void WriteSomething(string message)
        {
            ForegroundColor = ConsoleColor.White;
            BackgroundColor = ConsoleColor.Black;
            WriteLine(message);
        }
    }
}
```

Notice the `using static System.Console;` line. This can be thought of as "importing" all of the public static members from the `System.Console` class into the file. Now rather than needing to explicitly reference the `Console` class before accessing its public static members, the static members are automatically available to be referenced.

To reduce the amount of source code further, the members of the `ConsoleColor` enumeration could be imported.

C# 6 code accessing values of the ConsoleColor enum

```
using static System.ConsoleColor;
using static System.Console;

namespace CSharp6
{
    public class ConsoleWriter2
    {
        public void WriteSomething(string message)
        {
            ForegroundColor = White;
            BackgroundColor = Black;
            WriteLine(message);
        }
    }
}
```

Now instead of writing `ConsoleColor.Black`, this can be reduced to `Black`.

The `using static` feature can also be applied to custom code. Take the following class:

Custom code with a static method

```
namespace CSharp6
{
    public class Greetings
    {
        public static string GenerateGreeting()
        {
            return "Hi!";
        }
    }
}
```

Notice in the preceding code that the `Greetings` class itself is not static, however the static `GenerateGreeting` method can be imported, as the following code demonstrates:

Applying using static against custom code

```
using static CSharp6.Greetings;

namespace CSharp6
{
    class GreetingsUser
    {
        public void DoSomething()
        {
            var message = GenerateGreeting();
        }
    }
}
```

Benefits

In line with the design goals of C# 6.0 to improve expressiveness and reduce boilerplate code, `using static` can reduce the amount of code that needs to be typed and read. It can reduce some of the "clutter" and make reading the code a less mentally taxing experience.

Considerations

In smaller, more functionally cohesive classes where it's very obvious what the class does, `using static` can improve readability. However in larger, less functionally cohesive classes it has the potential to harm readability, more so if lots of `static usings` are being applied in the same file. In the case of larger classes (where refactoring to multiple highly cohesive classes is not possible) the application of static usings should be given consideration of whether or not it will help. In these cases there may still be benefits to be gained, especially where one set of static methods is used many times throughout the large class.

String Interpolation

The string interpolation feature of C# 6.0 makes it less cumbersome to concatenate strings and values. Typically the joining of strings is done by using the + operator on strings and other object types, or through the using of `string.Format()` and format strings.

The following code shows the use of `string.Format` in an override of `ToString`.

C# 5 string concatenation using string.Format

```
namespace CSharp5
{
    public class Person
    {
        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            return string.Format(
                "{0} {1} {2} is {3} years old",
                    Title, FirstName, SecondName, Age);
        }
    }
}
```

This implementation could be tested in a unit test as the following code demonstrates:

C# 5 string concatenation using string.Format

```
using Xunit;

namespace CSharp5
{
    public class PersonTests
    {
        [Fact]
        public void PersonToStringShouldBeCustomized()
        {
            var p = new Person
            {
                Title = "Mrs",
                FirstName = "Sarah",
                SecondName = "Smith",
                Age = 30
            };

            Assert.Equal("Mrs Sarah Smith is 30 years old", p.ToString());
        }
    }
}
```

Running the preceding test would pass and `ToString` would produce the result: "Mrs Sarah Smith is 30 years old".

In C# 6.0 the code could be changed and the same result produced by using string interpolation as the following code demonstrates:

C# 6 string concatenation using string interpolation

```
namespace CSharp6
{
    public class Person
    {
        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            return $"{Title} {FirstName} {SecondName} is {Age} years old";
        }
    }
}
```

Notice in the preceding code the addition of the `$` before the open quote of the string. This signals that string interpolation is to be used. Within the string, where we want the values of objects to be used, we reference the name of the variable inside braces, e.g. `{Title}`.

When using `string.Format`, the placeholders can have formats applied as the following modified Person class shows:

C# 5 string.Format formatting

```
namespace CSharp5
{
    public class Person2
    {
        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            return string.Format(
                "{0} {1} {2} is {3:#.00} years old",
                    Title, FirstName, SecondName, Age);
        }
    }
}
```

Running the previous test would result in `ToString` returning: "Mrs Sarah Smith is 30.00 years old".

Format specifiers can also be used with string interpolation. The following code shows the modified C# 6.0 version, using the same formatting for age.

C# 6 formatting with string interpolation

```
namespace CSharp6
{
    public class Person2
    {
        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            return
                $"{Title} {FirstName} {SecondName} is {Age:#.00} years old";
        }
    }
}
```

Advanced Usage

The following code shows a modified person class with a `DateTime` property for the birth date.

C# 6 Modified person with birth date

```
using System;

namespace CSharp6
{
    public class Person3
    {
        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public DateTime BirthDate { get; set; }

        public override string ToString()
        {
            return $"{Title} {FirstName} {SecondName} was born {BirthDate}";
        }
    }
}
```

If the `ToString` method should always use the invariant culture, the following test can be written that switches the culture to Germany:

C# 6 Failing test expecting invariant date output

```
using System;
using System.Globalization;
using System.Threading;
using Xunit;

namespace CSharp6
{
    public class Person3Tests
    {
        [Fact]
        public void PersonToStringShouldBeCustomized()
        {
            var p = new Person3
            {
                Title = "Mrs",
                FirstName = "Sarah",
                SecondName = "Smith",
                BirthDate = new DateTime(2000, 1, 20)
            };

            Thread.CurrentThread.CurrentCulture = new CultureInfo("de-De");

            Assert.Equal(
                "Mrs Sarah Smith was born 01/20/2000 00:00:00",
                p.ToString());
        }
    }
}
```

This test will fail because the assert is expecting "..was born 01/20/2000 00:00:00" but the actual value was "20.01.2000 00:00:00".

String interpolation can be used with an explicitly specified culture by first creating an `System.IFormattable` variable and assigning it the interpolated string (from .NET 4.6). Next the `System.IFormattable` can be converted to a string and the required culture specified as a parameter as the following code demonstrates:

C# 6 Using an IFormattable to specify a culture

```
using System;
using System.Globalization;

namespace CSharp6
{
    public class Person4
    {
        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public DateTime BirthDate { get; set; }

        public override string ToString()
        {
            System.IFormattable formattable =
                $"{Title} {FirstName} {SecondName} was born {BirthDate}";

            return formattable.ToString(null, CultureInfo.InvariantCulture);
        }
    }
}
```

An shorthand version of the previous code is to use the static `FormattableString.Invariant` method as the following code shows:

C# 6 Using FormattableString.Invariant

```
using System;

namespace CSharp6
{
    public class Person5
    {
        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public DateTime BirthDate { get; set; }

        public override string ToString()
        {
            return FormattableString.Invariant(
                $"{Title} {FirstName} {SecondName} was born {BirthDate}");    \

        }
```

```
    }
}
```

Benefits

When using `string.Format`, if one of the values has been forgotten there will be no compile time error. For example `string.Format("{0} {1} {2} is {3} years old", Title, FirstName, SecondName)` is missing the `Age` variable. With string interpolation, it is harder to forget to include the variable as it is part of the string itself, rather than a `{0}` or `{3}`. If the following code was compiled (note the misspelled age variable) `$"{Title} {FirstName} {SecondName} is {Agge} years old"` the compiler will generate an error: "The name 'Agge' does not exist in the current context ". This compile time checking is a strong case to use string interpolation.

The Null-Conditional Operators

The null-conditional operators (`?.` and `?[]`) reduce the amount of code that needs to be written to check for null values.

Take the following example code that attempts to get the `Person`'s name. First the code checks that the person is not null. If the person is not null, the code can then access the `FirstName` property and check if that is not null. Finally now the code is sure that `p` and `FirstName` are not null, it can access the value of `FirstName` without causing a `NullReferenceException`.

C# 5 code to check for nulls

```
[Fact]
public void DealingWithNullPersonOrNullFirstName()
{
    var p = new Person
    {
        Title = "Mrs",
        FirstName = null,
        SecondName = "Smith",
        Age = 30
    };

    string firstName = null;

    // Check if p is not null and if not, check FirstName is not null
    if (p != null && p.FirstName != null)
    {
        firstName = p.FirstName;
    }

    Assert.Null(firstName);
}
```

Using the null-conditional operator `?.` the code can be shortened to the following:

C# 6 code using a null-conditional operator

```
[Fact]
public void DealingWithNullPersonOrNullFirstName()
{
    var p = new Person
    {
        Title = "Mrs",
        FirstName = null,
        SecondName = "Smith",
        Age = 30
    };

    // Check if p is not null and if not, get the value of FirstName
    string firstName = p?.FirstName;

    Assert.Null(firstName);
}
```

If `p` were null, this code will still execute without throwing a `NullReferenceException` as the following code shows:

Null Person

```
[Fact]
public void DealingWithNullPerson()
{
    Person p = null;

    string firstName = p?.FirstName;

    Assert.Null(firstName);
}
```

If neither `p` or `FirstName` are null, then the following test will pass:

Person and FirstName set

```
[Fact]
public void PersonAndNameNotNull()
{
    var p = new Person
    {
        Title = "Mrs",
        FirstName = "Sarah",
        SecondName = "Smith",
        Age = 30
    };

    var firstName = p?.FirstName;

    Assert.Equal("Sarah", firstName);
}
```

Value Type Results

If a null-conditional operator is applied to a value type result, the result could be null. Because of this if `var` is used the result will be a System.Nullable<T>. The following code shows an example of a nullable value type specified explicitly as `int?`.

Nullable value type result

```
[Fact]
public void ValueTypes()
{
    List<Person> people = null;

    // Causes a NullReferenceException
    // var numberOfPeople = people.Count;

    // Compile error - non nullable int
    // int numberOfPeople = people?.Count;

    int? numberOfPeople = people?.Count;

    Assert.Null(numberOfPeople);
}
```

Use With Array Indexers

Just as `?.` can be used with individual members, the `?[` null-conditional operator can be used with arrays as follows.

Nullable value type result

```
[Fact]
public void Arrays()
{
    List<Person> nullPeople = null;

    List<Person> people = new List<Person>();
    people.Add(new Person {FirstName = "A"});
    people.Add(new Person {FirstName = null});


    Assert.Null(nullPeople?[0].FirstName);

    Assert.Equal("A", people[0].FirstName);

    Assert.Null(people?[1].FirstName);
}
```

Use with the Null-Coalescing Operator

In previous C# versions, the null-coalescing operator returns the left hand value if not null, or the right hand value if null. For example `firstName = x ?? "n/a"`; if x is not null its value is used, if x is null the right hand "n/a" is used. The null-coalescing operator can be combined with the null-conditional operators.

Combining the null-coalescing and null-conditional operators

```
[Fact]
public void NullCoalescing()
{
    var p1 = new Person
    {
        FirstName = null
    };

    var p2 = new Person
    {
        FirstName = "Sarah"
    };

    Person p3 = null;

    var p1FirstName = p1?.FirstName ?? "n/a";
    var p2FirstName = p2?.FirstName ?? "n/a";
    var p3FirstName = p3?.FirstName ?? "n/a";

    Assert.Equal("n/a", p1FirstName);
    Assert.Equal("Sarah", p2FirstName);
    Assert.Equal("n/a", p3FirstName);
}
```

Thread-Safe Delegate Invocation

Before C# 6.0, to invoke a delegate in a thread-safe way the following pattern was often used:

C# 5.0 Checking for null before invoking a delegate

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace CSharp5
{
    class OrderCart : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void OnPropertyChanged(
                                        [CallerMemberName] string propertyName = null)
        {
            var handler = PropertyChanged;

            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

With C# 6.0, the code can be reduced to the following:

C# 6.0 Using the null-conditional operator to simplify delegate invocation

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace CSharp6
{
    class OrderCart : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void OnPropertyChanged(
                                        [CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this,
                                    new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Notice in the preceding code that no temporary variable is required to account for thread-safety, the compiler will now take care of this.

Benefits

There are many benefits to be gained by using the null-conditional operators. From reducing the amount of code that needs to be written (and read) to simplifying delegate invocation and potentially avoiding threading problems if the programmer forgets to account for it.

Considerations

Any time new language features are introduced that make code more terse, there may be criticisms that code is becoming more unreadable. The null-conditional operators could be thought of in this regard. The same criticisms could be directed at other operators such as `??`. Over time as familiarity increases, `?.` and `?[` will likely become everyday occurrences in the majority of C# code.

Getter Only Auto Properties

In C# 5.0 the use of auto-implemented properties required both a getter and setter to be specified. Non auto-implemented properties could specify just a getter though this then required additional code to return a value. With auto-implemented properties, the setter could be made `private` to make it unavailable to external callers, but the value could still be changed inside the class by accessing the private setter.

In C# 6.0 getter only auto-implemented properties allow the creation of immutable types with less code and more clarity/expression of intent.

C# 5 attempt at an immutable Point class

```
namespace CSharp5
{
    // Attempt at immutable class
    public class Point
    {
        public Point(int x, int y)
        {
            X = x;
            Y = y;
        }

        public int X { get; private set; }
        public int Y { get; private set; }

        public void SomeMethod()
        {
            // Bug:  X is not read only, the private
            // setter only hides it from the outside world
            X += 1;
        }
    }
}
```

In the preceding code, whilst an external caller cannot modify `X` or `Y` because of the private setter, they can be accessed internally as shown in the `SomeMethod` code. This could be a typo or mistake but there is no compile time error here. The intent is also unclear.

The following code shows a second attempt:

C# 5 immutable Point class

```
namespace CSharp5
{
    // Immutable class - increased lines of code
    public class Point2
    {
        private readonly int _x;
        private readonly int _y;

        public Point2(int x, int y)
        {
            _x = x;
            _y = y;
        }

        public int X
        {
            get { return _x; }
        }

        public int Y
        {
            get { return _y; }
        }

        public void SomeMethod()
        {
            // Cannot create same bug here because _x is readonly
            // cannot be changed here

            // _x += 1; // A readonly field cannot be assigned
            // to (except in a constructor or a variable initializer)
        }
    }
}
```

Notice in the preceding code, auto-implemented properties cannot be used because the backing fields _x and _y need to be made readonly to enforce the idea of immutability. If the same typo occurred in `SomeMethod` we would now get help from the compiler and get a build error.

There is however a lot more boilerplate code now, but there is also an increase in the intent because of the use of the `readonly` modifier.

C# 6.0 reduces this boilerplate code by allowing getter only auto-properties as shown in the following code:

C# 6.0 immutable Point class

```
namespace CSharp6
{
    // Immutable class
    public class Point2
    {
        public Point2(int x, int y)
        {
            X = x;
            Y = y;
        }

        public int Y { get; } // No setter
        public int X { get; } // No setter

        public void SomeMethod()
        {
            // Cannot create same bug here because X is read only

            // X += 1; // Error: Property or indexer 'Point2.X' cannot
            // be assigned to - it is read only
        }
    }
}
```

Notice that there is less code and also the intent is expressed more cleanly.

Getter only auto-implemented properties can be assigned to from the constructor or by using the new C# 6.0 property initializer features as shown in the following code that represents an immutable centre of a coordinate system.

C# 6.0 getter only initializers

```
namespace CSharp6
{
    // Immutable class
    public sealed class CentrePoint
    {
        public int Y { get; } = 0;
        public int X { get; } = 0;
    }
}
```

Using Await in Catch and Finally Blocks

In C# 5.0 the use of `await` in `catch` or `finally` blocks was not allowed. The following code in C# 5.0 would produce the build error "Cannot await in a catch clause."

C# 5.0 attempt at using await in a catch block

```
using System;
using System.Threading.Tasks;

namespace CSharp5
{
    public class Calculator
    {
        public async Task<int> Divide(int number, int by)
        {
            try
            {
                return number / by;
            }
            catch (Exception ex)
            {
                // Error: Cannot await in a catch clause
                // await SimpleLogger.LogAsync(ex.ToString());

                throw;
            }
        }
    }
}
```

In the preceding code, when an exception occurs it needs to be logged but in an asynchronous way using the `SimpleLogger` class.

SimpleLogger class writing asynchronously to a file

```
using System.IO;
using System.Threading.Tasks;

namespace CSharp5
{
    public static class SimpleLogger
    {
        public static async Task LogAsync(string message)
        {
            using (var q = new StreamWriter(@"c:\temp\demolog.txt",
                                            append: true))
            {
                await q.WriteLineAsync(message);
            }
        }
    }
}
```

With C# 6.0 this awaitable logging is now allowed to be called in the `catch` block:

C# 6.0 using await in a catch block

```
using System;
using System.Threading.Tasks;

namespace CSharp6
{
    public class Calculator
    {
        public async Task<int> Divide(int number, int by)
        {
            try
            {
                return number / by;
            }
            catch (Exception ex)
            {
                await SimpleLogger.LogAsync(ex.ToString()); // No error
                throw;
            }
        }
    }
}
```

In C# 6.0 `await` can also be used in the `finally` block.

Property, Dictionary, and Index Initializers

Property Initializers

In C# 5.0, to initialize an auto-implemented property we could perform the initialization in the constructor:

C# 5.0 initializing an auto-implemented property from the constructor

```
namespace CSharp5
{
    public class Person6
    {
        public Person6()
        {
            Title = "Mr";
        }

        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }
    }
}
```

With C# 6.0, property initializers can be used instead:

C# 6.0 auto-implemented property initializers

```
namespace CSharp6
{
    public class Person6
    {
        public string Title { get; set; } = "Mr";
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }
    }
}
```

This means that the constructor can be removed if it is only being used to initialize properties.

Getter only auto implemented properties can also use property initializers.

Dictionary Initializers

In C# 5.0, dictionaries can be initialized using key/value pairs as shown in the following code.

C# 5.0 dictionary initialization

```
using System.Collections.Generic;

namespace CSharp5
{
    public class Person7
    {
        public Person7()
        {
            Title = "Mr";
        }

        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }

        public Dictionary<string, string> ToNameDictionary()
        {
            return new Dictionary<string,string>
            {
                {"Title", Title },
                {"FirstName", FirstName },
                {"SecondName", SecondName}
            };
        }
    }
}
```

With C# 6.0 rather than use the (somewhat inelegant) key/value pair, dictionary initializers can simply use the index:

C#6.0 dictionary initialization

```
using System.Collections.Generic;

namespace CSharp6
{
    public class Person7
    {
        public string Title { get; set; } = "Mr";
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }

        public Dictionary<string, string> ToNameDictionary()
        {
            return new Dictionary<string,string>
            {
                ["Title"] = Title,
                ["FirstName"] = FirstName,
                ["SecondName"] = SecondName
            };
        }
    }
}
```

Index Initializers

Creation of an object that uses indexers required a temporary variable in C# 5.0 as the following code demonstrates:

Person class with an indexer for favourite colors

```
namespace CSharp6
{
    public class Person8
    {
        private readonly string[] _favouriteColors = new string[10];
        public string Title { get; set; }
        public string FirstName { get; set; }
        public string SecondName { get; set; }
        public int Age { get; set; }

        public string this[int index]
        {
            get
            {
                return _favouriteColors[index];
            }
            set
            {
                _favouriteColors[index] = value;
            }
        }
    }
}
```

C# 5.0 object indexer initialization

```
public Person8 MakeAPerson()
{
    var person = new Person8
    {
        Title = "Mrs",
        FirstName = "Sarah",
        SecondName = "Smith",
        Age = 30,
        // [0] = "Red" // Error
    };

    person[0] = "Red";
    person[1] = "Blue";
```

```
    return person;
}
```

With C# 6.0 index initializers, the temporary variable is no longer needed as initializing index values can be performed alongside regular property initializers.

C# 6.0 index value initialization

```
public Person8 MakeAPersonNoTemp()
{
    return new Person8
    {
        Title = "Mrs",
        FirstName = "Sarah",
        SecondName = "Smith",
        Age = 30,
        [0] = "Red",
        [1] = "Blue"
    };
}
```

Auto-Implemented Property Initializers With Indexers

This technique can also be used with property initializers as shown in the following code:

C# 6.0 index value initialization

```
namespace CSharp6
{
    public class Couple
    {
        public Person8 Partner1 { get; set; } = new Person8
        {
            Title = "Mrs",
            FirstName = "Sarah",
            SecondName = "Smith",
            Age = 30,
            [0] = "Red",
            [1] = "Blue"
        };
        public Person8 Partner2 { get; set; } = new Person8
        {
            Title = "Mrs",
            FirstName = "Alisha",
            SecondName = "Smith",
            Age = 32,
            [0] = "Orange",
            [1] = "Pink"
        };
    }
}
```

The nameof Operator

The new `nameof` operator returns a string representing the name of a variable, the name of a type (e.g. a class) or the member of a type (e.g. a property).

The following example code shows how `nameof` can be used to get a string representation of a variable name, a member of a type (the `Title` property), and a type name itself.

Usages of nameof

```
var sarah = new Person2
{
    Title = "Mrs",
    FirstName = "Sarah",
    SecondName = "Smith",
    Age = 30
};

string varName = nameof(sarah);
// varName now equals "sarah"

string titlePropertyName = nameof(sarah.Title);
// titlePropertyName now equals "Title"

string typeName = nameof(Person2);
// typeName now equals "Person2"
```

The main benefit of using `nameof` is to reduce the brittleness of code. Instead of having strings embedded in the application that match the name of a type/variable/member, it can be programmatically created with `nameof`. This means that the code (type/variable/member names) can be changed without having to go and manually change the corresponding string.

Notice in the preceding code that the line `string typeName = nameof(Person2);` returns the unqualified type name. To get the fully qualified name the existing `typeof` operator can be used, for example: `string fullTypeName = typeof(Person2).FullName;`.

To get the name of a member, it is not necessary to have an instance of the type, so to get the `Title` without an instance of `Person2` the following code can be written: `nameof(Person2.Title)`.

Parameter Validation Exceptions

When validating the incoming value of parameters in methods, with C# 5.0 the parameter name in the new exception is specified as a string. This means that it is possible to make a typo and spell the parameter name incorrectly. It also means that if the parameter name is refactored the developer needs to remember to also change the string.

C# 5.0 throwing parameter checking exceptions

```
using System;

namespace CSharp5
{
    public class Point3
    {
        private readonly int _x;
        private readonly int _y;

        public Point3(int x, int y)
        {
            if (x < 1)
            {
                throw new ArgumentOutOfRangeException("x");
            }

            if (y < 1)
            {
                throw new ArgumentOutOfRangeException("y");
            }

            _x = x;
            _y = y;
        }
    }
}
```

Notice in the preceding code that if the parameter names “x” or “y” change, the strings also need to be changed, i.e. `throw new ArgumentOutOfRangeException("x");`. Compare this with the following C# 6.0 version:

C# 6.0 throwing parameter checking exceptions

```
using System;

namespace CSharp6
{
    public class Point3
    {
        private readonly int _x;
        private readonly int _y;

        public Point3(int x, int y)
        {
            if (x < 1)
            {
                throw new ArgumentOutOfRangeException(nameof(x));
            }

            if (y < 1)
            {
                throw new ArgumentOutOfRangeException(nameof(y));
            }

            _x = x;
            _y = y;
        }
    }
}
```

In the preceding code, it is now impossible to set the wrong parameter name when creating the exception, if the parameter name `x` is changed it must also be changed in the line `throw new ArgumentOutOfRangeException(nameof(x));` otherwise there will be a compilation exception.

Reducing Maintenance Costs and Errors in Razor Views

The `nameof` operator can be used in Razor views to reduce the number of magic strings such as referencing actions or controllers as strings. These strings can be replaced with `nameof` references.

For example, in C# 5.0 `@Html.ActionLink("About", "About", "Home")` executes the `Action` method on the `HomeController` class. If the action method or controller are renamed, this creates a maintenance cost because the strings "About" and "Home" now also need to be changed in the Razor view.

With C# 6.0, the action string can be replaced with a `nameof` reference:

```
@Html.ActionLink("About", nameof(HomeController.About), "Home")
```

The approach can be extended to the "Home" reference:

```
@Html.ActionLink("About", nameof(HomeController.About), nameof(HomeController))
```

This code will produce an invalid link however; rather than "/Home/About" it will be "HomeController/About".

This could be "fixed" with the removal of the "Controller":

```
@Html.ActionLink("About", nameof(HomeController.About), nameof(HomeController)
                  .Replace("Controller", ""))
```

This behaviour could also be encapsulated into an extension method as shown in the following code:

Extension method to remove Controller

```
namespace WebApplication1
{
    public static class StringExtensions
    {
        public static string ToControllerString(
            this string fullControllerTypeName)
        {
            return fullControllerTypeName.Replace("Controller", "");
        }
    }
}
```

With this extension method, the link could be written:

```
@Html.ActionLink("About", nameof(HomeController.About), nameof(HomeController)
                  .ToControllerString())
```

Expression Bodied Functions and Properties

C# 5.0 already allows the use of a single expression rather than the entire statement body when using lambda expressions. The following code shows an example of this:

Lambda expressions in C# 5.0

```
var someNumbers = Enumerable.Range(1, 10);

// Lambda statement
var evenNumbersStatement = someNumbers.Where(i =>
{
    return i % 2 == 0;
});

// Lambda expression
var evenNumbersExpression = someNumbers.Where(i => i % 2 == 0);
```

With C# 6.0 the idea of this "expression body" is expanded and can be applied to method bodies and (getter only) properties. The use of these new expression bodied members can reduce the number of lines of code and the number of curly braces. Whilst they may take some getting used to they can improve the succinctness and readability of code.

Getter Only Expression Bodied Properties

In the chapter on Getter Only Auto Properties, the concept of getter only auto implemented properties was introduced. Another enhancement to property syntax in C# 6.0 is the use of expression bodies in getter only properties. This means that it is no longer necessary to have a block `{....}` that encloses a single `return` statement in the `get`.

The following code shows a getter only property `Area` implemented in C# 5.0.

C# 5.0 getter only statement body

```
namespace CSharp5
{
    public class Square
    {
        public int SideLength { get; set; }

        public int Area
        {
            get
            {
                return SideLength * 2;
            }
        }
    }
}
```

In the preceding code, the entire class file definition is 15 lines long; compare this to the following shorter C# 6.0 version that uses an expression body for the getter only property:

C# 6.0 getter only expression bodied property

```
namespace CSharp6
{
    public class Square
    {
        public int SideLength { get; set; }
        public int Area => SideLength * 2;
    }
}
```

Expression Bodied Methods

The same syntax (with the addition of the usual method parenthesis after the method name) can be applied to methods that only have a single return; again this reduces the need for a statement block `{...}` and can reduce the number of lines of code in the file.

C# 5.0 method with statement body

```
namespace CSharp5
{
    public class Square2
    {
        public int SideLength { get; set; }

        public int Area
        {
            get
            {
                return SideLength * 2;
            }
        }

        public int CalculatePerimeter()
        {
          return SideLength * 4;
        }
    }
}
```

C# 6.0 method with expression body

```
namespace CSharp6
{
    public class Square2
    {
        public int SideLength { get; set; }
        public int Area => SideLength * 2;
        public int CalculatePerimeter() => SideLength * 4;
    }
}
```

Expression Bodied Operator Overload Methods

Operator overload methods that contain a single return can also take advantage of expression bodies. The follow shows an updated Square class that declares an implicit string conversion:

C# 5.0 operator overload

```
using System.CodeDom;

namespace CSharp5
{
    public class Square3
    {
        public int SideLength { get; set; }

        public int Area
        {
            get
            {
                return SideLength * 2;
            }
        }

        public int CalculatePerimeter()
        {
          return SideLength * 4;
        }

        public static implicit operator string(Square3 square)
        {
            return string.Format("{0}x{0}", square.SideLength);
        }
    }
}
```

Compare this with the following C# 6.0 version (that also replaces the `string.format` with string interpolation):

C# 6.0 expression bodied operator overload

```
namespace CSharp6
{
    public class Square3
    {
        public int SideLength { get; set; }
        public int Area => SideLength * 2;
        public int CalculatePerimeter() => SideLength * 4;

        public static implicit operator string(Square3 square) =>
            $"{square.SideLength}x{square.SideLength}";
    }
}
```

Test Methods

Expression bodied methods can also be used with `void` methods. One use for this is to reduce the number of lines of test code.

The follow class shows a number of SpecFlow[9] steps:

C# 5.0 SpecFlow steps

```
using System;
using TechTalk.SpecFlow;
using Xunit;

namespace CSharp5
{
    [Binding]
    public class Calculator2Steps
    {
        private Calculator2 _calculator;

        [Given(@"I have a clear calculator")]
        public void GivenIHaveAClearCalculator()
        {
            _calculator = new Calculator2();
        }

        [When(@"I add (.*)")]
        public void WhenIAdd(int number)
        {
            _calculator.Add(number);
        }

        [Then(@"The value should be (.*)")]
        public void ThenTheValueShouldBe(int expectedValue)
        {
            Assert.Equal(expectedValue, _calculator.Value);
        }
    }
}
```

The methods in the preceding class contain only a single statement, the following shows the equivalent C# 6.0 version:

[9] http://dontcodetired.com/blog/?tag=/specflow

C# 6.0 SpecFlow steps using method expression bodies

```
using System;
using TechTalk.SpecFlow;
using Xunit;

namespace CSharp6
{
    [Binding]
    public class Calculator2Steps
    {
        private Calculator2 _calculator;

        [Given(@"I have a clear calculator")]
        public void GivenIHaveAClearCalculator() =>
            _calculator = new Calculator2();

        [When(@"I add (.*)")]
        public void WhenIAdd(int number) => _calculator.Add(number);

        [Then(@"The value should be (.*)")]
        public void ThenTheValueShouldBe(int expectedValue) =>
            Assert.Equal(expectedValue, _calculator.Value);
    }
}
```

Exception Filters

Exception filters in C# 6.0 allow the writing of more readable catch blocks.

With C# 5.0, to perform some action when a particular exception is thrown *and* the exception object meets some condition, the following code was required:

C# 5.0 Exception "filtering"

```
using System.Net;

namespace CSharp5
{
    public class WebDownloader
    {
        public string Download()
        {
            using (var web = new WebClient())
            {
                try
                {
                    return web.DownloadString(
                        "http://dontcodetired.com/notexisto");
                }
                catch (WebException ex)
                {
                    if (ex.Status == WebExceptionStatus.ProtocolError)
                    {
                        return "DEFAULT CONTENT";
                    }

                    throw;
                }
            }
        }
    }
}
```

With C# 6.0, the `catch` expression can be extended with the `when` keyword:

C# 6.0 Exception filtering with the when keyword

```
using System.Net;

namespace CSharp6
{
    public class WebDownloader
    {
        public string Download()
        {
            using (var web = new WebClient())
            {
                try
                {
                    return web.DownloadString(
                        "http://dontcodetired.com/notexisto");
                }
                catch (WebException ex) when (ex.Status ==
                                       WebExceptionStatus.ProtocolError)
                {
                    return "DEFAULT CONTENT";
                }
            }
        }
    }
}
```

Notice in the preceding code, an exception filter is of the form `where (Boolean expression)`. This Boolean expression (predicate) can use the usual logical operators (&& || etc.).

The code inside an exception filtered catch block will only be executed if the exception is of the correct type and the predicate expression returns `true`.

In addition to an expression, an exception filter can also call a (Boolean returning) method as shown in the following code:

C# 6.0 Exception filter calling a method

```
using System.Net;

namespace CSharp6
{
    public class WebDownloader2
    {
        public string Download()
        {
            using (var web = new WebClient())
            {
                try
                {
                    return web.DownloadString(
                        "http://dontcodetired.com/notexisto");
                }
                catch (WebException ex) when (ShouldHandle(ex))
                {
                    return "DEFAULT CONTENT";
                }
            }
        }

        private static bool ShouldHandle(WebException ex)
        {
            return ex.Status == WebExceptionStatus.ProtocolError;
        }
    }
}
```

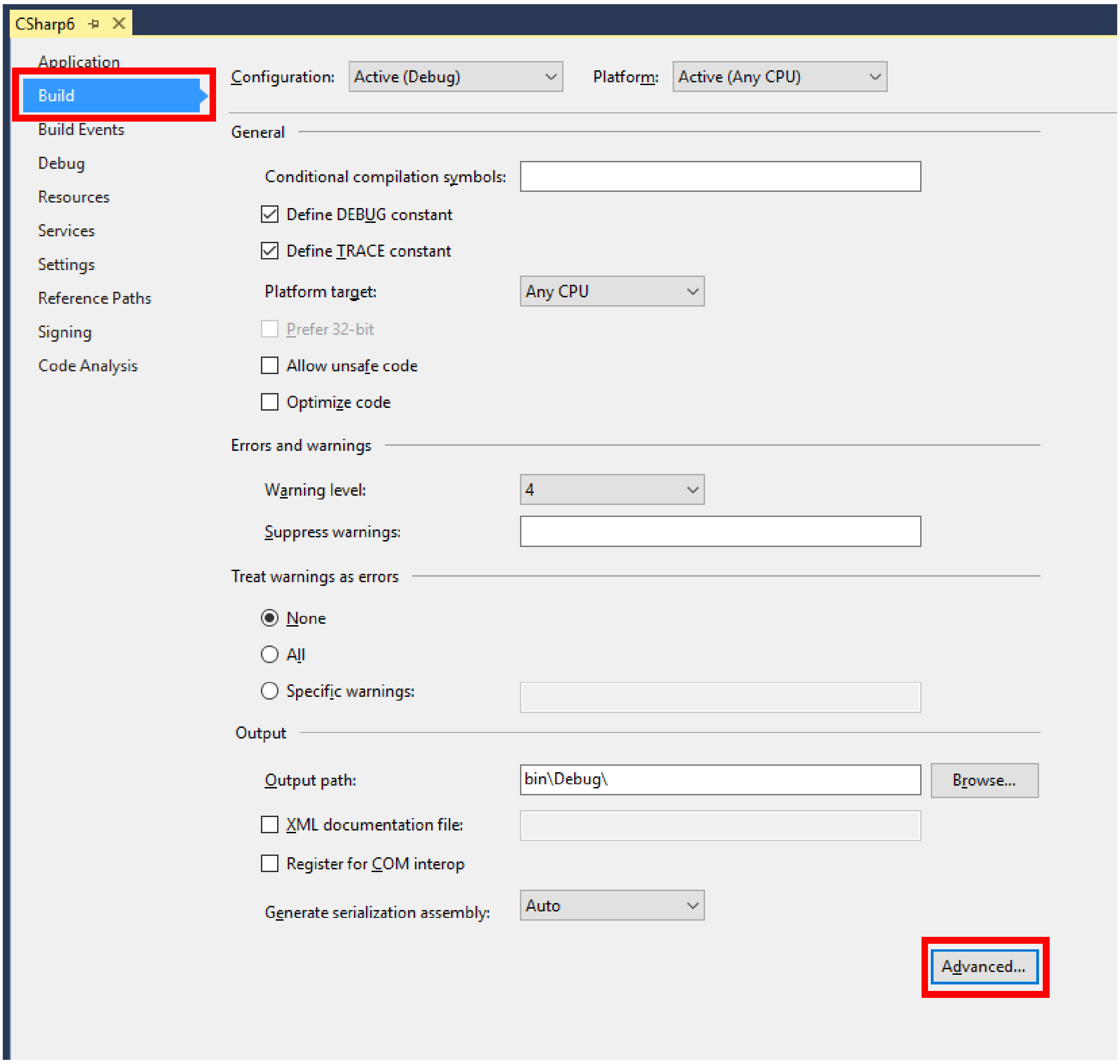
Visual Studio 2015 and C# 6

Downgrading a project to C# 5.0

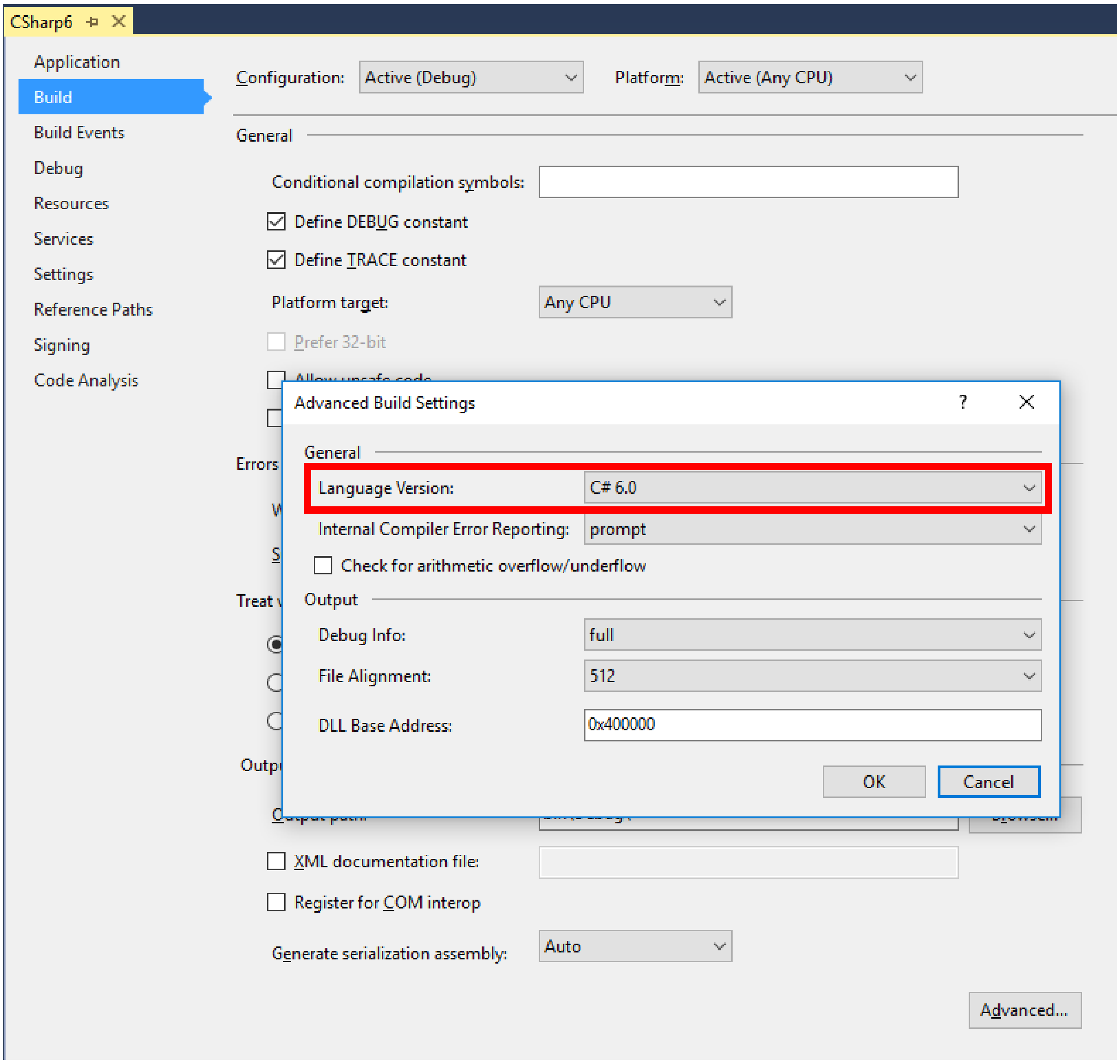
If you need to interoperate with other developers working in a version of Visual Studio that does not support C# 6.0, the project properties can be modified to compile for C# 5.0 only language features. Another example of using this feature might be for corporate policy reasons, for example the use of Visual Studio 2015 has been authorized but not the use of C# 6.0.

To "downgrade" a project and enable only C# 5.0 language features the following procedure can be followed.

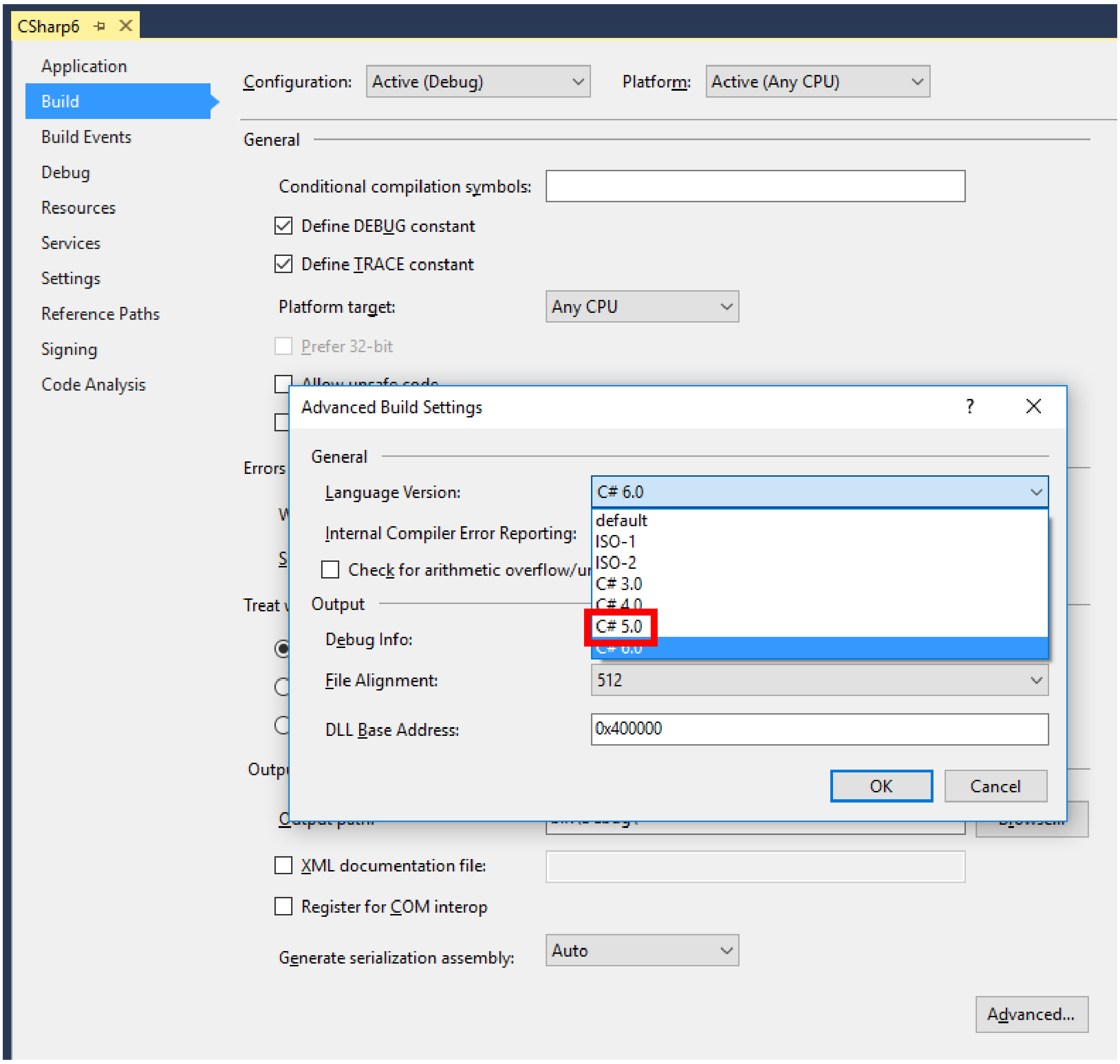
1 - Open Advanced Project Build Options



2 - Select Language Version Dropdown



3 - Choose C# 5.0


CSharp6
Application
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing
Code Analysis
Configuration: Active (Debug)
Platform: Active (Any CPU)
General
Conditional compilation symbols:
Define DEBUG constant
Define TRACE constant
Platform target: Any CPU
Prefer 32-bit
Errors
Treat
Outp
XML documentation file:
Register for COM interop
Generate serialization assembly: Auto
Advanced...
Advanced Build Settings
?
General
Language Version: C# 6.0
default
ISO-1
ISO-2
C# 3.0
C# 5.0
Internal Compiler Error Reporting:
Check for arithmetic overflow/u
Output
Debug Info:
File Alignment: 512
DLL Base Address: 0x400000
OK
Cancel