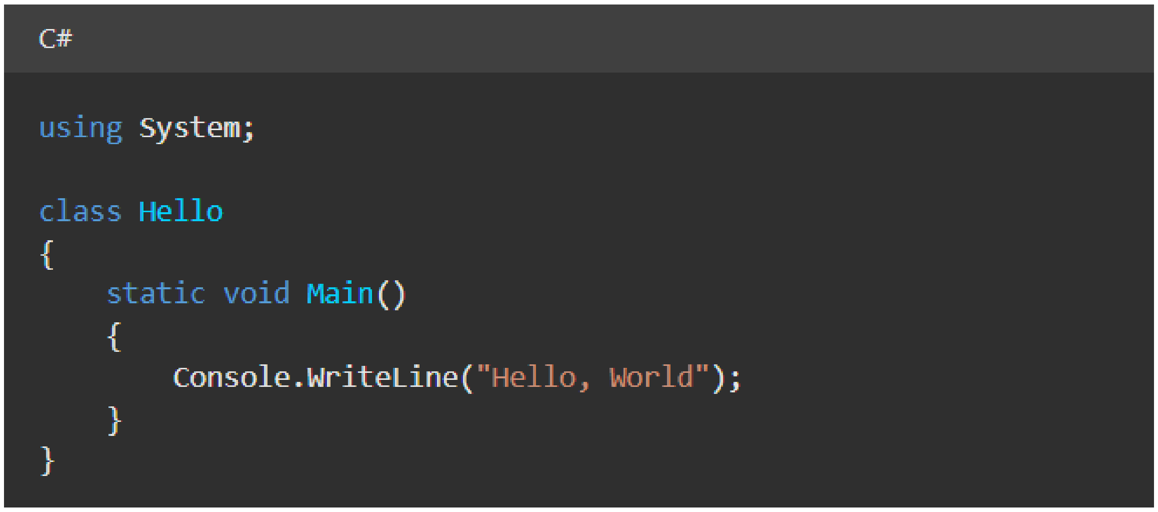
Initial Analysis of C#: History and Implementation

Daniel Kruze

September 9, 2022

CSCI-35500



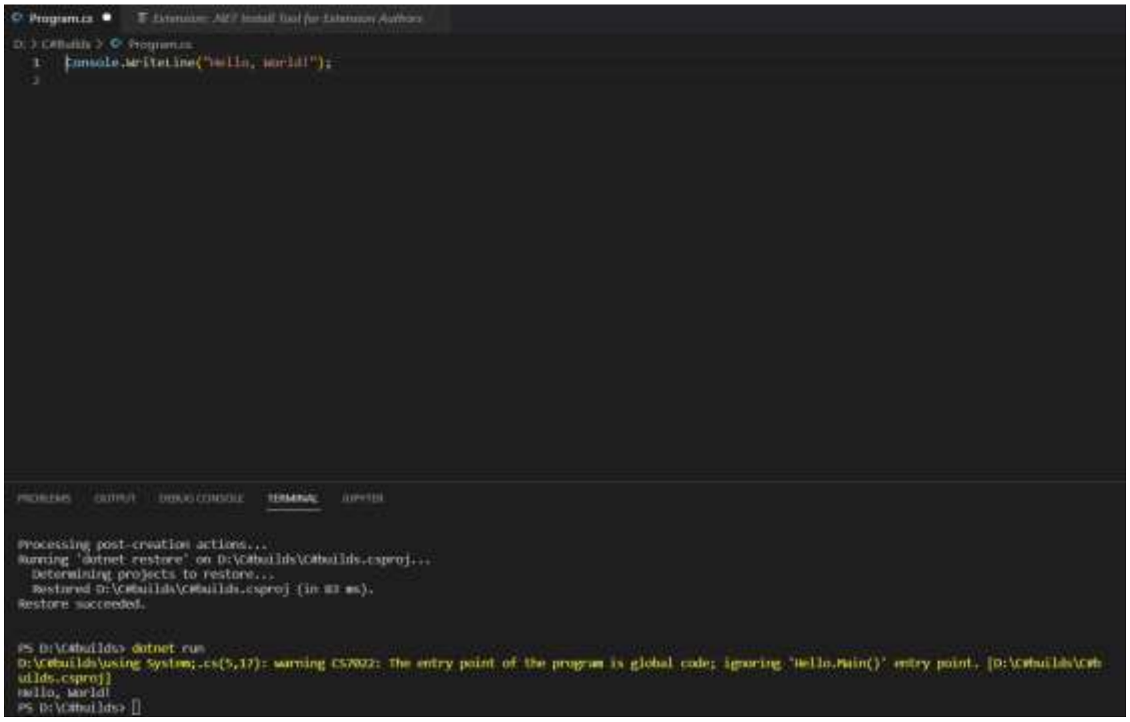
```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

The above code snippet is a glimpse into the content of this report, representing the simplest possible implementation of the C# grammar. This grammar was developed by Anders Hejlsberg around the year 2000 for Microsoft, and it served initially as a way of marrying the central concepts of Java and C++, both of which have always been in heated competition for use in object-oriented environments. The goal was to be simpler than both at once—less verbose than Java, less confusing than C++, wholesale—which it sought to do by being as versatile as possible while maintaining the familiar functions of both languages. From Java, it retained automatic garbage collection, uninitialized variables, and support via libraries for embedded system implementation. From C++, it retained strong typing, overloading, pointers, and quite a few syntactical decisions, although it retained many from Java as well. In fact, many early builds of the language were accused of being "clones" of Java, provoking anger from the developers who quickly sought to distinguish it moving forward.

One primary and famous *difference* from Java is in its approach to typing; originally, the typing attitude was similar to that of C++, where types were very strict and declared with **var** keywords <u>and</u> data types (with brackets for arrays, of course.) This still exists in C#, except with many caveats that even C++ and its peers in strong typing, like TypeScript, *don't* employ for the sake of user-friendliness. For example, Booleans in C# cannot be parsed from other variable types, and comparison statements <u>must</u> use specifically Boolean variables to function properly. In C++, an integer is considered **true** so long as it has a value, allowing for **if** statements in the form of "**if [int] = true {}"** to be valid for compilation—C# necessitates that a programmer <u>only</u> use the **=** operator with Booleans, preventing the common, accidental syntax error in other languages wherein an **=** operator is used where a **==** operator should be; this essentially makes a comparison true or false at all times. This may seem harsh and difficult to work with at first, but the general design philosophy behind C# is that it should use all the best decisions from its parent languages—even the hard to learn ones—in order to make code safe and easy to read.

This attitude is also represented by its approach to memory management, which is another area where its distinctions from Java and C++ are evident. In the former, memory management with pointers is <u>not</u> allowed, and useless pointers to memory (which are created implicitly during compilation) are automatically deleted to prevent negligent memory leaks. In the latter, however, pointers can be <u>manually</u> created and must be <u>manually</u> deleted, allowing dangerous but highly versatile manipulation of data and data structures, such as linked lists or trees. In C#, where the good ideas are implemented in tandem to make multiple approaches viable, pointers *can* be created inside of blocks that are designated **unsafe,** which can be accessed and run from the parent object only with the correct permissions. Most code is **safe** by default, and automatically deleted when it is no longer being used. The decision to implement both decisions, demarcated only by a single keyword, is a testament to C#'s commitment to being different enough to stand alone, but similar enough to be safe, and the juxtaposition thereof is the root of many disagreements surrounding its implementation.

These disagreements seem null when one considers how powerful a language like C# must be for any number of common tasks, but actually trying to write in C# will make its faults very evident. With programming, it is <u>definitely</u> possible to try to have one's cake and eat it too, and C# can often be very difficult to implement or even install for beginners. Take, for example, the case of the "Hello World!" program displayed on the previous page. To run this program requires quite a few steps, and the end result doesn't look anything like one might expect on first glance:


Program.cs
Extension: .NET Install Tool for Extension Authors
D: > C#builds > Program.cs
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
JUPYTER

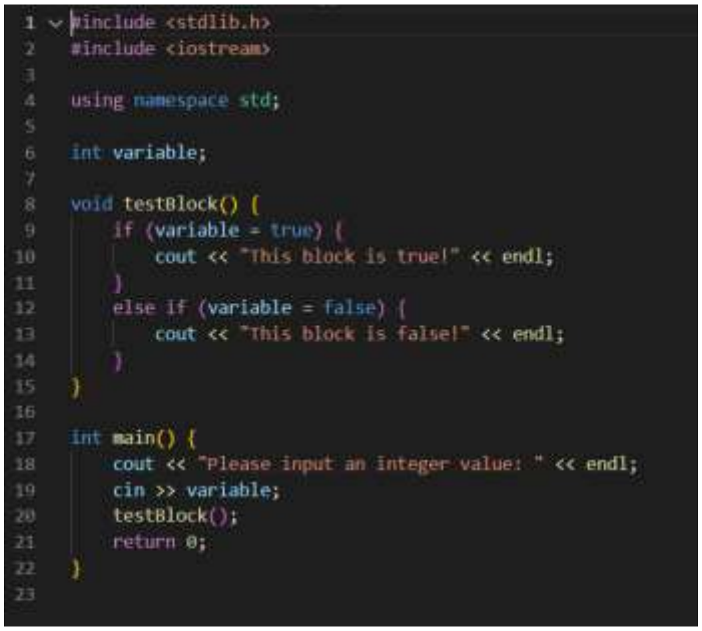
```
Console.WriteLine("Hello, World!");
```

```
Processing post-creation actions...
Running 'dotnet restore' on D:\C#builds\C#builds.csproj...
  Determining projects to restore...
  Restored D:\C#builds\C#builds.csproj (in 83 ms).
Restore succeeded.


PS D:\C#builds> dotnet run
D:\C#builds\using System;.cs(5,17): warning CS7022: The entry point of the program is global code; ignoring 'Hello.Main()' entry point. [D:\C#builds\C#b
uilds.csproj]
Hello, World!
PS D:\C#builds> 
```

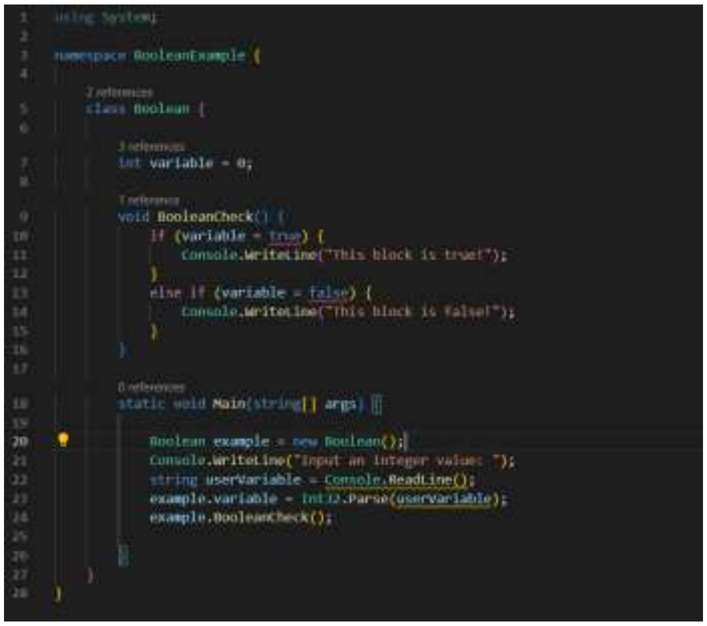
The creation and packaging of this code predicates on the installation of Microsoft's .NET framework and subsequent SDK, as well as the use of Microsoft's own, proprietary IDE called Visual Studio Code. These days, essentially every programmer who isn't a devotee to Unix systems uses Visual Studio. They also generally use a machine that boots, or at least dual boots a compatible operating system prepackaged with the SDK for .NET—not every user knows that, though. Furthermore, not every user falls into that category, and even once the environment has been set up, the code still looks…bizarre. To code in C#, it isn't enough to have the compiler, debugger, and SDK installed, then simply hit the "Run/Debug" button to open a terminal in the preferred directory. Instead, one must navigate to that chosen directory from the terminal and initialize that directory as a .NET application, whereupon the .NET compatibility files for launch and compilation are automatically generated. The programmer is then asked to consent to having their code converted into legitimate instruction—this consent allows some commands to be hidden from the programmer in the **Main** program, giving them the freedom to work within objects using **namespace** shorthand to delineate which methods can be accessed where. This can be very confusing and frustrating at first: the idea that compatibility and ease-of-access features would be so intrusive. But, as one becomes more familiar with coding and debugging inside of .NET apps, it can feel intuitive very quickly.

To fully understand the advantages and disadvantages of C#, some use cases should be demonstrated and analyzed. Four unique features of C# will be explained thusly.

The first feature of C# that ought to be mentioned is one that has already been mentioned in passing, which is the approach to Boolean variables. Take the below code snippets, the first of which is written using C++ and the second of which is written using C#, both to perform the same, simple function to test if a variable is **true** or not:



```cpp
#include <stdlib.h>
#include <iostream>

using namespace std;

int variable;

void testBlock() {
    if (variable = true) {
        cout << "This block is true!" << endl;
    }
    else if (variable = false) {
        cout << "This block is false!" << endl;
    }
}

int main() {
    cout << "Please input an integer value: " << endl;
    cin >> variable;
    testBlock();
    return 0;
}
```

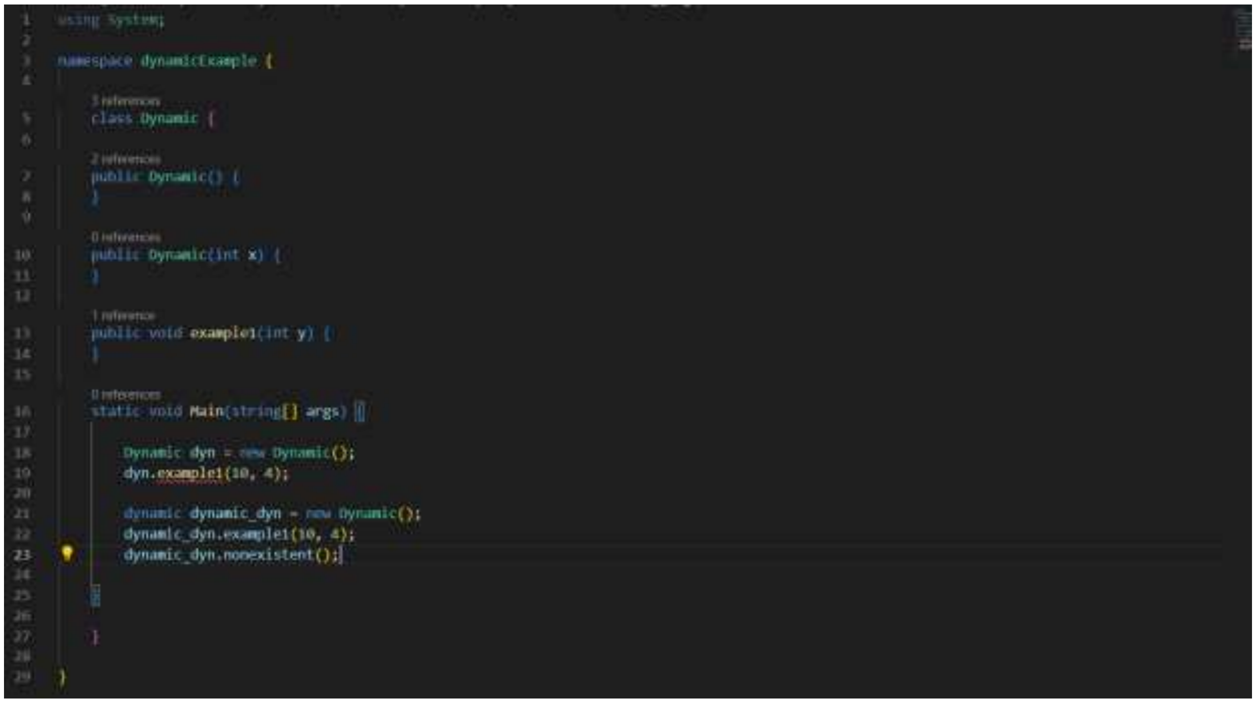


```csharp
using System;

namespace BooleanExample {

    2 references
    class Boolean {

        3 references
        int variable = 0;

        1 reference
        void BooleanCheck() {
            if (variable = true) {
                Console.WriteLine("This block is true!");
            }
            else if (variable = false) {
                Console.WriteLine("This block is false!");
            }
        }

        0 references
        static void Main(string[] args) {

            Boolean example = new Boolean();
            Console.WriteLine("Input an integer value: ");
            string userVariable = Console.ReadLine();
            example.variable = Int32.Parse(userVariable);
            example.BooleanCheck();

        }
    }
}
```

In the first example, the code shows no errors at all, and will in fact compile and prompt the user for input successfully. However, no matter what the user enters, the **else if** block will never be called, because integers are, by default, considered true if they have a value. To make this run properly, one would need to use the **==** operator, and the false block would be called when **variable** is set to 0. In the second example, notice that the **true** and **false** values are underlined in red, indicating a compiler error (Visual Studio Code often detects errors prior to compilation.) This is because C# does not allow the programmer to stumble into this error, and forces them to use the correct operators or correct variable types where necessary—a helpful feature for newer programmers, or older programmers who have become complacent. Note that in the C# example, the user's input is parsed to ensure that whatever they type is treated as an integer; this can also be done in C++, but this example code was not written with the intent of catching type errors. That check was only included to reduce the amount of irrelevant compiler errors displayed in the example snippet.

Of course, this example is simple and was already partially discussed earlier. It does, though, provide a basis for discussing data typing on the larger, more ambiguous scale. This is another place where C# shows distinction, having support not only for strong typing, but for simple and fluid type casting. There are a couple ways of doing this, most commonly by using the **var** keyword to define a variable; when this is done, the data type of a variable doesn't need to be explicitly defined, nor does the value need to be initialized in a constructor (if that variable is a property of a class.) This isn't necessarily unique to C#, however, and many languages like Python or JavaScript are notorious for quietly treating all variable definitions this way, allowing data types to be parsed on the fly without invoking explicit parsing methods (see the **System.Int32.Parse()** method invoked in the previous C# snippet.) What *is* unique in C# though is its use of the **dynamic** keyword, which is only vaguely similar to the **transient** equivalent in Java. By defining a variable as **dynamic**, not only is the data type for the variable not explicitly defined, but unlike the **var** keyword it will never *become* explicitly defined, and remains mutable at all times. With the **var** keyword, once a data type is assigned to the variable after it has been initialized or altered, it is treated as that data type and only that data type as the compiler moves forward.

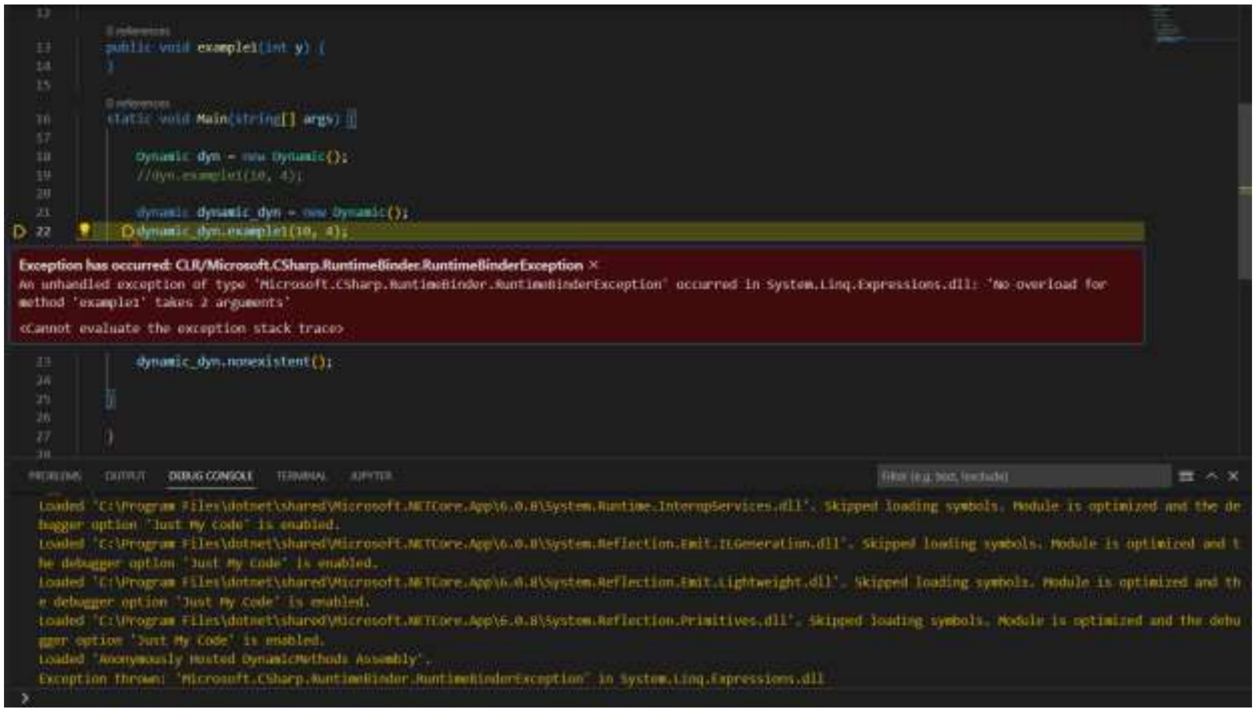
Dynamic variables do not have this restriction and can thusly be very helpful for any case where data types are expected to be fluid, like with user input. This also makes them very dangerous, however, which is what the below code snippet serves to illustrate. The **dynamic** keyword, in addition to being used in variable definition, can also be used for objects, as objects are considered data types, making it doubly dangerous during compilation. The following code utilizes this property, and invokes example methods with both a **dynamic** and a default, **static** instance of the example object:



```
using System;

namespace dynamicExample {

    3 references
    class Dynamic {

    2 references
    public Dynamic() {
    }

    0 references
    public Dynamic(int x) {
    }

    1 reference
    public void example1(int y) {
    }

    0 references
    static void Main(string[] args) {

        Dynamic dyn = new Dynamic();
        dyn.example1(10, 4);

        dynamic dynamic_dyn = new Dynamic();
        dynamic_dyn.example1(10, 4);
        dynamic_dyn.nonexistent();

    }


    }

}
```

This code creates an object called **Dynamic**, initializes it with an integer property **x**, and provides one method in the object, **example1(),** which is passed a parameter **y.** In the main method of the program, firstly a **static** instance is created of this object, and the single method is invoked and passed two parameters. Obviously, this—as the snippet shows—results in a compiler error, because the object only has one property and the method can only be passed one parameter. This follows as one would expect, but the second instance behaves differently due to the **dynamic** keyword. For that instance, the method is called again, also with an invalid amount of parameters: there is no compiler error here. Additionally, a method that <u>does not exist</u> is called for the **dynamic** object, and similarly does <u>not</u> cause any compiler errors. Why is this?

The C# compiler accepts the **dynamic** object as one with not explicit data type, and one that can have data types assigned to it on the fly. Therefore, the restriction that the object have the same number of parameters as is explicitly defined above is <u>not</u> applied, and the object can be compiled under the pretense that it will have a type assigned to it when necessary. This also explains why the nonexistent method is allowed to be “called,” because for all intents and purposes, the method could very well exist if defined elsewhere. In practice, however, there are obviously problems with this approach—how could this code execute at runtime? The methods are in one case entirely invalid and in another simply not real, so what would this code *do*? The answer is…nothing. These exceptions actually *can* be caught at runtime by the C# debugger:



```
        0 references
        public void example1(int y) {
        }

        0 references
        static void Main(string[] args) {

            Dynamic dyn = new Dynamic();
            //dyn.example1(10, 4);

            dynamic dynamic_dyn = new Dynamic();
            dynamic_dyn.example1(10, 4);
```

```
Exception has occurred: CLR/Microsoft.CSharp.RuntimeBinder.RuntimeBinderException
An unhandled exception of type 'Microsoft.CSharp.RuntimeBinder.RuntimeBinderException' occurred in System.Linq.Expressions.dll: 'No overload for
method 'example1' takes 2 arguments'
<Cannot evaluate the exception stack trace>
```

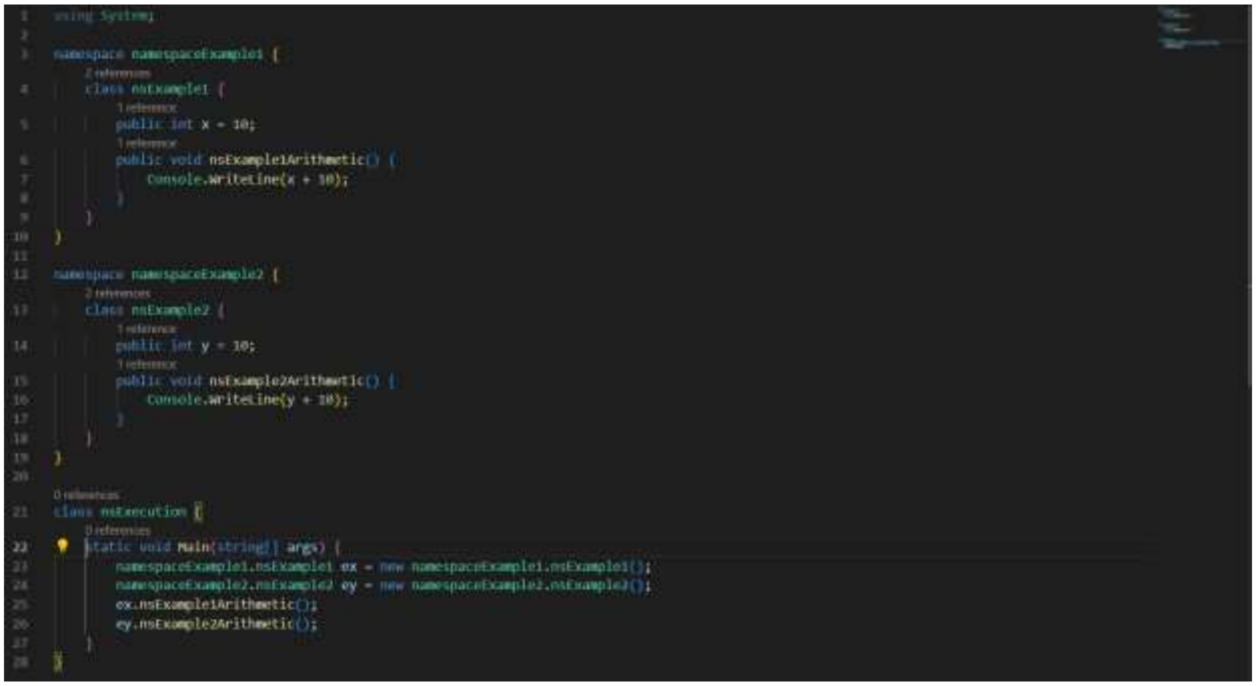
```
            dynamic_dyn.nonexistent();

        }

    }
```

```
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Runtime.InteropServices.dll'. Skipped loading symbols. Module is optimized and the de
bugger option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Reflection.Emit.ILGeneration.dll'. Skipped loading symbols. Module is optimized and t
he debugger option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Reflection.Emit.Lightweight.dll'. Skipped loading symbols. Module is optimized and th
e debugger option 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Reflection.Primitives.dll'. Skipped loading symbols. Module is optimized and the debu
gger option 'Just My Code' is enabled.
Loaded 'Anonymously Hosted DynamicMethods Assembly'.
Exception thrown: 'Microsoft.CSharp.RuntimeBinder.RuntimeBinderException' in System.Linq.Expressions.dll
```

What does this imply for the use of the **dynamic** keyword? It implies that it should be used very carefully and only when the programmer has complete, assured control over the data types to be used, and is generally better for defining and parsing variables than for objects. The use of the keyword for objects makes methods very difficult to use and define when needed, making them appear safe to use…only to cause bizarre errors at runtime. It is unique for C# not only because it can be used for objects but also because it is one of very few features of the language that allow the programmer to “shoot themselves in the foot,” meaning that they could ruin their own code without knowing why. Despite this, their use is still recommended for dealing with user input, albeit very carefully.

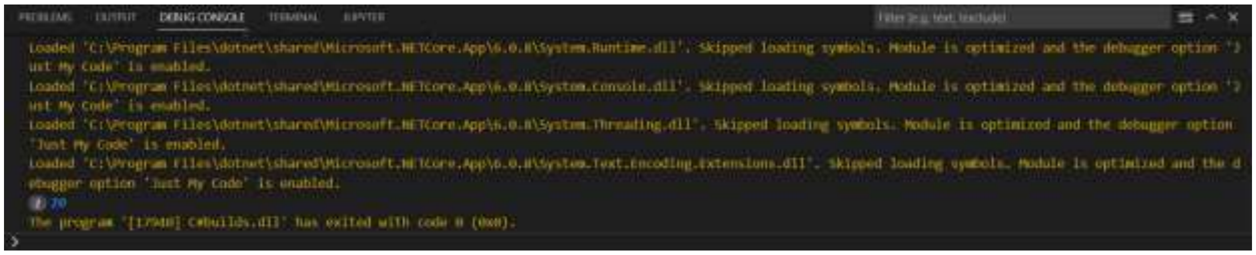
Not all features are intended for precise, dangerous operations with data, however, because most features unique to C# are intended to make code safer and more readable, as is the “point,” as it were, of the language as a whole. The **namespace** keyword is just that—a tool to make code more readable—and it’s implemented much more smoothly than in C++, where it originated.

Namespaces in C# are not simply ways for object files to be packaged separately for compilation via makefile, like they are in C++; neither are they just convenient blocks of classes bundled together to be imported at runtime and taken from like libraries, as the Java equivalent, **packages,** do. While the C# **namespace** can certainly serve these functions and is often used to do so, they are more versatile than either of their counterparts in that they can be defined multiple places in one file and for multiple objects, essentially whenever one wants. What they do is group specific classes together (like Java) so that they can be invoked specifically by other classes in other files at will without changing the type or permissions of the class instance doing the invoking (like C++.) Take the below code as a simple example, because it shows multiple namespaces with unique classes being created in the same file and invoked in that same file by a different class. Note that the optional whitespace is <u>not</u> included, in order to make the code fit this document better:



```
using System;

namespace namespaceExample1 {
    2 references
    class nsExample1 {
        1 reference
        public int x = 10;
        1 reference
        public void nsExample1Arithmetic() {
            Console.WriteLine(x + 10);
        }
    }
}

namespace namespaceExample2 {
    2 references
    class nsExample2 {
        1 reference
        public int y = 10;
        1 reference
        public void nsExample2Arithmetic() {
            Console.WriteLine(y + 10);
        }
    }
}

0 references
class nsExecution {
    0 references
    static void Main(string[] args) {
        namespaceExample1.nsExample1 ex = new namespaceExample1.nsExample1();
        namespaceExample2.nsExample2 ey = new namespaceExample2.nsExample2();
        ex.nsExample1Arithmetic();
        ey.nsExample2Arithmetic();
    }
}
```

Also, note that not only are multiple **namespaces** being created, invoked, and successfully compiled, but these **namespaces** (as well as the class containing methods to invoke them, which belongs to no particular **namespace**,) are **using** the same external **namespace**, **System**, to invoke methods as well. This function doesn't do much; it performs two arithmetic operations and prints them to the command line, as follows:



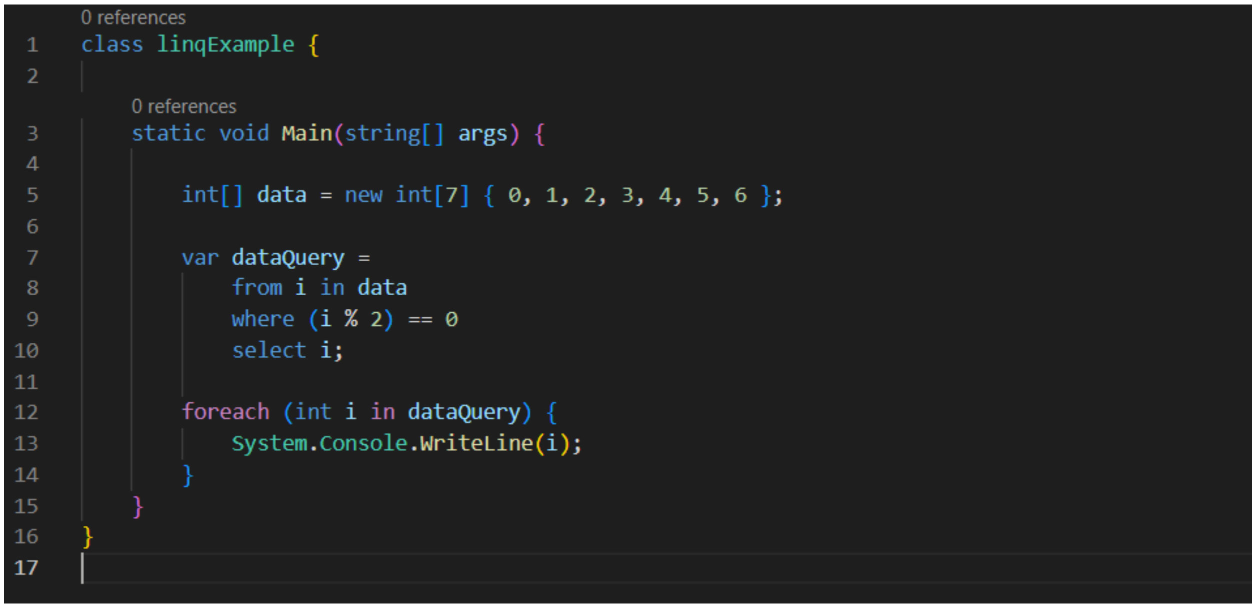
```
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Runtime.dll'. Skipped loading symbols. Module is optimized and the debugger option 'J
ust My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Console.dll'. Skipped loading symbols. Module is optimized and the debugger option 'J
ust My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Threading.dll'. Skipped loading symbols. Module is optimized and the debugger option
'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Text.Encoding.Extensions.dll'. Skipped loading symbols. Module is optimized and the d
ebugger option 'Just My Code' is enabled.
2 20
The program '[17940] C#builds.dll' has exited with code 0 (0x0).
>
```

What this code exemplifies, though, is that **namespaces** can be as general or as specific as one likes, for any number of reasons. The two example **namespaces** here are simply made to be invoked by a third class so that their methods can be specifically accessed. This is done by creating instances of those classes with the **namespaces** beforehand, specifying that only those classes within that **namespace** are to be used. In spite of this, the **System namespace** is invoked globally, so that all subsequent **namespaces** (and classes, by extension,) can use its methods at will. This **namespace** is <u>not</u> referenced by name with each instantiation, though, as none of the methods defined in this file conflict with any of the methods contained in the **System** namespace, meaning that it doesn't need to specified to avoid invoking the wrong method at runtime.

This may seem confusing at first, because it's hard to tell when one should specifically include the name of the **namespace** before the method, and when one doesn't need to, saving space by not doing so. The answer to this is simple; the name of the specified **namespace** only *needs* to be included during instantiation, or if two namespaces contain methods of the same name (typically in programs with many different files.) Note that in the example snippet, the two objects are instantiated, and then their methods are called without including the **namespace** by name each time—this is because the methods have distinct names that, if typed carefully, will never conflict with each other. This is the exact same reasoning by which the **System** methods do <u>not</u> need to be specified in the form of **System.[method]().** This means two things for the concept of **namespaces** in C#; first, one must be careful with what classes they package inside of **namespaces** and what methods those classes include, in order to ensure that the correct methods are called at runtime. Second, **namespaces** are critically important for projects of large volume, and can compartmentalize any size of project into small, easy to use blocks.

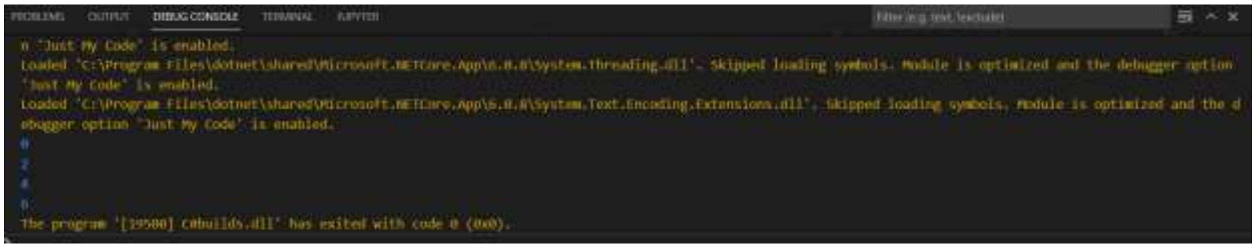
So far though, these unique features appear to be riffs on previous features in existing languages, altered to make them more versatile for the large amount of different projects coming through the .NET framework. It seems simple that C# would dedicate itself to that concept, being that it was made not as an introductory language but as a powerful "hybrid" of sorts, designed to make cross-platform projects with large teams much safer. This is the thesis of C#, yes, but C# happens to have more tricks up its sleeve than just that, and in fact it has divergently evolved some *very* interesting features that make it well-suited to highly specialized situations, rather than general ones.

The integration of the **LINQ** library is the most fascinating and useful of these features, boasting an enormous utility unique to Microsoft technologies that most other languages simply do <u>not</u> support. <u>**LINQ**</u>, or <u>L</u>anguage <u>IN</u>tegrated <u>Q</u>uery is the collective name for a set of compatibility files that allows code in the C#, .NET environment to perform queries of various and sundry databases directly inside of C# programs. What does this mean? It means that a C# program could include a method that *literally* runs code for SQL or XML documents, among others, to query an external database, or treat data in a C# file as a member of a database. For example, one could make an array or vector inside of a C# class and query it, as if it were a column in a database, thereby removing the need for the inclusion of separate libraries or **namespaces** with prebuilt operations for sorting or indexing in an array. An example of this is included in the following snippet of C# code:



```
0 references
class linqExample {

    0 references
    static void Main(string[] args) {

        int[] data = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        var dataQuery =
            from i in data
            where (i % 2) == 0
            select i;

        foreach (int i in dataQuery) {
            System.Console.WriteLine(i);
        }
    }
}

```

Note that this code does <u>not</u> use the **System namespace**, yet still performs operations on arrays. This is the power of the **LINQ** integration, to allow simple manipulation of data using syntax that is not only easy to read (and more familiar to people who are used to SQL,) but also easy to write without dependencies, as the data source, query, and output are all executed from a single class in a single, static method without a **namespace.** Note also that the lines in the query (**var dataQuery,**) are <u>not</u> delineated using semicolons—this is because this code is, quite literally, SQL code that is simply being executed by a C# program, making it exempt form some convetions of C#. The output to the command line is, then, exactly as one might expect in any equivalent SQL environment:



```
n 'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.0\System.Threading.dll'. Skipped loading symbols. Module is optimized and the debugger option
'Just My Code' is enabled.
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.0\System.Text.Encoding.Extensions.dll'. Skipped loading symbols. Module is optimized and the d
ebugger option 'Just My Code' is enabled.
0
2
4
6
The program '[19500] C#builds.dll' has exited with code 0 (0x0).
```

The array is treated as a member of a database, queried according to which values are even, and then each value the query selects is displayed to the command line by a simple **foreach** command. It is worth noting here that the **foreach** keyword is only accessible via libraries in Java and C++, adding another layer of speed and versatility to C# in comparison.

Succinctly, this example shows that C# not only has the ability to treat data structures as members of databases, or that it can read members of external databases (which the example does not directly illustrate,) but also that operations in *other languages* can be performed on those data completely internally, compiling then executing correctly. This allows a user of C# from any background to easily create simple data structures (of variable type, as shown earlier in this report,) then sort and organize them in ways that they may be more familiar with, or that may simply make the code more readable. This is a feature that *truly* no other language can advertise, and while it may not seem like the most useful or important feature, it makes C# a contender with database friendly languages like PYTHON qua compatibility, and not simply a safer replacement for object-oriented languages.

Overall, C# clearly has a host of features that make it complicated to learn, but incredibly widely applicable and, in some cases very specialized depending on the task at hand. How, then, does it really "rate" as a programming language? Is it useful? Is it powerful? Is it "good?" To adjudicate this, a set of criteria need to be agreed upon before evaluation, and the given examples (as well as the general syntax) should be weighed according to those criteria. These criteria, for the purposes of this report, are as follows: Typing, expressiveness, simplicity, exception handling, and soft metrics. Respectively: How reliable and usable are data types, how easily can a vision for a program be executed, how simple is the language itself to learn and implement, how safe is the language for compilation and runtime, and in what ways does the language appeal outside of its base functions.

…

TYPING:

Typing is obviously very central to C#, being the majority of the motivation for its existence. As has already been extensively discussed, C++ and Java's collective shortcomings with regards to control over data type fluidity were a major problem that C# sought to solve in its mission to overcome both languages; did it succeed?

In C#, data types can clearly be typed very strong, typed very weak, or not typed at all, entirely dependent on the programmer's own familiarity with the file structure of their .NET application. This makes C#'s approach similar to C++, which is certainly a good thing because strong types make for far fewer user errors, while the ability to make weak types at will allows users' input to be manipulated. Where this approach to typing—which is already very safe and efficient—*exceeds* its influences is in the ability to make **dynamic** types, which have already been thoroughly analyzed. This ability to make variables and even objects (classes and their methods) entirely fluid in the eyes of the compiler makes editing code easier, as well as just writing it in the beginning. This is because dynamic methods, even when mishandled, will compile regardless and simply serve no function, but at runtime will be specifically pointed out by the debugger; this, naturally, can save a lot of time for a programmer who is used to C++. Gone now are the woes of a single, unmarked segmentation fault ruining an entire application and not having the decency to make itself known for editing. A proper use of C#'s many approaches to non-static typing will allow the debugger to do a lot of work by itself, and keep a program save even after a successful compile. Be warned, though, because as it was implied in the above discussion, the use of **dynamic** types in excess could make code incredibly dangerous, so the learning curve for this keyword is to be respected prior to implementation.

In short, the typing in C# is very strong and just about as good as it may have seemed during the earlier discussion. Types can be anything a programmer wants but also anything a user needs, and it only necessitates careful planning from the outset.

EXPRESSIVENESS:

In terms of expressiveness, C# feels like it suffers in some cases. The learning curve is surprisingly steep, relying on prior experience with object-oriented concepts to really use effectively. It also relies on the use of the .NET framework, which isn't necessarily easy for everyone on every platform to use. As such, it isn't necessarily anybody's first choice for "vision," unless that vision involves technical, goal-oriented code that handles large volumes of data or data structures. In spite of this, there are some fields in which data structures and vision coincide, and in those fields C# is actually more popular than one might believe.

One such place is in video game engines, which is surprisingly the most popular and common place of implementation for C# outside of Microsoft itself. The Unity and Godot engines are both free and open-source engines for games of any size and scope that utilize C# (Unity, exclusively so) for their creation kits. The majority of "indie" games on the market for the last decade or so have been made using these engines due to their free status, despite the fact that they can feel very difficult to use due to C#'s difficulty in learning. Is it worth it in terms of expressiveness, however? Indeed, the popularity of C#-driven game engines has not declined at all since their inception, and as anyone who plays a lot of video games will say, new and interesting ideas are always popping up everywhere using these engines. It seems like C# can be used to make anything one can imagine for a video game if one is willing to discipline themselves in the use of C#, not to mention install all the .NET compatible frameworks for testing their game prior to actual release.

Creativity, yes—it is indeed possible to be *very* expressive with C#, with video games being arguably the most technically complicated and difficult to maintain programs in the consumer sphere, especially considering the typically small size of video game dev teams. This is not, however, the only point in C#'s favor; just because one can be creative with C# doesn't necessarily mean that C# is *good* at being expressive. After all, a programmer with enough time could be creative with anything, so C# needs to make itself more conducive to expressivity than other alternatives.

A good way that it manages this is the way it integrates enormous quantities of libraries to perform GUI-related tasks that other languages make *very* difficult. The **System** library, for example, isn't only useful as a way to print and read user data in a **namespace;** there is also support in this library for image manipulation, such as the **System.drawing** library that can output images to a GUI in a single method. Or, one could take the **System.windows** library that similarly allows input reading from users in very short, single method calls. Are there limitations to this simplicity? Certainly, because the use of Windows compatible GUIs has been on the decline for a long time, considering most Microsoft products necessitate the use of the app store or are simply installed from a web browser. Furthermore, on non-Microsoft platforms, or in any environment where .NET applications cannot be implemented these features may seem completely vestigial, much like Java's GUI libraries that now see next to no use at all. What does this mean for C#'s expressiveness? It means that C#, while it has some unique advantages in GUI application, these advantages may not be all that significant these days.

And, in general, that is the tragic tale of C#'s expressiveness. For any large, object-oriented project that seeks to represent something creative (essentially leaving 3D-rendering and video games as the sole representatives of this demographic,) C# can be incredibly thorough and powerful, and as such is used very often. But, on the large scale, C#'s strengths simply aren't played to very much, leaving it sort of a waste of time to learn for small-scale projects; nobody would be trying to configure C# to make, say, webapps or graphic design software.

SIMPLICITY:

Clearly C# is not a simple language. Experience, time, and caution are absolutely essential for any effective C# program, and the use of all the proprietary Microsoft technology for the C# runtime environment makes it difficult to even set up, *especially* on non-Windows machines. In a way, then, its simplicity depends upon the skill of the programmer, much like C++; the difference is that C++ is useful for any skill level because its obstinance can be an incredible teaching tool, while C# can really only be useful for the skill level where a programmer would rather jump off a bridge than keep using C++.

C# is a language for intermediates, certainly, because it is only distinct from languages like it when the programmer understands the shortcomings of its ancestors, and furthermore is in a situation where they need to write a robust enough program where those shortcomings become significant. For example, an introductory programmer who has never written a multi-file program might drive themselves insane trying to install, learn, and implement C# for a project involving only a few classes and simple user input reading. There would be no reason not to use Java if all one needed were to store a few properties and run a few methods to create output, and indeed that is how most people (including the author of this report) learned to program. But, if someone with experience were tasked to write a large program involving multiple people's code, perhaps an application for reformatting a removable disk on a Windows machine or anything similar, it would behoove them to learn C#. Where C# may seem like overkill to learn for simple tasks, it is comparatively much more readable and flexible than, say, Java for handling tasks with so many variables and so little input to be stored.

This can be seen at this general level, yes, but on the specific level it shows in many ways. Imagine a program in Java that contained multiple apps with **Main()** functions in every one; that would be quite a few **public static void main(string[] args)** that only continue multiplying as the size of the program increases. In C#, however, a main function can simply be denoted **static void Main()**, if a programmer should like to keep it flexible and simple. Even compared to languages C# is less similar to but still contends with, like database languages (keep the **LINQ** integration in mind,) there is an increase in simplicity to be found. Take Oracle PL/SQL, wherein a line printed to the console is denoted **DBMS_OUTPUT.PUT_LINE(),** which is an abominable mouthful of a method. In C#, to perform the same operation for a query, the method is simply **Console.WriteLine()** like any other print command in **namespace system.**

In short, C# is much simpler than many of its counterparts at the intermediate level of programming, where more conventional languages become too verbose to be viable for careful planning or revision. For a beginner, though, C# can seem very obtuse to try to learn due to its insistence on safety and library usage, which is a concept that many of the above examples with code snippets mention in passing.

<u>EXCEPTION HANDLING:</u>

How <u>does</u> C# handle exceptions? For the most part, it handles them with the packaged debugger that will point out any and all errors at runtime. The benefit of necessitating the proprietary IDE and technologies for running the code is that there isa swath of failsafes that prevent code from “breaking” prior to running. Much like C++, C# won’t even compile is syntax errors have not all been pointed out and fixed (aided by the code highlighting and checking central to Visual Studio Code,) by the programmer. Say, though, that a program has methods that compile and run, but may cause errors in tandem at runtime dependent on user input. How does C# handle this common situation? Or, what if data is to be streamed in and out of a device by a Windows machine, and the device could be considered corrupt or invalid for use? How would an error be thrown by a C# program?

The answer is the same as any other popular language: **try/catch** blocks. Specifically, C# has just as many different methods for catching errors at runtime without aborting the program as any language that it may compete with. This could be inferred from C#’s commitment to safety, as it would be confusing to put so much pressure on the debugger to catch everything and just kill any program that might throw an error, considering how safe C# code is intended to be. The debugger and compiler simply prevent negligence on behalf of the programmer, but it is the error throwing and catching blocks that actually ensure that input is valid. While it may not be as versatile as, say, JavaScript with all its **promises** and **await** clauses, C# is just about as good at handling exceptions as Java.

One key difference, though, is that **try/catch** blocks have an extra keyword in them that may be useful for large **try/catch** blocks, and that is the **finally** clause. This clause can be included after a **try** or a **try/catch** block to release the variables and resources created in the **try** clause, allowing for a very thorough **try** clause that could use any number of methods, even defined within the clause itself. Java also has this **finally** clause, but unlike in C# this clause is not portable; running Java via virtual machine can cause the **finally** clause to be skipped over for any number of reasons during execution of the clause to be skipped over for any number of reasons during execution of the **try/catch** block, making it very inconsistent. In C#, this problem does not apply as programs in C# are automatically packaged in .NET applications, and wouldn't necessarily encounter such issues with virtual machines outside of hardware difficulties.

This is, however, presumptive in a way, so for all intents and purposes exception handling in C# is very similar to Java's: very reliable. The debugger is incredibly thorough and will often prevent exceptions before they can even occur, and the stubborn compiler (much like C and C++) will only function when those exceptions have been nipped in the bud. Exceptions at runtime can be handled just like business as usual in Java programs, making it not only reasonable and effective, but one of the easier facets of the language to learn for those moving over from Java.

SOFT METRICS:

Not every function of C# is attractive because of its actual use, though, and there are many metrics by which it can be judged more subjectively.

Firstly, why use C# if nobody else does? After all, learning a programming language is a skill, and that skill is honed with the intention of marketing in a world as fast-paced and unreliable as technology work. Does learning C# give somebody any common ground with coworkers or employers? As it turns out, it does, for a couple of reason. C# is commonly used for implementing video games and 3D-rendering tools, which are very popular among tech hobbyists. At the very least, this could give programmers with common experience in C# the ability to make marketable consumer programs, or at least the ability to use existing ones in tandem. Further still, C# is the language of the .NET framework, the most common framework for Microsoft product applications; as Microsoft is one of the most popular companies in the tech world right now, everyone seems to use their products at one point or another. Being familiar with the language of the most popular applications for one of the most popular companies would put any programmer in a position to keep up during job interviews, and further knowledge of the Microsoft environment (outlook, teams, office, etc.) is practically *required* for tech work in America. As a whole, learning C# could help a programmer market themselves for jobs and connect with other programmers to create either functional or recreational programs, making it a helpful bandwagon to jump on.

Secondly, what if other languages are simply unfit or unappealing for similar tasks? It's no secret that the majority of programmers in informatics or data science *vastly* dislike languages like C++ because of their standoffishness. A language with garbage collection, flexible typing, GUI support, and easy debugging is much more appealing for large, object-oriented tasks when the alternative is a language with a 15-

miillion line compiler and an insistence on manual pointers, *especially* when that language throws segmentation faults every time a tree exceeds 16 values. Not only that, but Java is often skewered in popular use for the many crow's feet it displays, carrying with it everywhere it goes the most verbose set of keywords and commands of any modern language. Java also happens to have difficulty with serialization, necessitating that user input be saved using **transient** keywords and complex data structures stored in separate, inherited classes in whole other files. Java is strong, but unintuitive, and C# offers an alternative to the verbosity with **namespace** shorthand and **dynamic** data, among other things. In any case, C# is simply less annoying than its two parents, and for that it is commonly praised for its utility in large projects.

Lastly, is C# a likeable language? People who use it generally seem to think so, and after a careful analysis, I, the author, have come to appreciate many of its features, despite my frustrations setting up environment variables to make it run on my Visual Studio installation without whining. It's likeable because it solves a lot of problems but manages to feel familiar all the same, granting a breath of fresh air to a programmer who is used to "putting up" with many of the specifics of the older, less well thought-out languages. Things like Java or C++ were created to allow code to operate where it couldn't before, but C# was created to allow code to operate *better* where it was simply *possible* before.

…

In the end, then C# could be considered *incredibly* strong, but difficult to use for beginners, and fraught with compatibility issues that make it a pain to set up, despite its incredible flexibility. When, then, is definitely a *bad* idea to use C#? If it's so strong and versatile, why wouldn't somebody use it whenever they have object-oriented goals? For a start, C# can only be used in the context of the .NET framework. Writing in C# using a Unix machine can be very difficult, and for any application involving a single Unix machine that C# might seem applicable for, C++ is definitely going to be a better option for consistent compilation and execution, despite its archaic and challenging syntax. It also seems that C# would be exceedingly unhelpful for functional programs like compilers or parser generators because of its insistence on using objects as the basis for all of its functions. Functional programs would be made infinitely more confusing and difficult to edit with the use of C#, but programs like games or databases would not be. This is the basis for the use of C#: creating applications with objects and object-like data. This means that as a whole, C# struggles whenever a program does not predicate on the use of those concepts.

Conclusively, then, C# is an ideal language for large-scale, object-oriented projects that manipulate large amounts of data, *especially* with user input, and any system that includes support for the Microsoft products on which C# was built would benefit greatly from its use. As a language, C# is not only familiar enough for an intermediate or advanced programmer to get a handle on, but powerful enough to warrant a good look from a beginner, despite some of its shortcomings and its troublesome applications in the creative fields. C# is, for sure, the worthy successor to Java and C++ that it sets out to be, complete with definite strengths and weaknesses.

REFERENCES

- "C# Documentation." *C# Docs - Get Started, Tutorials, Reference. | Microsoft Docs*, Microsoft Corporation, https://docs.microsoft.com/en-us/dotnet/csharp/.
- "C# Tutorial." *C# Tutorial (C Sharp)*, Refsnes Data, https://www.w3schools.com/cs/.
- "ECMA-334." *Ecma International*, Ecma International, 10 Aug. 2022, https://www.ecma-international.org/publications-and-standards/standards/ecma-334/.
- North, Chris. "User Interface Programming in C#." Virginia Polytechnic Institute and State University, 2007.