

The State of C#

By Kevin Griffin

WHITEPAPER

Table of Contents

A Brief History Lesson	/ 3
Career Landscape for a C# Developer	/ 4
Tools of the Trade	/ 5
Platforms	/ 7
Fifteen Years of Innovation and Features	/ 10
What is New in C# 7.0?	/ 11
Stable Foundation, Stable Career	/ 17

A Brief History Lesson

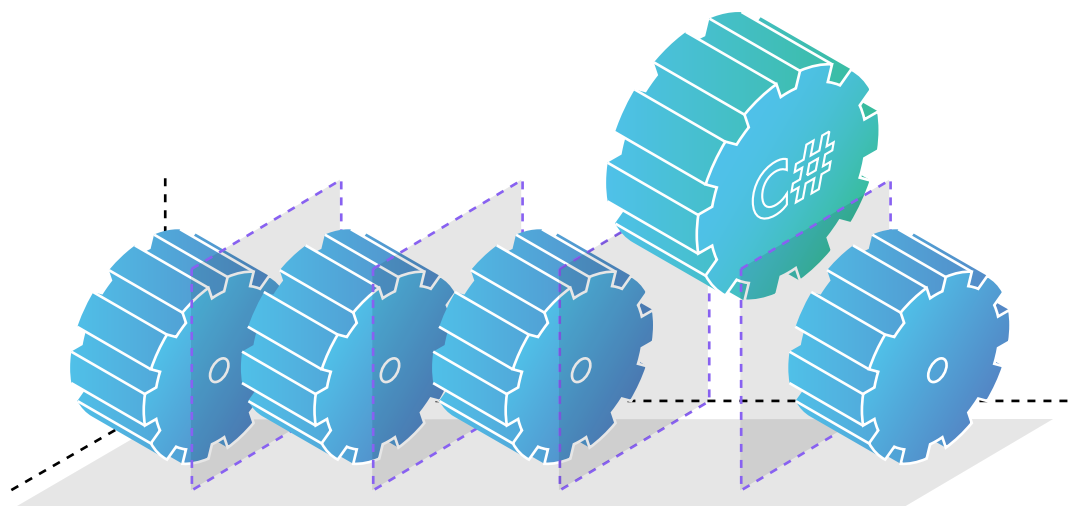
Imagine being a software developer in the late 90's. You have a wide variety of tools at your disposal: C++, Java, Delphi, Visual Basic, FoxPro—and this list goes on. Each of these languages came with an exhaustive list of benefits, but also with its own unique lists of flaws.

In the early 2000's, Microsoft publically announced plans for the .NET Framework, which was a series of managed libraries. This new framework, however, needed a new language. Originally codenamed "Cool," C# was born.

C# was originally designed to be an open standards-based language. Each version of C# has gone through approval by not just Microsoft, but by ECMA International and the International Organization for Standardization (ISO). What does that mean to a developer? Anyone can build a C# compiler for their operating system. The Mono Project, which is an open-sourced implementation of the C# compiler and the .NET Framework, is a perfect example of the C# standard in action.

C# has gone through five major revisions since its initial 1.0 in 2002. Over this time, there have been a multitude of significant additions such as: generics, anonymous methods and types, automatic properties, expression trees, dynamics, asynchronous methods and compiler-as-a-service. This is not an exhaustive list of course, but you can see how C# has grown over the years.

Whether you are a seasoned professional or a student looking to find a direction in this industry, the question you should ask yourself is, "Should C# be a viable tool to have in my toolbox?"



Career Landscape for a C# Developer

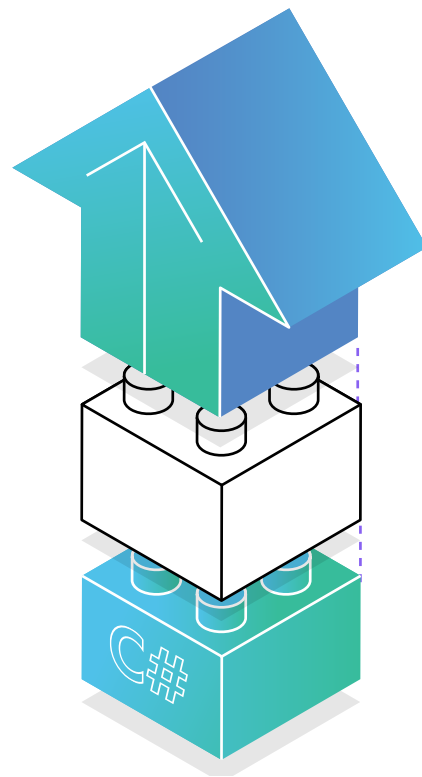
In its [annual survey of the technical community](#), StackOverflow rated C# as the 5th top tech to know for 2016. In addition to this, developers were asked to rate the languages they loved, dreaded and wanted to use most. If we look at a correlation of top tech compared to most loved, you would find that C# a language that is both loved by developers and in the top tech stack.

What does that mean for other languages in the top tech stack? While there is a lot of development being done in JavaScript, Java and Android, none of those are languages that developers would admit to love working with. C# falls into both categories.

What does the salary of an average C# developer look like? A full stack developer that uses C# can make on average \$95,000 per year. This is often combined with another useful skill, such as JavaScript. The only way to attain a higher average salary, say \$105,000, would require a developer to work with cloud-based technologies like React or Redis.

For a front-end developer, the pay scale decreases to an average of \$75,000 per year. The survey does not state whether front-end specifically means web developer or include client application development.

A C# developer specializing in either data science or machine learning can earn on average \$85,000 per year. The same number is also true for mobile developers that work with C# as their primary development language. Even though there is a slight leaning towards developers focusing on the iOS platform versus Android, it doesn't seem to reflect too much on a C# developer who is capable of developing for both platforms using Xamarin.



Tools of the Trade

The most well-known developer tool in the C# ecosystem is Visual Studio, an Integrated Development Environment. It is important to understand that Visual Studio is not just limited to C#. In fact, in the years leading up to the creation of C#, Visual Studio was the platform of choice for developers of Windows-based applications using C/C++ and Visual Basic.

Today, a wide variety of languages is supported within Visual Studio. These languages include C/C++, Visual Basic, C#, F#, Python, Ruby, HTML, JavaScript, CSS and more.

This variety of language support is made possible by the highly extensible plugin system exposed within the editor. It allows for a developer within Microsoft, and even out in the community at large, to develop plugins that work directly with the Visual Studio subsystems.

There are many features of Visual Studio that keep developers coming back. First, there is the world-class code editor. As you type code into the editor, you are often assisted directly by Visual Studio through a feature called Intellisense. This feature analyzes your code after every keystroke, and can often provide auto-completion of variables and methods. This intelligent foresight is crafted specifically for the language you're developing in, meaning C# Intellisense and CSS Intellisense are different experiences that solve the same type of problem. The built-in debugging tools help you narrow down specific problems and step through them one line of code at a time.

Is there a feature Visual Studio is missing, but would make your development experience much better?

There is a large group of industry vendors who develop tools to augment and enhance the development experience inside of Visual Studio. For example, [Telerik by Progress](#) is the maker of JustCode, a Visual Studio extension that makes coding faster and easier by adding more intelligence and shortcuts into the code editor than what comes out of the box.

Previous iterations of Visual Studio required payment directly for the license or allowed you access through an active MSDN subscription. With the release of Visual Studio 2015, Microsoft introduced [Visual Studio Community Edition](#), a free version of the Visual Studio platform that allows support for any plugin you would like to install. Unlike the previous Visual Studio Express SKUs, any application created by an individual developer can be used for commercial purposes.

Visual Studio is the Rolls Royce of development environments. The largest hurdle for some developers is that Visual Studio can be a heavy tool to use and is only supported on Windows. The industry today is more supportive of tools that can work across different platforms, specifically OSX or Linux.

In 2015, at the Microsoft Build conference, Microsoft announced the [Visual Studio Code](#) editor. Visual Studio (VS) Code is an open-source code editor built for use on Windows, OSX and Linux.

Even though it is a more lightweight tool, VS Code supports many of the same features that you can find in Visual Studio. It has built-in syntax highlighting support for a multitude of languages, such as C#, C++, F#, Elixir, Docker, Python, Ruby and more!

VS Code does not offer the complete Intellisense solution you would find in Visual Studio. VS Code, however, does provide robust code completion functionality for several languages.

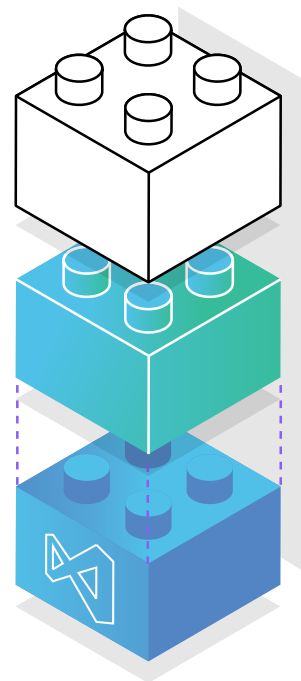
For developers working with C#, Node.js and Python, there is additional built-in debugging support.

By combining lightweight architecture and cross platform support, VS Code is an excellent tool for a developer getting started with C#.

Lastly, if you are already a fan of another editor that isn't VS Code, you are not out of luck. Developers from Microsoft, along with many outside contributors, have built an amazing plugin called [Omnisharp](#). With Omnisharp, you can inject the best parts of Visual Studio's Intellisense directly into your editor of choice.

Currently, Omnisharp is supported on Atom, Brackets, Emacs, Sublime Text and Vim. It is also the engine that powers the code completion features of VS Code.

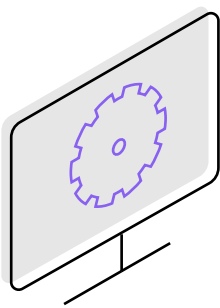
The underlying goal of all these tools, regardless of which one you choose to use, is that as a developer working with C# and the .NET Framework, your experience should be like none other.



Platforms

What have we covered so far? C# is a standards-based language that has grown exponentially in its feature set over the past 15 years. Combine that with a series of world-class development tools, and you are setting yourself up for a great career of development focused solely on Windows, correct?

Not at all! Given an investment of time and energy into C#, there is no limit to the number of platforms you will be able to develop for. Next, we will cover a few of the most common environments C# developers can deploy to.



Desktop

Building applications for Windows has always been a staple for the C# developer. In the beginning, our applications were written using Windows Forms (WinForms). WinForms, combined with the powerful designer tools within Visual Studio, allows a developer to quickly scaffold the look and feel of an application. Business logic for control interactions was simply done through the use of event handlers.

WinForms could also be extended further by the use of third-party control suites, such as [Progress Telerik WinForms](#). These control suites offer a wide variety of features that are missing from the base controls provided out of the box. Controls such as data grids, charts and more could easily take a team of developers months to build from scratch, and even then, these controls would not be as feature rich as what you would get from UI for WinForms.

A side effect of having an easy to use designer workflow for WinForms is that applications started to conform to the same look and feel. The term “battleship gray” was well known throughout the industry because it reflected how all WinForms applications looked. Deviating from the standard was difficult to do, even with the assistance of third-party tools.

With the release of C# 3.0, Microsoft announced the [Windows Presentation Foundation](#), or (WPF). This framework rethought the process of building Windows-centric applications. Instead of a designer-centric approach, developers could build out their user interfaces with a new markup language called XAML.

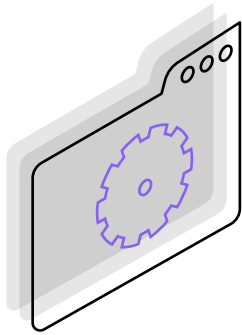
Applications built with WPF were still using C# underneath the covers. Yet developers now had the capability to build more robust interfaces as easily as they could with HTML.

Those who wanted a design experience similar to WinForms could use [Blend](#), a designer tool similar to Photoshop but specialized in generating XAML. As with WinForms, a developer who didn't want to reinvent the wheel could use one of the amazing third-party control suites available, such as UI for WPF.

The release of Windows 10 opened up a new world of opportunities for Windows developers. Windows is not just a desktop operating system anymore. It can run on desktops, but also on Windows Mobile devices, Xbox One and HoloLens.

The introduction of the [Universal Windows Platform](#) (UWP) created an environment where a developer could build a single application that could run natively on Windows 10, but also cross deploy to Windows 10 Mobile and Xbox One with minor adjustments. Underneath the veneer, there is a powerful subsystem accessible through C#, C++, Visual Basic and JavaScript. Existing skill sets with WPF and XAML easily translate.

Web



Next, let us turn our attention to the Web Stack, namely ASP.NET. Over the past several years, there have been three revolutions in the ASP.NET stack.

In the early days of .NET and WinForms, the Internet was still in its infancy. The concept of building a line of business application on the web was relatively new, and the tools were not designed to handle applications of this scale.

How does a developer-focused company like Microsoft take millions of developers with knowledge of WinForms and similar design patterns, and convert them to being web developers? The answer

lies in the birth and initial release of ASP.NET WebForms.

Any developer who had experience with WinForms could quickly translate those skills to the web. After all, similar to WinForms, a WebForms application started in a designer and supported application control through the use of event handlers.

At its core was, once more, C#.

The web has matured greatly over several years since the inception of WebForms. In 2007, ASP.NET dropped the initial version of ASP.NET MVC, an implementation of the Model View Controller pattern built on top of the ASP.NET framework. Developers wanting a purer environment to build web applications could use ASP.NET MVC to build applications that didn't have the overhead of ASP.NET WebForms.

There was one fundamental flaw with building ASP.NET applications—your deployment strategy focused solely around using Internet Information Services (IIS) on a Windows Server. Windows-based hosting was often more expensive to set up and maintain versus Linux or similar operating systems. The ASP.NET ecosystem was primarily for the enterprise, who could afford the licensing fees, or for low traffic sites that could run efficiently within shared or collocated environments.

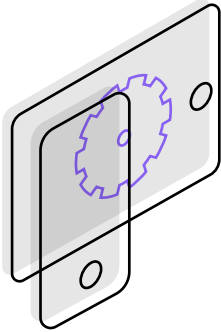
In late 2015, ASP.NET went through another renaissance by introducing ASP.NET Core (and the .NET Core framework).

Unlike its predecessor's, [ASP.NET Core](#) is an open-source, open-platform reimagining of how ASP.NET works. We are still in the early days of this platform, but imagine being able to build a full web application on ASP.NET that can be deployed quickly and easily

to either Windows, Linux, or even OSX without any code changes.

Currently, the only languages supported with ASP.NET Core are C# and F#, whereas previously Visual Basic was fully supported as well.

Mobile



Okay, C# is a viable language for the desktop and the web, but it doesn't do any good for a developer looking to focus on mobile applications, right?

As discussed above with Universal Windows Platform applications, you can easily target Windows 10 Mobile with a few code modifications. If we look at mobile market share for 2016, however, Windows Phone falls below 3% for the United States; the top competitors, of course, being iOS and Android, which consist of the majority of mobile users.

The fallacy in the industry is if you want to develop iOS applications, you need to learn Objective-C or Swift. If you want to develop Android applications, you need to know Java.

In February 2016, Microsoft bought Xamarin, a framework for building iOS and Android applications on top of the [Mono platform](#).

The trick to Xamarin is that you can build native Android, native iOS and even native Windows

applications using the same common codebase build with C#. No other platform can boast this type of feature set. Even with cross-platform mobile solutions such as PhoneGap or WebView, the experience is a simple shell that applications are injected into. Xamarin gives developers direct access to the native look and feel iOS and Android users expect to see.

Let us recap all of the platforms a C# developer could find themselves deploying to. First, on the desktop, you could write or maintain applications built with Windows Forms, Windows Presentation Foundation and the new Universal Windows Platform. Next is the web, where an application could be built with ASP.NET WebForms or ASP.NET MVC. Platform conscious developers can now even take advantage of ASP.NET Core, allowing for work on not just Windows, but Linux or OSX. Lastly, the mobile market is obtainable through using Xamarin to build cross-platform mobile applications that share a common codebase written in C#.

Fifteen Years of Innovation and Features

Up until now, we have talked at great length about the current state of the C# ecosystem. There are easily 15 years of platforms and systems built on top of C#. This is not the peak of the language however. Rather, all of this innovation is possible because C# continues to mature and grow as a language.

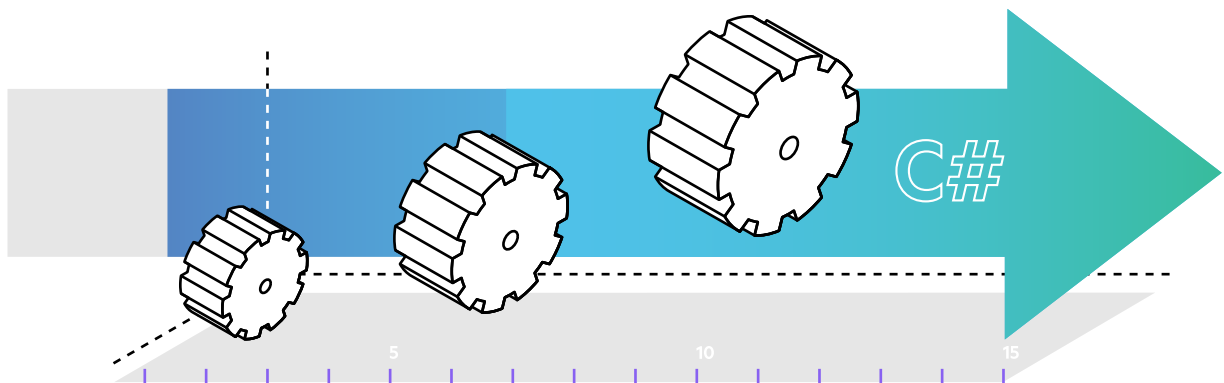
What are some major features that have been exposed through 15 years of innovation? If you are already an existing C# developer, these are probably features you take for granted.

Something as simple as Generics, Partial Types and Nullable types didn't exist until version 2.0 of the C# specification. Version 3.0 of C# introduced Extension Methods, Expression trees, Lambda expression and more. These features allowed LINQ (Language Integrated Query) to be possible, a feature that ultimately changed how C# developers worked with data.

C# 4.0 radically changed how developers used types, by introducing dynamic types to a statically typed language. Asynchronous methods were added in C# 5.0, which provided developers a process for building more performant applications by allowing long running tasks to execute asynchronously.

Lastly, in C# 6.0, we saw the Roslyn compiler (or Compiler as a service), which provides the C# compiler to developers to use at run-time. It also provides code analysis tools and powers many of the developer features built into Visual Studio.

This list is by no means exhaustive, but it provides a broad landscape of how C# has matured over the years. If you are new to C#, take some time to review the entire list of [features](#).



What is New in C# 7.0?

Next, we will shift gears and talk about what is coming down the line. We will examine the question, “What is new in C# 7.0?”

It is important to note that the following examples and concepts are still in preview. Their context and syntax is still being actively developed, and there is no guarantee the features will not be changed or removed in the future.

Out Variables

Using the **out** keyword within C# is nothing new. If you declare a variable within a method called with **out**, you are instructing the compiler that you are expecting the method to set the values of those at runtime.

```
public void TestMethod()
{
    string fullName;
    GetPersonName("Kevin", "Griffin", out fullName);
}
```

Commonly, the problem is that you have to declare the variable before the method call using **out**. In C# 7.0, there is the concept of out variables, which will save you a couple keystrokes by allowing you to declare the variable inline.

The above example can be quickly refactored:

```
public void TestMethod()
{
    GetPersonName("Kevin", "Griffin", **out string fullName**);
}
```

In parallel, if you do not know the type of the parameter, you can substitute the type of the parameter with **var**.

Pattern Matching

In C# 7.0, there is a heavy emphasis on pattern matching. Simply described, C# 7.0 will have the ability to see if data follows in a certain pattern or shape. As you will see in the next section, C# 7.0 will also have the ability to automatically extract a value if a pattern matches successfully.

The next example shows pattern matching in action with a Switch statement:

```
switch(person)
{
    case CEO ceo:
        CallCeo(ceo);
        break;
    case Manager salesManager when (salesManager.Department == "Sales"):
        CallSalesManager(salesManager);
        break;
    case Manager other
        CallManager(other);
        break;
    default:
        WriteLine("Standard employee");
        break;
    case null:
        throw new ArgumentNullException(nameof(person));
}
```

Walk through the Switch statement case-by-case. A “person” object is being fed into your Switch statement, and we want to evaluate the switch statement differently depending on the type and shape of the data within person.

In the first **case**, if the object matches the same pattern as a **CEO** object, we will match and call the appropriate case handler. Next, we have two types of statements that test whether the object is a manager. In contrast to that, in the case of a manager being from the “sales” department, our goal is to perform a different function than we would with a manager from any other department. The case statement can test for this by using the **when** keyword to perform a test against a generated salesManager object.

There are two fallback scenarios. Most commonly, the default case is that we passed an object that doesn’t match any of the previous patterns. Finally, there is also a null case that will ensure the object passed in has a value, and throw a null reference exception when a null is passed instead.

Is-Expressions

Building slightly on the above explanation of out variables, we can use pattern variables to create variables on demand within a block of code.

```
public void LoadPerson(object age)
{
    if (age is int ageInt || (age is string ageString &&
        int.TryParse(ageAsString, out ageInt)))
    {
        // ageInt contains a value.
    }
}
```

Looking at the code example, we have a case where the “age” of a person might come into our application as either a string or an integer. The `if` statement is an integer, immediately dropped into the `if` statement body. However, if “age” is a string, we will need to perform a conversion using the `int.TryParse` method.

Previously, the biggest issue with `TryParse` has been the need to define the variable outside of the `TryParse` method call. In C# 7.0, this is no longer the case. You can create the variable on demand if the `TryParse` method returns `True`.

Tuples

In C# 3.0, the `Tuple<>` reference type was added. What is a Tuple? Simply put, a Tuple is a collection of values.

Imagine our above example for loading a person’s information. There are several ways we can return data from a method:

```
public Person LoadPerson(){}
```



```
public Tuple<string, string> LoadPerson(){}
```

The first example returns a fully qualified type. Depending on your use case, this is perfectly acceptable. However, what if the type **Person** was throwaway? You are going to retrieve the data, use it and throw it away. Creating the **Person** type to simply support a single method is rather cumbersome.

The second example uses the **Tuple<>** generic type to retrieve the same information. What does a call to **LoadPerson** look like?

```
public void Foo()
{
    var person = LoadPerson();
    var fullName = $"{person.Item1} {person.Item2}";
}
```

For a simple, two-value Tuple, we are starting to veer off the road of readability. What is **Item1** or **Item2** supposed to reflect? Is it first name and last name, respectfully? Could it be name and address? City or state? There isn't much of a way to know because **Item1** and **Item2** are as explanatory as "a" or "b".

With C# 7.0, we are going to have access to real Tuples in a way that reflects closer to functional languages such as F#. Our **LoadPerson()** method signature would refactor as:

```
public (string firstName, string lastName) LoadPerson(){};
```

Notice the new syntax for the return type. The Tuple return type acts similar to the pattern you use for declaring function parameters. Lastly, if you want to create a Tuple inline, that can be done by using the **new** keyword.

```
return new (string firstName, string lastName) { firstName = "Kevin", lastName = "Griffin"};
```

Digit Separators

File this feature under "crazy useful." How many times have you created a numeric literal and had to count the number of digits to ensure you entered the correct number?

For example:

```
long Gigabyte = 1048576;
```

Seemingly harmless number, unless you forget a digit. Normally, if someone wrote this number longhand, it

would be represented as 1,048,576. The commas help dictate the position of the digits.

New “digit separators” in C# 7.0 allow you to do the same thing, except, instead of using a comma, you can use an underscore.

```
long Gigabyte = 1_048_576;
```

The compiler will ignore the underscores at build time.

Throw Exceptions

Writing defensive code often means throwing a lot of exceptions. In C# 7.0, there are new adjustments to where you can throw an exception.

For example, take this existing method:

```
public string GetName(Person person)
```

```
{  
    if (person.Name == null)  
        throw new Exception("Name not found");  
    return person.Name;  
}
```

With the new changes to where we can throw exceptions, that method can be quickly refactored to:

```
public string GetName(Person person)  
{  
    return person.Name ?? throw new Exception("Name not found.");  
}
```

Non-Nullable Reference Types

One of the most common exceptions thrown in .NET applications is the Null Reference exception. What is the problem? Reference types in C# are nullable by default. This precondition causes developers to bulletproof

their code constantly to ensure they are not attempting to access a variable that could potentially be null at runtime.

For value types, such as **int** or **DateTime**, it is not possible for the value to be null unless you create it as a Nullable type.

```
int? intCouldBeNull;
```

```
DateTime? nullableDateTime;
```

What if the designers of C# simply changed the defaults? All reference types were non-nullable by default, and you explicitly had to set a variable are nullable.

That would break 15 years of C# code already out in the wild.

As an alternative, what if we could explicitly declare that a value type was “non-nullable?” In C# 7.0, the “!” operator is used to dictate that a variable is non-nullable.

```
string! nonNullableString;
```

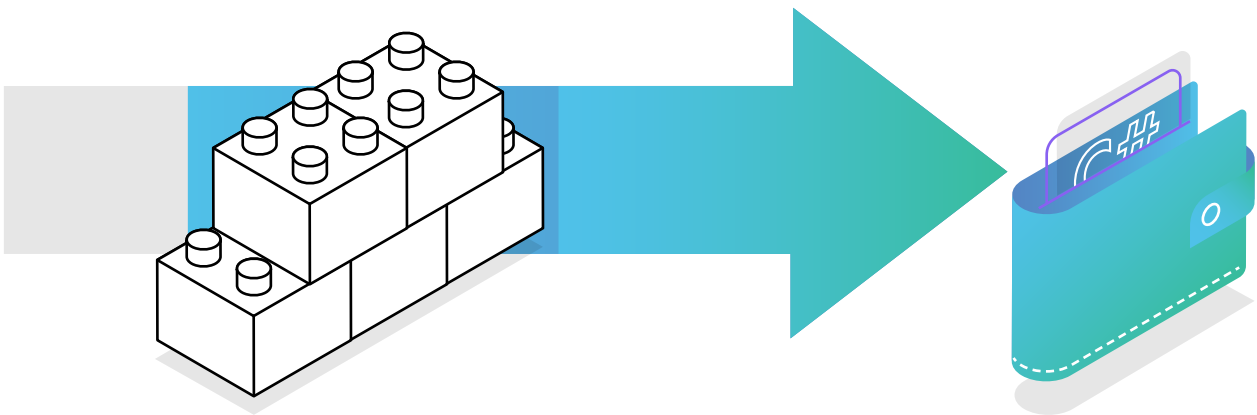
```
Person! nonNullablePerson;
```

By using the “!” operator, we are telling C# that **nonNullableString** and **nonNullablePerson** should NEVER be set to null. In the case of a null value being passed, this should cause a compiler error or warning.

Stable Foundation, Stable Career

If you are new to the industry or a seasoned professional, you have a huge decision to make about the technologies you want to align yourself with. C# has shown itself over time to be not only a stable platform, but a language which has matured and grown over the years to adapt as developer needs have changed. The latest version, C# 7.0, is currently in preview. It is boasting a plethora of new features designed to make code easier to read and write.

StackOverflow, the single most reliable resource for developers on the Internet, states that C# is one of the five top tech platforms in the world. If you compare the top tech category to the survey of languages that developers love to use, only C# will fall on both lists. Not even Java, which also holds rank as one of the top techs in the industry, can say it is loved by developers. To add icing to the cake, a developer who specializes in C# can, on average, earn between \$85,000 to \$95,000 per year. C# is a triple threat: loved, in demand and high paying.



A C# developer is not limited to a life of building battleship gray Windows applications either. As we discussed previously, C# allows you to build Windows applications on the new Universal Windows Platform, or you can use your skills to support older Windows Forms based applications. Web developers can now build fully functional web applications that run on Windows Server, Linux, or OSX. Xamarin is a framework for C# developers to build shared codebases which can be used to build native mobile applications.

World-class applications need world-class development environments. For over 15 years, Visual Studio has been the Cadillac of integrated development environments. With features like Intellisense, an amazing debugger, and a rich ecosystem of partner add-ons, Visual Studio continues to push forward the concept of what an IDE should be.

Developers who don't need all the power of Visual Studio, or want a similar powerhouse experience on OSX or Linux, should look at using Visual Studio Code. Open source and cross platform, Code gives you an amazing coding experience in a smaller package. If you already have an editor and cannot be convinced to switch, then the Omnisharp plugin allows you to take the C# Intellisense experience with you into whichever editor you use.

Is C# a good decision? The evidence above speaks wonders about the long-term viability of the language, the vast ecosystem of developer tools and the numerous places your knowledge can be applied. No matter if it you are targeting the desktop, the web, or mobile, C# is a sure bet for years to come.

Brought to You by Progress® Telerik®

Progress Telerik is a comprehensive .NET toolbox offering 600+ UI controls for all .NET technologies, HTML5, and Xamarin—plus code quality and reporting tools. Out-of-the-box support for numerous business scenarios ensures lower development cost, increased developer productivity and shorter time to market. A familiar API, similar to the Microsoft one, thousands of demos with source code and comprehensive technical documentation make Progress Telerik easy to learn and use.

Try Progress Telerik


About Progress


Progress (NASDAQ: PRGS) offers the leading platform for developing and deploying mission-critical business applications. Progress empowers enterprises and ISVs to build and deliver cognitive-first applications, that harness big data to derive business insights and competitive advantage. Progress offers leading technologies for easily building powerful user interfaces across any type of device, a reliable, scalable and secure backend platform to deploy modern applications, leading data connectivity to all sources, and award-winning predictive analytics that brings the power of machine learning to any organization. Over 1700 independent software vendors, 80,000 enterprise customers, and 2 million developers rely on Progress to power their applications. Learn about Progress at www.progress.com or +1-800-477-6473.


Worldwide Headquarters

Progress, 14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095

On the Web at: www.progress.com

Find us on  facebook.com/progresssw

 twitter.com/progresssw

 youtube.com/progresssw

For regional international office locations and contact information, please go to

www.progress.com/worldwide

Progress and Telerik are registered trademarks of Progress Software Corporation and/or one of its subsidiaries or affiliates in the U.S. and/or other countries. Any other trademarks contained herein are the property of their respective owners.

© 2016 Progress Software Corporation and/or its subsidiaries or affiliates.

All rights reserved.

Rev 16/10 | 160928-0051