# C# Version History:

C# is a simple & powerful object-oriented programming language developed by Microsoft.

C# has evolved much since its first release in 2002. C# was introduced with .NET Framework 1.0 and the current version of C# is 6.0.

The following table lists important features introduced in each version of C#:

|  |  |  |  |
| --- | --- | --- | --- |
| Version | .NET Framework | Visual Studio | Important Features |
| C# 1.0 | .NET Framework 1.0/1.1 | Visual Studio .NET 2002 | * Basic features |
| C# 2.0 | .NET Framework 2.0 | Visual Studio 2005 | * Generics * Partial types * Anonymous methods * Iterators * Nullable types * Private setters (properties) * Method group conversions (delegates) * Covariance and Contra-variance * Static classes |
| C# 3.0 | .NET Framework 3.0\3.5 | Visual Studio 2008 | * Implicitly typed local variables * Object and collection initializers * Auto-Implemented properties * Anonymous types * Extension methods * Query expressions * Lambda expressions * Expression trees * Partial Methods |
| C# 4.0 | .NET Framework 4.0 | Visual Studio 2010 | * Dynamic binding (late binding) * Named and optional arguments * Generic co- and contravariance * Embedded interop types |
| C# 5.0 | .NET Framework 4.5 | Visual Studio 2012/2013 | * Async features * Caller information |
| C# 6.0 | .NET Framework 4.6 | Visual Studio 2013/2015 | * Expression Bodied Methods * Auto-property initializer * nameof Expression * Primary constructor * Await in catch block * Exception Filter * String Interpolation |

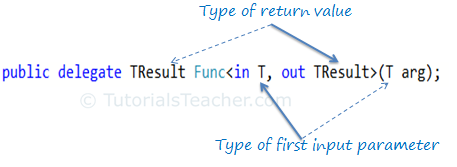
Func

C# 3.0 includes built-in generic delegate types **Func** and **Action**, so that you don't need to define custom delegates as above.

Func is a generic delegate included in the *System* namespace. It has zero or more **input** parameters and one **out** parameter. The last parameter is considered as an out parameter.

|  |
| --- |
| namespace System  {  public delegate TResult Func<in T, out TResult>(T arg);  } |

The last parameter in the angle brackets <> is considered as the return type and remaining parameters are considered as input parameter types as shown in the following figure.



### Points to Remember:

1. Func is built-in delegate type.
2. Func delegate type must return a value.
3. Func delegate type can have zero to 16 input parameters.
4. Func delegate type can be used with an [anonymous method](http://www.tutorialsteacher.com/csharp/csharp-anonymous-method) or [lambda expression](http://www.tutorialsteacher.com/linq/linq-lambda-expression).

# Action

Action is also a delegate type defined in the System namespace. An Action type delegate is the same as [Func delegate](http://www.tutorialsteacher.com/csharp/csharp-func-delegate) except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.

## Advantages of Action and Func delegates:

1. Easy and quick to define delegates.
2. Makes code short.
3. Compatible type throughout the application.

### Points to Remember:

1. Action delegate is same as Func delegate except that it does not return anything. Return type must be void.
2. Action delegate can have 1 to 16 input parameters.
3. Action delegate can be used with **anonymous methods** or **lambda expressions**.

# Predicate:

A predicate is also a delegate like [Func](http://www.tutorialsteacher.com/csharp/csharp-func-delegate) and [Action](http://www.tutorialsteacher.com/csharp/csharp-action-delegate) delegates. It represents a method that contains a set of criteria and checks whether the passed parameter meets those criteria or not. A predicate delegate methods must take one input parameter and it then returns a Boolean value - true or false.

The Predicate delegate is defined in the System namespace as shown below:

Predicate signature: public delegate bool Predicate<in T>(T obj);

### Points to Remember:

1. Predicate delegate takes one input parameter and Boolean return type.
2. Predicate delegate must contain some criteria to check whether supplied parameter meets those criteria or not.
3. [Anonymous method](http://www.tutorialsteacher.com/csharp/csharp-anonymous-method) and [Lambda expression](http://www.tutorialsteacher.com/linq/linq-lambda-expression) can be assigned to the predicate delegate.

# Extension Method:

### Points to Remember:

1. Extension methods are additional custom methods which were originally not included with the class.
2. Extension methods can be added to custom, .NET Framework or third party classes, structs or interfaces.
3. The first parameter of the extension method must be of the type for which the extension method is applicable, preceded by the **this** keyword.
4. Extension methods can be used anywhere in the application by including the namespace of the extension method.

# Anonymous Type:

### Points to Remember:

1. Anonymous type can be defined using the new keyword and object initializer syntax.
2. The implicitly typed variable- **var**, is used to hold an anonymous type.
3. Anonymous type is a **reference type** and all the properties are read-only.
4. The scope of an anonymous type is local to the method where it is defined. Usually, you cannot pass an anonymous type to another method; however, you can pass it to a method that accepts a parameter of [dynamic type](http://www.tutorialsteacher.com/csharp/csharp-dynamic-type). Please note that Passing anonymous types using dynamic is not recommended.

# Dynamic Type:

### Points to Remember:

1. The dynamic types are resolved at runtime instead of compile time.
2. The compiler skips the type checking for dynamic type. So, it doesn't give any error about dynamic types at compile time.
3. The dynamic types do not have intellisense support in visual studio.
4. A method can have parameters of the dynamic type.
5. An exception is thrown at runtime if a method or property is not compatible.

# Var - Implicit Type:

C# 3.0 introduced the implicit typed local variable "var". Var can only be defined in a method as a local variable. The compiler will infer its type based on the value to the right of the "=" operator.

Var can be used in the following different contexts:

* Local variable in a function
* For loop
* Foreach loop
* Using statement
* As an anonymous type
* In a LINQ query expression

### Points to Remember:

1. *var* can only be declared and initialized in a single statement. Following is not valid:   
   var i;   
   i = 10;
2. *var* cannot be used as a field type at the class level.
3. *var* cannot be used in an expression like var i += 10;
4. Multiple vars cannot be declared and initialized in a single statement. For example, var i=10, j=20; is invalid.

# Hashtable:

C# includes Hashtable collection in *System.Collections* namespace, which is similar to generic [Dictionary](http://www.tutorialsteacher.com/csharp/csharp-dictionary) collection. The Hashtable collection stores key-value pairs. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.

### Points to Remember:

1. Hashtable stores key-value pairs of any datatype where the Key must be unique.
2. The Hashtable key cannot be null whereas the value can be null.
3. Hashtable retrieves an item by comparing the hashcode of keys. So it is slower in performance than Dictionary collection.
4. Hashtable uses the default hashcode provider which is object.GetHashCode(). You can also use a custom hashcode provider.
5. Use DictionaryEntry with foreach statement to iterate Hashtable.

# Indexer:

An Indexer is a special type of property that allows a class or structure to be accessed the same way as array for its internal collection. It is same as property except that it defined with **this** keyword with square bracket and parameters.

|  |
| --- |
| class StringDataStore  {    private string[] strArr = new string[10]; // internal data storage  public StringDataStore()  {  }  public string this[int index]  {  get  {  if (index < 0 && index >= strArr.Length)  throw new IndexOutOfRangeException("Cannot store more than 10 objects");  return strArr[index];  }  set  {  if (index < 0 && index >= strArr.Length)  throw new IndexOutOfRangeException("Cannot store more than 10 objects");  strArr[index] = value;  }  }  }  class Program  {  static void Main(string[] args)  {  StringDataStore strStore = new StringDataStore();  strStore[0] = "One";  strStore[1] = "Two";  strStore[2] = "Three";  strStore[3] = "Four";    for(int i = 0; i < 10 ; i++)  Console.WriteLine(strStore[i]);  }  } |

### Points to Remember:

1. An indexer is same as property except that it defined with **this** keyword with square bracket that takes paramter.
2. Indexer can be override by having different types of parameters.
3. Ref and out parameter with the indexer is not supported.
4. Indexer can be included as an interface member.
5. Use code snippet to insert indexer syntax automatically in the visual studio.

# Stream:

C# includes following standard IO (Input/Output) classes to read/write from different sources like a file, memory, network, isolated storage, etc.

**Stream:** *System.IO.Stream* is an abstract class that provides standard methods to transfer bytes (read, write, etc.) to the source. It is like a wrapper class to transfer bytes. Classes that need to read/write bytes from a particular source must implement the Stream class.

The following classes inherits Stream class to provide functionality to Read/Write bytes from a particular source:

**FileStream** reads or writes bytes from/to a physical file whether it is a .txt, .exe, .jpg or any other file. FileStream is derived from the Stream class.

**MemoryStream:** MemoryStream reads or writes bytes that are stored in memory.

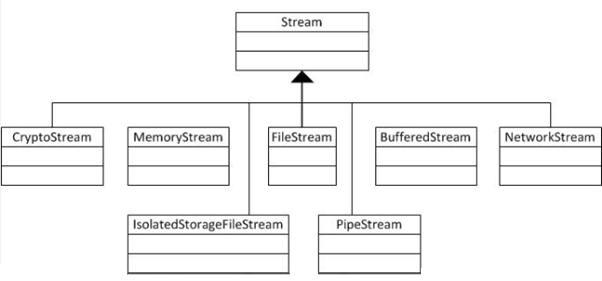
**BufferedStream:** BufferedStream reads or writes bytes from other Streams to improve the performance of certain I/O operations.

**NetworkStream:** NetworkStream reads or writes bytes from a network socket.

**PipeStream:** PipeStream reads or writes bytes from different processes.

**CryptoStream:** CryptoStream is for linking data streams to cryptographic transformations.

The following diagram shows the hierarchy of stream classes:

**[](http://www.tutorialsteacher.com/Content/images/csharp/stream-heirarchy.png)Stream Classes Hierarchy**

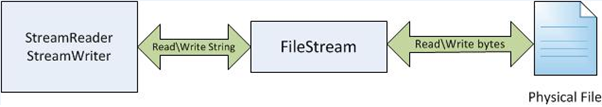
## Readers and Writers:

**StreamReader**: StreamReader is a helper class for reading characters from a Stream by converting bytes into characters using an encoded value. It can be used to read strings (characters) from different Streams like FileStream, MemoryStream, etc.

**StreamWriter**: StreamWriter is a helper class for writing a string to a Stream by converting characters into bytes. It can be used to write strings to different Streams such as FileStream, MemoryStream, etc.

**BinaryReader**: BinaryReader is a helper class for reading primitive datatype from bytes.

**BinaryWriter**: BinaryWriter writes primitive types in binary.

**[](http://www.tutorialsteacher.com/Content/images/csharp/stream-relations.png)Stream IO**

The above image shows that FileStream reads bytes from a physical file and then StreamReader reads strings by converting those bytes to strings. In the same way, StreamWriter takes a string and converts it into bytes and writes to FileStream and then FileStream writes the bytes to a physical file. So FileStream deals with bytes where as StreamReader & StreamWriter deals with strings.

### Points to Remember:

1. **Stream** is an abstract class for transferring bytes from different sources. It is base class for all other class that reads\writes bytes to different sources.
2. **FileStream** class provides reading and writing functionality of bytes to physical file.
3. Reader & writer classes provides functionality to read bytes from Stream classes (FileStream, MemoryStream etc) and converts bytes into appropriate encoding.
4. StreamReader provides a helper method to read string from FileStream by converting bytes into strings. StreamWriter provides a helper method to write string to FileStream by converting strings into bytes.

# Generics in C#

Generics introduced in C# 2.0. Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.

## Advantages of Generic:

1. Increases the reusability of the code.
2. Generic are type safe. You get compile time errors if you try to use a different type of data than the one specified in the definition.
3. Generic has a performance advantage because it removes the possibilities of boxing and unboxing.

### Points to Remember:

1. Generics denotes with angel bracket <>.
2. Compiler apply specified type for generics at compile time.
3. Generics can be applied to interface, abstract class, method, static method, property, event, delegate and operator.
4. Generics performs faster by not doing boxing & unboxing.

# Constraints in Generics:

C# includes **Constraints** to specify which type of placeholder type with the generic class is allowed. It will give a compile time error if you try to instantiate a generic class using a placeholder type that is not allowed by a constraints. For example, if the generic constraints specifies that only reference type can be used with the generic class then you cannot use value type to create an object of generic type.

Constraints can be applied using the **where** keyword.

|  |
| --- |
| class GenericClass<T> where T: class  {  } |

## Multiple constraints:

A generic class can have multiple constraints as shown below.

|  |
| --- |
| class GenericClass<T, U> where T: class where U: struct  {  } |

### Points to Remember:

1. Constraints specifies the kind of types allowed with the generics.
2. Constraints can be applied using the where keyword.
3. Six types of constraints can be applied: class, struct, new(), base class name, interface and derived type.
4. Multiple constraints also can be applied.