Shane Stacy

Ira Woodring

CIS 343-01

Apr 18 2020

C#: A flexible framework language for enterprise, web, and industry

The origins of C# stem from another framework language it shares a lot in common with: Java. Back in the late 90s, Sun Microsystems was hard at work to make Java the foundation of the world-wide-web. Microsoft was keen on investing in the web since the mid-90s and saw how Java was taking off. They decided that they too would use Java, but instead of following Sun's JVM standard reference, Microsoft chose to make their own JVM and its respective language. As a result, the language J++ and the Microsoft JVM were born. As per usual, Microsoft's development tools for their JVM were really good. Microsoft began using their weight in the software industry to push developers to target the Microsoft JVM instead of the reference one provided by Sun. The Microsoft JVM did not conform entirely to the reference implementation, and even added additional APIs not found in the reference JVM. In a way, Microsoft was seen as taking over Java and making it Windows exclusive since the MSJVM was exclusive to Windows and apps written for that JVM would (probably) not work on reference-accurate JVMs. Sun Microsystems sued Microsoft in 1997 for their incompatible implementation as it violated the license for Java. It became clear that Microsoft were not going to be able to do what they wanted to with Java, so Microsoft began investigating making their

own framework. In the late 90s, Microsoft brought on Anders Hejslberg to develop C#, while the Common Language Runtime (CLR) technology seems to have been acquired from Colusa Software. Co-founder of Colusa Steven Lucco continued to work for Microsoft to adapt the technology with Anders. C# and the CLR formed the basis for the .NET initiative where Microsoft was hoping to streamline developing applications for devices of many different software and hardware configurations. In 2002, .NET was released to the world. .NET's principles have changed somewhat since its launch. Originally, .NET was proprietary and closed-source, however over the 5 years the enterprise has been requesting more open-source and community driven solutions to their issues. Microsoft began reworking .NET from the ground-up as an open-source cross-platform framework, named .NET Core. Unfortunately, regular .NET apps are almost never directly compatible and need to be ported to Core. .NET Core is the future of the .NET framework, as it will be succeeding the main .NET line with .NET 5; .NET Core will replace regular .NET by the end of 2020. C# has also dominated the web. C# is the language used for coding in the ASP.NET web-development framework, which is one of the most popular web-frameworks.

C# is an object-oriented language. C# is heavily inspired by Java, although it makes some fundamental design changes from it. Microsoft wanted to correct some of the perceived issues with Java while also allowing Java fluent programmers to easily settle into the framework. These design choices were made to entice their target market, the enterprise, into adopting the language and by extension the .NET framework.

C# provides a huge number of data-types. C# has predefined types such as byte, sbyte, short, ushort, int, uint, long, ulong, float, double, decimal, char (16-bit unicode), bool, string, and

DateTime, but also provides a large number of user definable types such as Objects (generic typing), structs, and enums.

The .NET framework seems to use these names as aliases for its internal representation of data-types. For example, in C# an ‘int’ is the same as an ‘Int32’ and are interchangeable in use. It’s likely these aliases were implemented to make it easier for new developers to start using the language since they could use previous experience from languages with C-like syntax.

Type	Description	Range	Suffix
byte	8-bit unsigned integer	0 to 255	
sbyte	8-bit signed integer	-128 to 127	
short	16-bit signed integer	-32,768 to 32,767	
ushort	16-bit unsigned integer	0 to 65,535	
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647	
uint	32-bit unsigned integer	0 to 4,294,967,295	u
long	64-bit signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	l
ulong	64-bit unsigned integer	0 to 18,446,744,073,709,551,615	ul
float	32-bit Single-precision floating point type	-3.402823e38 to 3.402823e38	f
double	64-bit double-precision floating point type	-1.79769313486232e308 to 1.79769313486232e308	d
decimal	128-bit decimal type for financial and monetary calculations	(+ or -)1.0 x 10e-28 to 7.9 x 10e28	m
char	16-bit single Unicode character	Any valid character, e.g. a,*, \x0058 (hex), or\u0058 (Unicode)	
bool	8-bit logical true/false value	True or False	
object	Base type of all other types.		
string	A sequence of Unicode characters		
DateTime	Represents date and time	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999	

(Table source: https://www.w3schools.com/cs/cs_data_types.asp)

C# is a mostly statically-typed language. There are some methods where dynamic-typing is possible, such as a dynamic variable declaration/definition:

```
dynamic a = 5;
```

C# does most of its type-evaluation at compile-time as a result.

C# is both explicitly and implicitly-typed. C# allows for variables to have their types specified, but it also allows for them to be implied.

```
Explicit:    int a = 5;
Implicit:    var a = 5;
```

As a result, data-types can be implied by the definition. In the above example, we can explicitly say 'a' is a 32-bit integer (int), but we can also use the var keyword, meaning the compiler determines the type depending on how it's value is defined. Since we have defined 'a' to have an integer value, it assigns 'a' an int type (Int32) at compile time.

C# is a strongly typed language. Type coercion is not automatic.

```
int a = "2" * 8;
```

The above example would not compile as the string of "2" is not automatically coerced into an integer-type so the operation could proceed. The "2" would first have to be converted to an integer type.

```
System.Convert.ToInt32("2");
```

The above is an example of a manual coercion.

C# has a huge number of operators. These include member-access, null-conditional, function invocation, indexing, increment, decrement, constructor invocation, type-checking,

overflow-checking, default, name, delegation, size, stack allocation, pointer-member access, not, range, switch, and traditional arithmetic operators.

Operators	Category or name
x.y, x?.y, x?[y], f(x), a[i], x++, x--, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Primary
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false	Unary
x..y	Range
switch	`switch` expression
x * y, x / y, x % y	Multiplicative
x + y, x – y	Additive
x << y, x >> y	Shift
x < y, x > y, x <= y, x >= y, is, as	Relational and type-testing
x == y, x != y	Equality
`x & y`	Boolean logical AND or bitwise logical AND
`x ^ y`	Boolean logical XOR or bitwise logical XOR
`x | y`	Boolean logical OR or bitwise logical OR
x &&& y	Conditional AND
x || y	Conditional OR
x ?? y	Null-coalescing operator
c ? t : f	Conditional operator
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y, x ??= y, =>	Assignment and lambda declaration

(Table source: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/)

While most of these operators are fairly standard in programming, the null conditional operator is one of the most useful and unique operators. It allows programs to more safely use null values and helps prevent out-of-bounds errors. For example, normally when attempting to access an index of a null, it will trigger an out-of-bounds exception. However, with null conditional operators, any attempt to evaluate a null will instead return 'null', meaning the program will not crash and can continue, provided the program is written to handle nulls. This however does not prevent out-of-bounds exceptions when using non-null data-types. Example:

- a?[x]; : If a is null, this expression will evaluate to null. If a is not null, and x is out-of-bounds, an out-of-bounds exception will occur.
- a?.b?[c]; : Short-circuiting. If a is null, expression evaluates to null. b and c are not evaluated.

Operator precedence rules conform to conventional math, meaning the traditional order of operations from mathematics take hold in C#.

- a / (b + c) * d; : Evaluation order is (a, b, c, +, /, d, *)

C# has the following selection constructs: if, if-else, if-else if, nested-if, and switch case. These are all very conventional.

For iteration, C# has a couple different ways to carry it out. Both of them are traditional and are used in many other languages.

C-Style iteration:

```
for (int i = 0; i < size; ++i) {
    Console.WriteLine(i);
}
```

For-each iteration:

```
for (int num in array) {


}
```

C# follows conventional method definitions. The first specifier is the visibility, followed by the return-type, followed by the name, followed by the parameters, and lastly the method code itself. By default, all methods and variables are private in C#. This is one big difference to most other framework languages, which default to public visibility.

```
public bool same(int a, int b) {
        if (a == b) {
                return true;
        }
        return false;
}
```

Parameter passing is very traditional in C#, with some added benefits. By default, parameters are passed by value in C#. This means the parameters' values will be copied into the method's new variables. However, C# also allows for passing parameters by reference. This means the reference to a variable is passed instead of the value. This is very useful when dealing with large or complicated data structures because copying the value(s) of a large structure can impose a significant performance penalty. There are a few different keywords for achieving this. The "ref" keyword is conventional pass-by-reference, meaning the reference is modifiable by the method the parameter is passed to. The "in" keyword means that the reference passed is not

modifiable by the method. For “ref” and “in”, the parameters must be initialized. The “out” keyword means the parameter is passed by reference like “ref”, except the parameter does not need to be initialized. To use “ref ” or “out”, both the calling method and method definition must use the keyword, while “in” only requires it in the method definition.

C# follows conventional scope rules, meaning variables are block-scoped. This means that a variable’s scope is limited to the block it is in. Blocks include namespaces, classes, methods, conditionals, switches, and iterators. Once a variable is out of scope, it can no longer be accessed.

C# has namespace control. This is accessed with the “namespace” keyword. The namespaces are used to organize a project’s classes and also it’s scoping.

C# has exception handling:

```
try {
}
catch (Exception e) {
}
finally {
}
```

The code in the try block is attempted. If it fails, it will proceed to the corresponding catch block depending on the exception type. If the exception is not properly handled, the program exits. Otherwise, the program will proceed to run the finally block after the exception is handled.

C# uses class to define classes. C# does not support multiple class-inheritance. C# supports interfaces. Interfaces can inherit from one or more interfaces, and can have instance methods, properties, events, and indexers. They can also have fields, constants, and static constructors. However, they can't have instance fields, instance constructors, or finalizers. Classes implementing an interface must fully implement an interface, with a minor exception to abstract classes since they can map interfaces onto abstract methods instead of implementing them.

As noted before, C# has private visibility by default. This is in contrast to many other frameworks which instead have public visibility as default. This is done to protect data, as it's much better from a security standpoint to have your software not compile due to a visibility issue rather than have data accidentally modified or stolen because it was left unprotected. This is also likely due to the marketing of C# being focused on the enterprise, where data security and project stability are important.

C# even has pointers! However, they are generally not recommended because they can make code unsafe since it's much easier to make mistakes when working with pointers instead of references. This is why most frameworks don't have pointers. C# only allows pointers in unsafe defined contexts.

C# is implemented as a Just-In-Time (JIT) compiled language. This means the code is compiled as it's run. This is a very common approach in framework languages since it allows a really good compromise between speed and portability, as opposed to pre-compiled languages such as C and C++ which on one hand are a good bit faster usually, but need to be pre-compiled

differently for every platform. Java and ECMAScript (JavaScript) are another couple popular languages that use JIT compilation.

As with most frameworks, the implementation across hardware has a consistency pre-compiled languages can only dream of. Data-types have consistent structure between devices. The number of bits won't change! This is the beauty of framework languages.

In terms of readability, C# is one of the most readable languages out there, assuming you are already familiar with programming. New objects are instantiated with the "new" keyword, which is perhaps verbose but makes sense. Assignment is done with the equals sign, which might seem simple enough but when working with references this can cause problems. Logically, people might think if they set a reference equal to another reference, the data is simply copied from the original object to another. This is not the case for Java, and C# follows the same convention. What this actually does is just assign the new reference to the same object as the original reference. This confusion results in many bugs in programs, but it seems like this design decision makes sense because there's no reason to copy data unnecessarily and use up memory. It means references and how their assignment work is key to reading and understanding C#. I would say references overall help readability, but there's definitely a tradeoff here.

In terms of writability, C# is pretty average. C# is no more to somewhat more verbose than comparable languages. C# tends to suffer from a lot of "boilerplate code", meaning the programmer tends to write repetitive and redundant code just to get a basic class up and running. Coming from Kotlin and Java, C# tries to keep habits in line with Java, while Kotlin offers a far more concise programming language but isn't as easy to pick up and use due to its concise and different syntax. C#'s writability is harmed by its desire to stick close to Java-style syntax, but

the ease of picking up the language was probably considered more important than changing the syntax everyone was used to at the time.

In terms of reliability, C# sticks close to Java, even at points where it doesn't have to due to its improvements. One of the stickling points with Java is string comparison. Instinctively coming from most languages, in Java you'd want to compare strings with the == comparison. However, in Java this doesn't work. In C# however, this DOES work as expected. Java instead compares string values using the string.equals method. C# has this as well, meaning you can do it either way. C# and Java treat object comparisons the same otherwise, meaning comparing object references will only be true if the references refer to the same object, so the Object.Equals method must be used in C# to compare values of the objects, otherwise. In a way, this new inconsistency can be viewed either is an improvement or a step-back in terms of reliability because ideally you want all value comparisons to be done the same way, but C# has added an additional exception breaking this consistency.

C# has some of the best documentation I've ever seen. The .NET Docs are almost all you will ever need. That's why most of my sources are from there.

In conclusion, C# is what you'd expect of an enterprise focused language. It's not very exciting, but it's flexible, familiar, and well documented. It doesn't totally please everyone, but it's good enough where a bunch of people can agree to use it in a project. It doesn't improve on other frameworks significantly, but if it did that it would risk a learning entry cost. I can't say it's my favorite language, but it's in my top 5 for sure.

Calculator Source Code (Written for .NET Core 3.1):

https://drive.google.com/open?id=19aBKHB67P4OpnbdAwFMMf1kH1zNzito2

Sources:

Object.Equals

https://docs.microsoft.com/en-us/dotnet/api/system.object.equals?view=netframework-4.8

String.Equals

https://docs.microsoft.com/en-us/dotnet/api/system.string.equals?view=netframework-4.8

Pointers

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/unsafe-code-pointers/pointer-types

History

https://wiki.c2.com/?HistoryOfCsharp