An Evaluation of the Programming Language C#

R.J. Hamilton

CIS 343 – 01

December 9, 2012

Introduction

C# is a high level, object-oriented programming language that has become very popular in the past 10 years, currently ranking in at number six on the Tiobe Index. The programming language C# was developed by Microsoft in July of the year 2000 at a Professional Developers Conference as part of their larger initiative called .NET (Horn 2018). The name C# was used by a variant programming language that was based on C and designed for incremental compilations. The sharp or "#" in the name was chosen because it embodies the concept of a sharp note from music were the sharp note is higher than the normal note. In this case, C# is higher than C. The hashtag or "#' was also thought to be a step up from C++ because if you added two more addition symbols above the "C++" then you would have "C#" (Kovacs 2007).

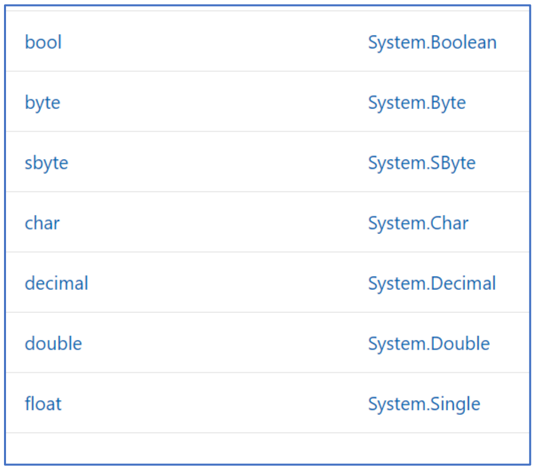
The goal for C# was to create a language that was modeled to look similar to JAVA but also managed to contain some of the functionality of C++. C# was designed to be a general purpose, multi-paradigm programming language that encompassed the following programming disciplines and characteristics (ECMA 2006):

- strong typing
- imperative
- declarative
- functional
- generic
- object-oriented (class-based)
- component-oriented

C# was intended to be a simple, modern, general-purpose, object-oriented language (ECMA 2006). Some of the design goals stated in the ECMA indicate that C# was intended to be used to develop software components in distributed environments. The domain where C# was designed to be used was hosted and embedded systems, covering everything from operating systems to small, simple functions. Even though C# was designed to have the capacity to design operating systems, it is noted that C# was not intended to be a programming language that would be able to compete with the performance and size of C or assembly language. The language C# is primarily used today to develop mobile apps (specifically Android) and in game development (Mkhitaryan 2017).

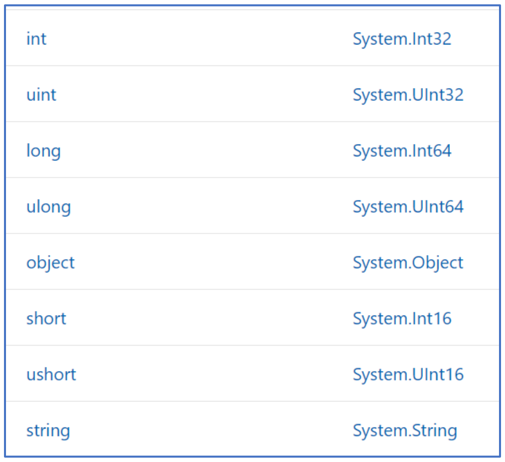
Data Abstractions

C# supports strongly-typed implicit variable declarations. The following images portray a graphical representation of the different data types that are provided in C#:



bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single

Figure 1. Data Types



int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

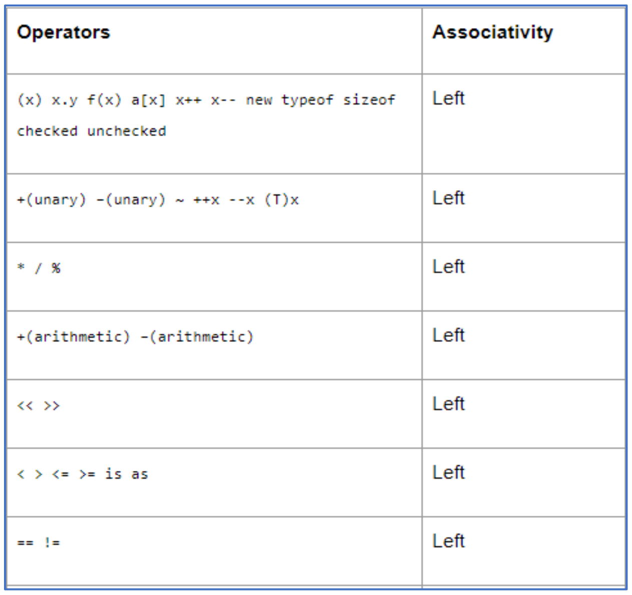
Figure 2. Data Types (cont.)

These lists were taken from the documentation provided by Microsoft for C#. The type checking system that C# emphasizes relies on data being strongly typed. This concept of being strongly typed means that C# is a language that is very readable and reliable. Being strongly typed does have the downside of generating more errors during compile time. With the large number of types, it is important for the programmer to know when to use each type during the proper scenario. Improper use of these types could potentially lead to increased errors. Conversely, a weakly typed language will be easier to write code in but will potentially end up having unpredictable code.

Control Abstractions

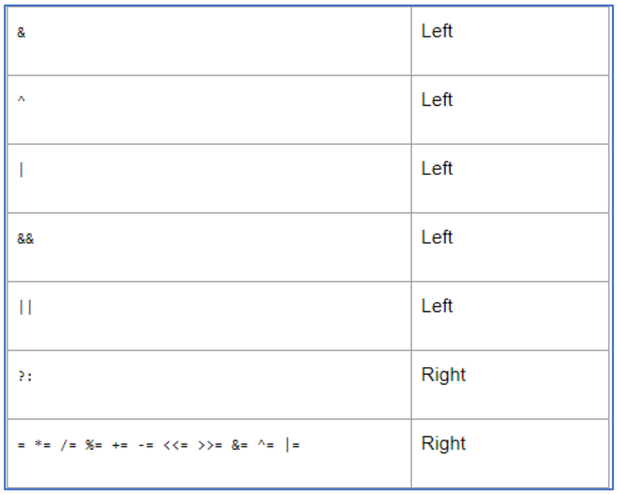
On top of using *binary expressions,* C# also utilizes what is called *lambda expressions* in its programming language. The use of lambda expressions is what allows C# to be considered a functional programming language. The idea of a lambda expression involves using functions that are created to then take it a step farther to define another function based on the current function. This concept is the idea behind creating what's called *delegates* or *expression trees*.

Operators are determined in C# by an order of operator precedence and associativity. Figure 3 will identify and show examples of some of the rules for operator precedence that are used in C#. The order of precedence is top down respectively with the graphic.



Operators	Associativity
(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked	Left
+(unary) -(unary) ~ ++x --x (T)x	Left
* / %	Left
+(arithmetic) -(arithmetic)	Left
<< >>	Left
< > <= >= is as	Left
== !=	Left

Figure 3. Order of Operations



&	Left
^	Left
|	Left
&&	Left
||	Left
?:	Right
= *= /= %= += -= <<= >>= &= ^= |=	Right

Figure 4. Order of Operations (cont.)

Due to the design concepts of C#, the *selection constructs* for the language were modeled identically to JAVA. The basic selection constructs are (Bhimani 2013):

- if construct
- if … else construct
- if … else if construct
- nested if construct
- switch … case construct

C# also supports the following iteration statements:

- The while loop
- The do … while loop
- The for loop
- The foreach loop

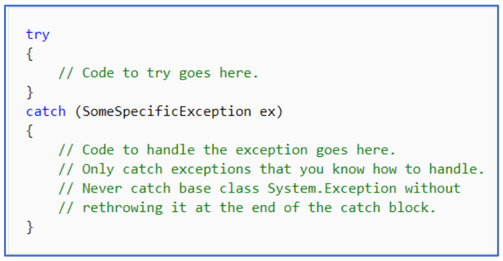
To add on to the flexibility of C#, the designers implemented multiple ways to pass parameters. By default, C# passes parameters by value. However, it is also possible to pass values by reference. To do this, the programmer just needs to utilize the “ref” keyword. An example of this would be:

private void Add(ref int x, ref int y)

Variable scopes in C# are very conventional. There are three types of scopes: class-level scope, method-level scope, and nested scope. The class-level scopes are variables that are declared within a class and are available to any non-static method within the class. Method-level scopes are available for use by any other part of the method including any nested code blocks. Nested scopes are variables that are declared within a nested block of code. They cannot be used by anything outside the code block where they were defined.

One of the many features that C# borrowed from C++ is the use of namespaces for program organization. Namespaces are used in C# to organize and provide a level of separation for code. Just like in C++, namespaces in C# can also be considered a container.

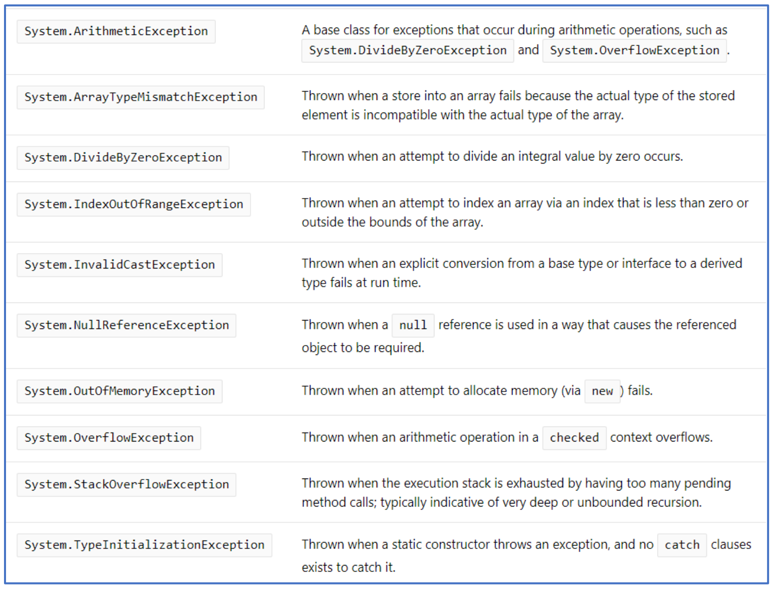
Exceptions in C# are handled similarly to how exceptions are handled in JAVA. The basic building blocks used to handle exceptions are try blocks, catch blocks, and finally blocks. A very common technique to handle exceptions in C# is as follows:



```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

Figure 5. Common Exception Technique

Some of the most commonly used exceptions are listed below.



`System.ArithmeticException`	A base class for exceptions that occur during arithmetic operations, such as `System.DivideByZeroException` and `System.OverflowException`.
`System.ArrayTypeMismatchException`	Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
`System.DivideByZeroException`	Thrown when an attempt to divide an integral value by zero occurs.
`System.IndexOutOfRangeException`	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
`System.InvalidCastException`	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
`System.NullReferenceException`	Thrown when a `null` reference is used in a way that causes the referenced object to be required.
`System.OutOfMemoryException`	Thrown when an attempt to allocate memory (via `new`) fails.
`System.OverflowException`	Thrown when an arithmetic operation in a `checked` context overflows.
`System.StackOverflowException`	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
`System.TypeInitializationException`	Thrown when a static constructor throws an exception, and no `catch` clauses exists to catch it.

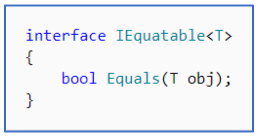
Figure 6. Common Exceptions

Support for Object Oriented Programming

Class declaration is very flexible in C#. Declaration of a class always contains the keyword “class” followed by an identifier (name). Along with this paradigm, there are other optional components that can be used for class declaration. The following components can be used, in order, pertaining to class declaration (Geeksforgeeks.org):

- **Modifiers**: A class can be public or internal etc. By default modifier of class is internal.
- **Keyword class**: A class keyword is used to declare the type class.
- **Class Identifier**: The variable of type class is provided. The identifier(or name of class) should begin with a initial letter which should be capitalized by convention.
- **Base class or Super class**: The name of the class’s parent (superclass), if any, preceded by the : (colon). This is optional.
- **Interfaces**: A comma-separated list of interfaces implemented by the class, if any, preceded by the : (colon). A class can implement more than one interface. This is optional.
- **Body**: The class body is surrounded by { } (curly braces).

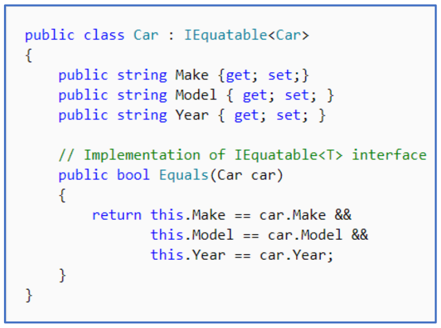
Interfaces in C# are recognized and declared by the keyword “interface” followed by the identifier (name). An example of an interface in C# is as follows:



```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

Figure 7. Interface Declaration

The implementation of an interface can be recognized and/or declared by a colon after the class declaration. See below for an example of interface implementation and usage.

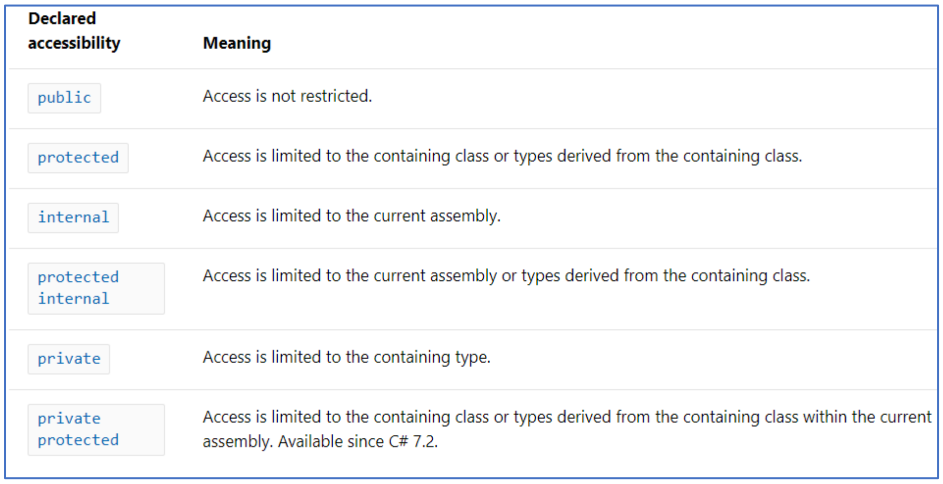


```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return this.Make == car.Make &&
               this.Model == car.Model &&
               this.Year == car.Year;
    }
}
```

Figure 8. Interface Implementation

One of the important qualities of C# is its ability to provide data protection by use of access modifiers in its structures. The access modifiers in C# closely model the access modifiers in JAVA. Listed below are the access modifiers that are used in C#.



Declared accessibility	Meaning
`public`	Access is not restricted.
`protected`	Access is limited to the containing class or types derived from the containing class.
`internal`	Access is limited to the current assembly.
`protected internal`	Access is limited to the current assembly or types derived from the containing class.
`private`	Access is limited to the containing type.
`private protected`	Access is limited to the containing class or types derived from the containing class within the current assembly. Available since C# 7.2.

Figure 9. Access Modifiers in C#

This idea of protection also introduces inheritance within C#. Objects in C# can be defined as parent or child objects of one another, meaning that the qualities of the parent object can be passed down to the child object. This includes any security via access modifiers that are being used by the parent class.

Sample Program

```
/******************************************************************************
Simple calculator program that keeps a running tally of previous calculations.
Continuously takes user input until prompted to quit. Supports +, -, *, /, and
exponentiation of floating point and integer numbers. Also has a built-in
feature to catch divide by zero exceptions.

@author R.J. Hamilton
@version December 2019
******************************************************************************/

using System;
using System.Collections.Generic;

namespace SampleCalculator
{
    public class Calculator
    {
        List<float> tally;        //Tally of the computation results.
        bool isRunning;           //Check to see if the calculator is running

        public Calculator()
```

```
{
    this.tally = new List<float>();
    isRunning = true;
}


/*******************************************************************
Main method that creates a calculator object and executes the
starting procedure.
*******************************************************************/
public static void Main(string[] args)
{
    Calculator calc1 = new Calculator();
    while (calc1.isRunning)
    {
        calc1.Start();
    }
}

/*******************************************************************
Start method displays the text that presents the user with all of the
different options to use the calculator.
*******************************************************************/
public void Start()
{
    Console.WriteLine("Please enter values that you wish to compute.\n" +
                      "Please use the following format: 3 + 5\n\n" +
                      "To quit press q.\nTo show the running tally, press t.\n" +
                      "------------------------------------------------");
    string userInput = Console.ReadLine();
    Console.Write("\n");

    if (userInput == "q")
    {
        Quit();
    }
    else if (userInput == "t")
    {
        ShowTally();
    }
    else if (userInput == "clear")
    {
        Console.Clear();
    }
    else
    {
        try
        {
            //Reads the input from the user and attemps to split the string into
            //usable pieces for our calculator.
            string[] seperatedInput = userInput.Split(null);
            float x = float.Parse(seperatedInput[0]);
            float y = float.Parse(seperatedInput[2]);
            string mathOperator = seperatedInput[1];

            Compute(x, mathOperator, y);
        }
        catch (Exception e)
        {
            Console.WriteLine("\nPlease use the following examples as the proper format:\n" +
                              "1 + 1\n2 - 2\n4 / 2\n5 * 5\n");
        }
    }
}

/*******************************************************************
Displays all of the previous results from our computations.
*******************************************************************/
```

```
        public void ShowTally()
        {
            if (tally.Count == 0)
            {
                Console.WriteLine("There are currently no values in the tally.");
            }
            else
            {
                Console.WriteLine("The values stored in the tally are:\n");
                tally.ForEach(Console.WriteLine);
                Console.WriteLine("\n");
            }
        }

        /***********************************************************************
        Performs the math involved with the desired operation selected by the
        user. Also displays the result as an output.
        ***********************************************************************/
        public void Compute(float x, string mathOperator, float y)
        {
            switch (mathOperator)
            {
                case "+":
                    float total = Addition(x, y);
                    Console.WriteLine("Result: " + x + " + " + y + " = " + total + "\n");
                    tally.Add(total);
                    break;
                case "-":
                    total = Subtract(x, y);
                    Console.WriteLine("Result: " + x + " - " + y + " = " + total + "\n");
                    tally.Add(total);
                    break;
                case "*":
                    total = Multiply(x, y);
                    Console.WriteLine("Result: " + x + " * " + y + " = " + total + "\n");
                    tally.Add(total);
                    break;
                case "/":
                    total = Divide(x, y);
                    Console.WriteLine("Result: " + x + " / " + y + " = " + total + "\n");
                    tally.Add(total);
                    break;
                default:
                    Console.WriteLine("Default");
                    break;
            }
        }

        public float Addition(float x, float y)
        {
            float sum = x + y;
            return sum;
        }

        public float Subtract(float x, float y)
        {
            float sum = x - y;
            return sum;
        }

        public float Multiply(float x, float y)
        {
            float sum = x * y;
            return sum;
        }

        public float Divide(float x, float y)
        {
            if (y == 0)
            {
                Console.WriteLine("Cannot divide by zero.\n");
                Start();
                return 0;
            }
            else
            {
                float sum = x / y;
                return sum;
            }
        }

        public void Quit()
        {
            Console.WriteLine("Are you sure you want to quit? (y/n)");
            string decision = Console.ReadLine();
            if (decision == "y" || decision == "Y")
            {
                System.Environment.Exit(1);
            }
        }
    }
}
```

References

Bhimani, Aakash. “Fundamentals of C#.” C# Corner, 18 Oct. 2013, www.c-sharpcorner.com/UploadFile/d0a1c8/basics-content-in-C-Sharp-net/.

“C# | Class and Object.” GeeksforGeeks, 12 July 2018, www.geeksforgeeks.org/c-class-and-object/.

C# Language Specification (PDF) (4th ed.). Ecma International. June 2006. Retrieved January 26, 2012.

Kovacs, James (September 7, 2007). "C#/.NET History Lesson". Retrieved June 18, 2009.

Horn, Bruce Van. “The History of C#.” *Lynda.com - from LinkedIn*, Lynda.com, 17 Jan. 2018, www.lynda.com/C-tutorials/history-C/642480/689716-4.html.

Mayo, Joe. “Informit.” *Policy | InformIT*, 2 Sept. 2001, www.informit.com/articles/article.aspx?p=23210&seqNum=12.

Mkhitaryan, Armina. “Why Is C# Among The Most Popular Programming Languages in The World?” Medium.com, Medium, 13 Oct. 2017, medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb.

Wagner, Bill. “Built-in Types Table - C# Reference.” Microsoft Docs, 16 Aug. 2018, docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/built-in-types-table.