***Extension Methods***

Essentially, the Extension Method allows us to add a new method to an existing class:

* Without modifying it or adding code
* Without extending it or creating a new derived type
* Without recompiling the class

Code of extension methods

namespace Extension\_Methods

{

public static class StringExtension

{

public static int WordCount(this string s)

{

int count = 0;

for (int i = 0; i < s.Length; i++)

{

if (s[i] != ' ')

{

if ((i + 1) == s.Length)

{

count++;

}

else

{

if (s[i + 1] == ' ')

{

count++;

}

}

}

}

return count;

}

}

}

<https://www.infragistics.com/community/blogs/dhananjay_kumar/archive/2016/07/05/what-is-the-extension-method-in-c.aspx>

***Generics***

Generics allow you to delay the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

// Declare the generic class.

public class GenericList<T>

{

void Add(T input) { }

}

class TestGenericList

{

private class ExampleClass { }

static void Main()

{

// Declare a list of type int.

GenericList<int> list1 = new GenericList<int>();

// Declare a list of type string.

GenericList<string> list2 = new GenericList<string>();

// Declare a list of type ExampleClass.

GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();

}

}

***C# releases:***

2.0

Undoubtedly, the highlight feature in .NET 2.0 and C# 2.0 is generics

3.0

C# 3.0 supports implicitly typed local variables by making use of a new keyword var. These variables are not typeless; rather, their type is inferred at compile time.

Auto-implemented properties are another helpful new feature to save us some typing and reduce the potential to introduce bugs ex get; set;

C# 3.0 introduces two new features that help when instantiating and initializing object instances. Using object and collection initializers, you can instantiate and initialize either an object or a collection in one compound statement.

C# 2.0 introduced partial class definitions.C# 3.0 adds to that by introducing partial methods. Using partial methods, We can implement the methods in separate file (same partial class)

Extension methods are one of the most exciting new features.Extension methods enable you to add methods to existing types without creating a new derived type

Language Integrated Query(LINQ)

4.0

Main focus is on dynamic programming

Optional parameters

***Classes and structs***

In .NET, there are two categories of types, *reference types* and *value types*.

Structs are *value types* and classes are *reference types*.

The general difference is that a *reference type* lives on the heap, and a *value type* lives inline, that is, wherever it is your variable or field is defined.

A variable containing a *value type* contains the entire *value type* value. For a struct, that means that the variable contains the entire struct, with all its fields.

A variable containing a *reference type* contains a pointer, or a *reference* to somewhere else in memory where the actual value resides.

This has one benefit, to begin with:

* value types always contains a value
* reference types can contain a null-reference, meaning that they don't refer to anything at all at the moment

Internally, reference types are implemented as pointers, and knowing that, and knowing how variable assignment works, there are other behavioral patterns:

* copying the contents of a value type variable into another variable, copies the entire contents into the new variable, making the two distinct. In other words, after the copy, changes to one won't affect the other
* copying the contents of a reference type variable into another variable, copies the reference, which means you now have two references to the same somewhere else storage of the actual data. In other words, after the copy, changing the data in one reference will appear to affect the other as well, but only because you're really just looking at the same data both places

When you declare variables or fields, here's how the two types differ:

* variable: value type lives on the stack, reference type lives on the stack as a pointer to somewhere in heap memory where the actual memory lives (though note [Eric Lipperts article series: The Stack Is An Implementation Detail](https://blogs.msdn.microsoft.com/ericlippert/2009/04/27/the-stack-is-an-implementation-detail-part-one/).)
* class/struct-field: value type lives completely inside the type, reference type lives inside the type as a pointer to somewhere in heap memory where the actual memory lives.

**Classes Only:**

* Can support inheritance
* Are reference (pointer) types
* The reference can be null
* Have memory overhead per new instance

**Structs Only:**

* Cannot support inheritance
* Are value types
* Are passed by value (like integers)
* Cannot have a null reference (unless Nullable is used)
* Do not have a memory overhead per new instance - unless 'boxed'

**Both Classes and Structs:**

* Are compound data types typically used to contain a few variables that have some logical relationship
* Can contain methods and events
* Can support interfaces

**Passing by value and passing by reference**

When a parameter is passed by reference, the caller and the callee use the same variable for the parameter. If the callee modifies the parameter variable, the effect is visible to the caller's variable.

When a parameter is passed by value, the caller and callee have two independent variables with the same value. If the callee modifies the parameter variable, the effect is not visible to the caller.

**Anonymous methods**

C# delegates are similar to pointers to functions, in C or C++. A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the System.Delegateclass.

<https://www.tutorialspoint.com/csharp/csharp_delegates.htm>

Anonymous methods provide a technique to pass a code block as a delegate parameter. Anonymous methods are the methods without a name, just the body.

You need not specify the return type in an anonymous method; it is inferred from the return statement inside the method body

<https://www.tutorialspoint.com/csharp/csharp_anonymous_methods.htm>

**Properties**

Setters and getters where excellently covered in the faculty

Abstract Classes were covered moderately in the faculty

https://www.tutorialspoint.com/csharp/csharp\_properties.htm

**Indexer**

An indexer allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves similar to a virtual array. You can then access the instance of this class using the array access operator ([ ]).

<https://www.tutorialspoint.com/csharp/csharp_indexers.htm>

**Events**

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication

https://www.tutorialspoint.com/csharp/csharp\_events.htm

**Collections**

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

<https://www.tutorialspoint.com/csharp/csharp_collections.htm>

**Unsafe code**

The same as pointers in C++ , but in C# any part of code that uses pointers(addrsses) is unsafe code. The unsafe part may be a function or a part of it