CSCI 208 Research Paper

An investigation into C#

Arjuna Kankipati

12/9/2013

1. History of C#

C# was conceived by a team of Microsoft developers, led by Anders Hejlsberg and its development started towards the end of the 1990's. It was originally called Cool, which was an acronym for *C-Like Object Oriented Language*, (Hamilton) and this describes its main motivation, to bring Object-Oriented Programming to the C language. In its original release in 2002, it originally took a lot of concepts from Java, such as auto boxing, and changed some design choices to bring it closer to C++ in its overall design. It is based upon an imperative and object oriented design, while also bringing in declarative and functional features, without fully bringing in these paradigms. Its functional features can be found in the LINQ extension that adds features such as lambda expressions and anonymous functions and types. C# is run in a JIT compiler such as Java, however when it differs in that the JIT C# code interacts with a .NET runtime, and therefore you can use additional features in the .NET API. C# is fast becoming a popular programming language for software development over Java because of its adoption by Microsoft and thus its increasing feature set that exists on Windows.

2. Hello World

In C#, the Hello World program is very simple, and helps show some of the basics of the languages. You must first declare the namespace, which helps the compiler determine the scope of code. Within the namespace, you can declare structs as you would like, and in the case of this program, we declare a class Hello. By default, this is public. In C#, the main function is declared similar to Java in that it is and takes the command line arguments as a string array, but it differs in that you can either have it return void, or an int. By default, when you create a .cs file in Visual Studio, it imports the System class that allows you to abbreviate several function calls that we can see in the program below. Instead of having to write System.Console.WriteLine(arg), we can write Console.WriteLine(arg). This is a similar design choice to C, but is not a feature we find in Java, the most syntactically similar programming language to C#.

```
using System;
namespace HelloWorld {
    class Hello {
        static void Main(string[] args) { // this can return void or an int
            Console.WriteLine("Hello World!");
            // This code is used to keep the console window open when
            // running on windows
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

3, 4, 5. Static or Dynamic Typing? Strongly or Weakly Typed? Explicit or Implicit Coercion?

In this response, I will answer two questions. In C#, it is easiest to define the language as statically typed, with mostly strongly typed syntax, although it does incorporate some dynamic typing and weakly typed ideas into the language. For example, if we look in the following code sample, we see that it is statically typed because you there is type checking that occurs at compile time. However, C# has an interesting design choice where you can declare the type as a *var,* and C# will use dynamic typing. This tells the complier to implicitly infer the type of the variable, and cast it to that type, and then it will revert to being statically typed and it will require type checking further in the program. We can also look at if C# is a strongly or weakly typed. Because you are required to declare types, we can say that C# is strongly typed, and it also requires explicit casting when casting between some primitives. However, C# also has some weakly typed as it is possible to implicitly cast variables as we see in Console.WriteLine(s + x), where we are implicitly coercing the int to a string to print it. As I have mentioned before, it is possible to implicitly coerce variables, but some coercions need an explicit cast. C# follows Java's design choice to decide if it will implicitly coerce – no data must be lost, otherwise called a type safe coercion in C#. However, it is still possible to explicitly cast variables as we can see in the code, and C# also provides a System.Convert class that will help to convert between non-compatible types.

```
static void Main(string[] args)
{
    int x = 6; //strongly typed - must declare type
    string s = "This is a string!";
    var d = s; // dynamic typing will automatically infer and set type;
    double a = 5; // will implicitly coerce from int to double
    int e = (int) 5.6; // must explicitly cast from double to int
    Console.WriteLine(s + x); // weakly typed - can still implicitly coerce
    x = s; //statically typed - checking types at compile
}
```

```
(local variable) string s

Error:
  Cannot implicitly convert type 'string' to 'int'
```

6, 7, 8. Primitive size, range and String operations

In C#, we have many different primitive types available to a programmer, and many of the options available are very similar to what is found in Java. However, C# has some additional types that offer increased accuracy for numbers that are not considered primitives. To get the length of primitives, as well as if they are primitives, I wrote a short program that we can see below. Here, we are just getting the size, and the type using functions offered by the language.

```
static void Main(string[] args){
	Console.WriteLine("The size of a boolean is: " + sizeof(bool)); // Value = 1
	Console.WriteLine("Is Boolean primitive? " + typeof(bool).IsPrimitive); // True
}
```

Using this function, I have compiled the table below that shows both the length of the available types, and if they are considered primitives by the language. As we can see, string is not considered a primitive by the language, a design choice we find in Java as well. Another design choice it shares with Java is that strings are immutable, thus meaning if you wish to change a string, another string object will be created with the result of any operation done on the string. It is possible to use the + operator to concatenate strings, and apart from the == operator, not other math operator is available for strings. Interestingly, in Java, where due to the fact that strings are stored as references, the design choice was that the == operator will compare if the strings point to the same place in memory. In C# however, the design choice was that the == operator, despite string also being a reference type, will compare the value of the string instead of where it points to. Some additional functions that are available on strings are SubString, IndexOf, ToLower, ToUpper and many others.

C# name	Size	Primitive	Range	.NET Framework
bool	1	True	True or False	System.Boolean
byte	1	True	0 to 255	System.Byte
sbyte	1	True	-128 to 127	System.SByte
char	2	True	Unicode symbols used in text	System.Char
decimal	16	False	±1.0 × 10e−28 to ±7.9 × 10e28	System.Decimal
double	8	True	-1.79769313486232e308 to 1.79769313486232e308	System.Double
float	4	True	-3.402823e38 to 3.402823e38	System.Single
int	4	True	-2,147,483,648 to 2,147,483,647	System.Int32
uint	4	True	0 to 4294967295	System.UInt32
long	8	True	-9223372036854775808 to 9223372036854775807	System.Int64
ulong	8	True	0 to 18446744073709551615	System.UInt64
object	2	True	n/a	System.Object
short	2	True	-32,768 to 32,767	System.Int16
ushort	n/a	False	0 to 65535	System.UInt16
string	n/a	False	n/a	System.String

9. Pointers and References

In C#, it is possible to use pointers, a feature that was brought over from C++. However, because it is also a garbage-collected language (will be shown later), it is necessary to tell the compiler that you are going to be manipulating the memory through the use of the keyword *unsafe*. This tells the compiler, and as a result, the program will no longer garbage collect. It is possible to directly manipulate pointers, and it is also possible to do other operations, such as compare pointers, increment and decrement the pointers and also access methods of a struct

using the pointer. By default, C# does not support pointer arithmetic, but by using the unsafe keyword we are declaring a block where pointers can be used (Curnrow). In this example, I have declared the whole function as unsafe, but it is also possible to have a block declared in curly brackets as unsafe.

```
static unsafe void Main(string[] args) {
	int x = 6;
	int* p1 = &x;
	int** p2 = &p1;
	*p1 += 3;
	**p2 -= 3;
	Console.WriteLine("Do p1 and p2 point to the same place? " + (*p2 == p1));
}
```

10. Arrays

C# uses a similar model to Java for its arrays, and it also allows multi-dimensional. In the following code below, I show how you are required to declare arrays within C#. The declaration of a single-dimensional array is identical to the syntax used within Java, although the syntax differs for multi-dimensional arrays. To declare a multi-dimensional array, you first indicate how many additional dimensions using a comma in the initial square brackets. In tests that I performed where I checked the number of additional dimensions that you could create, I found that I could not get the program to force an error regarding the number of dimensions, so I can only assume that C# allows an infinite number of dimensions up to the point where you overload the memory. It would also be possible to do this in an extremely large single dimensional array. It is nice that C# allows this as part of its feature set, although a seven-dimensional array is not extremely practical in a real-world situation. There is an additional form that multi-dimensional arrays can take. In a traditional array, you have a grid of values that you can index using a row and column. In this second form, you create an array of arrays, such that you have separate arrays at each location. This is then a jagged array, as you can specify dimensions for each individual array.

```
static void Main(string[] args) {
	int[] array = {1, 3, 4, 6, 7};
	int[,] arrayOne = new int[4, 2]; // uniform multi-d array
	int[, , , , , ,] arrayTwo = new int[4, 2, 3, 4, 6, 7, 8];
	int[][] arr = new int[4][]; // this is an array of arrays - jagged
	arr[0] = new int[5]; // each index has a set of arrays
	for (int x = 0; x < 4; x++) {
			for (int y = 0; y < 2; y++) {
			arrayOne[x, y] = x + y; // default value is 0
			}
	}
	arrayTwo[1, 1, 2, 2, 3, 4, 2] = 4;
}
```

11. Short-Circuiting

C# does have an implementation of short-circuit that I tested using a short program. Using two if statements that will test short-circuiting, the expected output should be "This should happen", and we can trace the program to prove it. In the first if-statement, if C# short-circuits, it should skip the second test, and thus not print anything. In the second if-statement, if short-circuiting applies, then the program should straight away go into the loop, thus skipping foo(), and printing "This should happen!". When we run the program, we find that this is the case.

```
static void Main(string[] args) {
	if (false && foo()) {
		Console.WriteLine("Can never happen!");
	}
	if (true || foo()) {
		Console.WriteLine("This should happen!");
	}
}
static bool foo() {
	Console.WriteLine("Tested!");
	return false;
}
```

12. Anonymous Functions/Lambda Expressions

In C#, there is an implementation of anonymous functions that they call Lambda Expressions. It is a fairly simple implementation, and while it doesn't offer the depth that Haskell allow in anonymous functions, it still is a feature that many languages used for software engineering do not have. The main limitation that exists with its implementation is that it is necessary to declare the return type, type and number of parameters that the function will have. As we can see in the example below, I declare a function structure that will return an int, and take in three ints as parameters. It is possible to declare the function in one line, or pass the function separately as we can see in the second example.

```
public delegate int MyFunc(int x, int y, int z); // declares the structure for functions
static void Main(string[] args) {
	MyFunc func = (x, y, z) => (int) Math.Pow(x, y) * z;
	int answer = doMath(3, 4, 5, func); // pass the function and get answer
	answer = doMath(5, 6, 7, (x, y, z) => x * y * z);// pass function when declaring
}
public static int doMath(int test, int testTwo, int testThree, MyFunc math) {
	return math(test, testTwo, testThree);
}
```

13. The Dangling Else problem and C#'s Solution

The dangling else problem is common, and as all languages have done, C# has a solution. In C#, your primary IDE is Visual Studio, and this helps form the solution. C#'s primary solution is obviously to use curly braces, but in the case where there is no whitespace, C# will match the

innermost if. To help reinforce this, Visual Studio uses smart-indenting to move the appropriate block to the correct indent. If you do decide to use curly braces, it will then move the else block back to the left if you place the curly brackets around the second if statement. The design choice to match the innermost if is very common in C-derivatives.

```
static void Main(string[] args){
	int x = 5;
	if (x > 4) // true
		if (x > 6) // false
			Console.WriteLine("Hello");
		else // C# will then print this as it innermost matches
			Console.WriteLine("World");
}
```

14, 15. Method and Operator Overloading

C# allows you to overload both methods that you create, as well as operators that are present in the language. To demonstrate this, I have written a short class that represents a Line, and uses both method and operator overloading. To overload a operator, you must first declare the function as static, and use the keyword static in order to tell the compiler you are overloading an operator. You must also add the appropriate return type, as well as take in the correct parameters. Once you have done this, you can add behaviors within the method that you choose. To overload a method that you have written, you do not need to tell the compiler anything, instead you just need to declare the function with different parameters. As in other languages, it is necessary to keep the return types the same

```
class Line // class to represent a line in the form y = mx + c
{
	public int m, c;
	public Line(int m1, int c1){ this.m = m1; this.c = c1; }

	public static Line operator +(Line l1, Line l2) { // overloading the + operator
		return new Line(l1.m + l2.m, l1.c + l2.c);
	}
	public static bool operator ==(Line l1, Line l2) { // overloading == operator
		if ((l1.m == l2.m) && (l1.c == l2.c)) { return true; }
		return false;
	}
	public static bool operator !=(Line l1, Line l2) { return !(l1 == l2); }
	public void increment() { this.m++; this.c++; }
	public void increment(int x) { this.m += x; this.c += x; }
	public void increment(int x, int y) { this.m += x; this.c += y; }
}
```

16. Exceptions

In C#, exception handling is again designed very similarly to that of Java, in that error handling is found in the use of try and catch blocks. All errors derive from the parent class System.Exception, which allows custom errors to be created that extend from BaseException.

It is possible to throw errors that can be caught using a try-catch block. Catch blocks are specific to errors, although it is also possible to throw a finally block, that will be executed if any error is caught.

```
static void Main(string[] args) {
       double result = 0;
       try { result = divide(5, 0); }
       catch (DivideByZeroException e) { Console.WriteLine("ERROR!"); }
}
public static double divide(double one, double two) {
       if (two == 0) throw new System.DivideByZeroException();
       return x / y;
}
```

17 & 18. Precedence in math operators and associativity & operators

In C#, all binary operators, excluding assignment operators, are considered left-associative, meaning that the following: x + y + z would be evaluated as (x + y) + z. The assignment operator, as well as the conditional operator (?:) is considered right-associative, meaning that x = y = z is evaluated as x = (y = z). The math operators within C# have the following order of precedence. In this list, you can also see the different operators that are present such as the unary operator:

1. **Primary:** x.y, f(x), a[x], x++, x--, new, typeof, checked, unchecked
2. **Unary:** +, -, !, ~, ++x, --x, (T)x
3. **Multiplicative:** *, /, %
4. **Additive:** +, -
5. **Shift:** <<, >>
6. **Relational and type testing:** <, >, <=, >=, is, as
7. **Equality:** ==, !=
8. **Logical :** &, ^, | - the precedence of logic operators is in this order, moving left to right
9. **Conditional:** &&, ||, ?: - the precedence of these operators is in order, moving left to right
10. **Assignment:** =, *=, /=, +=, -=, <<=, >>=, &=, ^=, |=

It is possible to manipulate precedence and associativity using brackets, as in other languages, and this allows for more complex expressions to be evaluated.

19. Parameter Passing

C# has some interesting features available to programmers with regards to parameter passing. In terms of memory itself, C# follows Java's design choice of having primitive variables containing its data, whereas constructed types, otherwise known as reference-types in C#, containing a reference to the data on the heap. While in Java, it is only possible to pass

primitives by value, and reference types by reference, in C#, it is possible using some interesting design choices to choose for both primitives and reference types which strategy you would like to use. By default, primitives and reference types both pass by value, although for reference types, it is not a pure pass by value strategy. It is not possible to change where the reference type points to, as is shown below – trying to point the entire variable to a new memory location will be local – but it is possible to change individual elements of the type. However, if you would like to pass by reference, it requires that you create an identical function, which has the same method header, except the parameters require a ref parameter. Then, when you choose to call this function, you must pass the reference to the variable by prefixing the variable name with the keyword ref.

```
static void Main(string[] args) {
	int x = 6; // primitive
	square(x); // value is still 6 after called
	square(ref x); // passes by reference, x will be 36 out of block
	int[] arr = { 1, 2, 3 }; // reference type - points to {1,2,3}
	change(arr); // arr[0] will equal 4
	change(ref arr); // arr[0] will equal -3
}
static void change(int[] pArray) { // for change(ref arr) to work, ref must be added
	pArray[0] = 4; // this will change globally
	pArray = new int[5] { -3, -1, -2, -3, -4 }; // this is local
}
static void square(ref int x) { x = x * x; }
static void square(int x) { x = x * x; }
```

20. Order of parameter evaluation and evaluation strategy

In C#, if you want to pass a function's value to a function, it is easy to test in which order they will be evaluated. As we can see in the code below, when passing a function call to another function, the function firsts evaluates the function, from left to right, making C#'s design choice both call-by-value and call-by-reference, as it uses both parameter passing strategy. It also evaluates from left to right as we can find by running the program below.

```
static void Main(string[] args) { // prints 5, 6, 7, Starting, 10.488
	Console.WriteLine(squareRoot(square(5), square(6), square(7)));
}
static double squareRoot(double x, double y, double z) {
	Console.WriteLine("Starting");
	return Math.Sqrt(x + y + z);
}
static double square(double x) {
	Console.WriteLine("Hi! " + x);
	return Math.Pow(x, 2);
}
```

21. Named and Optional parameters

In C#, there is a very cool and useful feature relating to function parameters and calling functions. There are many situations where you may have a function that requires a lot of parameters to get an answer, and therefore, you may forget the order of the parameters as declared in the header. In C#, it is possible to name the parameters in the function call as is shown below such that means that as long as you can remember the parameter names, you can call the function with parameters in any order. It also works that if you remember the order of some variables, you can pass them regularly, and then affix the named parameters after that! Also present in C# as of recently is optional parameters, which work very similarly to Python, meaning you can declare a parameter in the method header, and choose whether or not a different value is needed!

```
static void Main(string[] args) {
	int answer = foo(y: 4, d: 5, c: 4, e: 5, x: 3); // mix up order, but named
	answer = foo(4, 5, q: 3, e: 3); // x, y ordered, named some, and missed optional
}
static int foo(int x, int y, int d, int q, int e, int c = 4) {
	return x * y / (d + q) % (e + c);
}
```

22 & 23. Inheritance & Method Binding

With recent updates to C#, the newest version of C# brought dynamic binding officially to C#, although the implementation of it can be seen in two ways. Firstly, we can see that C# does implicitly dynamically bind objects by creating a new object C c = new D(). When we then run the overridden method printClass(), we get the D implementation of the function. However, C# also allows you to declare the variable using the keyword dynamic, which then will get the type itself.

C# also has a simple implementation of inheritance available. Firstly, the design choice when the language was created was not to allow multiple inheritance, although it is possible to have a chain of inheriting classes, which is shown below to demonstrate an interesting design choice in C#. Here we have four classes, of which we have the path – D -> C -> B -> A. we can see that it is possible to override functions using the keyword override, and I have in all four classes. However, in C, we can demonstrate a unique feature. It is possible using the keyword new in the method header, to hide the superclass's implementation of the function, such that further classes that inherit from this class will recognize C's implementation as the "super" function. This means that if we declare A a = new D(), the furthest the super class A can look down the chain is to B, which means it will print B's implementation of printClass().

```
class A {
	public virtual void printClass() {  Console.WriteLine(" A"); }
}
class B : A {
	public override void printClass() { Console.WriteLine("B - overriden A"); }
}
class C : B {
	public new virtual void printClass() {
		Console.WriteLine("C - new fn w/ no override "); }
}
class D : C {
	public override void printClass() { Console.WriteLine("D - overriden C"); }
}

static void Main(string[] args)
{
	C c = new D();
	c.printClass(); // prints D
	A a = new D();
	a.printClass(); // prints B
	dynamic aOne = new B();
	aOne.printClass(); // also prints B
}
```

---

24 & 25. Comments in C# and Grammar

In C#, we have two forms of comments that we can use. The first is a single-line comment, whose structure is similar to most C-derivative languages. These comments begin with a double-backslash, and cannot extend over several lines. The second form of comment is a delimited comment, which can extend over several lines. These begin with /*, and are ended by */. While by default, Visual Studio will enter a * on each continuous line of comment, these are not required by the language, they are instead there to help readability of code. Note that in comments, the character sequences that define a comment have no meaning within each other, so you can use them as wanted.

```
// This is a single line comment. /* doesn't have a meaning withing here
/* this is a multi line comment
 * // this has not meaning, and neither does /*
 */
```

Being a feature-filled programming language makes it useful for programmers, but also means that it has a fairly complicated grammar. Unfortunately, the grammar that Microsoft have released to describe C# does not follow the traditional BNF grammar system, and in addition, it is a fairly long grammar, comprising several different sections that must be broken up to be readable. To show you the grammar that C# uses, I have printed below the section relating to comments

comment:
single-line-comment
delimited-comment
single-line-comment:
// input-characters$_{opt}$
input-characters:
input-character
input-characters input-character
input-character:
Any Unicode character except a *new-line-character*
new-line-character:
Carriage return character (U+000D)
Line feed character (U+000A)
Line separator character (U+2028)
Paragraph separator character (U+2029)
delimited-comment:
/ *delimited-comment-characters*$_{opt}$ **/*
delimited-comment-characters:
delimited-comment-character
delimited-comment-characters delimited-comment-character
delimited-comment-character:
not-asterisk
** not-slash*
not-asterisk:
Any Unicode character except *
not-slash:
Any Unicode character except /

Bibliography

Curnrow, Phil. *Using Pointers in C#*. 2 October 2007. 25 November 2013. <http://www.c-sharpcorner.com/UploadFile/pcurnow/usingpointers10022007082330AM/usingpointers.aspx>.

Hamilton, Naomi. *The A-Z of Programming Languages: C#*. 01 October 2008. Interview. 28 November 2013. <http://www.computerworld.com.au/article/261958/a-z_programming_languages_c_/>.

MSDN Articles Referenced

http://msdn.microsoft.com/en-us/library/vstudio/bb397687.aspx - Lambda Expressions

http://msdn.microsoft.com/en-us/library/ms173105.aspx - Casting and Type Conversions

http://msdn.microsoft.com/en-us/library/vstudio/y31yhkeb.aspx - Pointer Types

http://msdn.microsoft.com/en-us/library/vstudio/2yd9wwz4.aspx - Multi-dimensional Arrays

http://msdn.microsoft.com/en-us/library/vstudio/9b9dty7d.aspx - Arrays

http://msdn.microsoft.com/en-us/library/2s05feca(v=vs.80).aspx – Jagged Arrays

http://msdn.microsoft.com/en-us/library/aa288467(v=vs.71).aspx - Operator Overloading

http://msdn.microsoft.com/en-us/library/ms229029(v=vs.110).aspx - Method Overloading

http://msdn.microsoft.com/en-us/library/vstudio/ms229064(v=vs.100).aspx – Custom Exceptions

http://msdn.microsoft.com/en-us/library/ms173160.aspx - Exception Handling

http://msdn.microsoft.com/en-us/library/aa691323(v=vs.71).aspx – Precedence and Associativity

http://msdn.microsoft.com/en-us/library/ms228360(v=vs.90).aspx – Primitive Sizes

http://msdn.microsoft.com/en-us/library/9t0za5es.aspx - Passing Value Types

http://msdn.microsoft.com/en-us/library/s6938f28.aspx - Passing Reference Types

http://msdn.microsoft.com/en-us/library/dd264739.aspx – Named and Optional Parameters

http://msdn.microsoft.com/en-us/library/435f1dw2.aspx - The New Modifier for Inheritance

http://msdn.microsoft.com/en-us/magazine/gg598922.aspx - Dynamic Binding

http://msdn.microsoft.com/en-us/library/aa664812(v=vs.71).aspx – The full C# BNF Grammar