**C# - In ties with windows applications:**

**In this section we discuss just how important some of C#’s functionalities are, and why it’s one of the best choices to use when it comes to windows applications.**

**Future of C#:**

Today, C# is not only a Windows development programming language but can be used to build Web applications, Windows store apps, and mobile apps including iOS and Android. C# can also do more than that. If you’ve not already read my article, I highly recommend going and reading What C# Can Do For You.  
  
At the Build 2016 event, Microsoft made several exciting announcements and one of them was integrating Xamarin as a part of Visual Studio “15” and beyond. Now C# developers can build iOS and Android apps that can spit out native iOS and Android code.  
  
...the future of C# is very bright, why?

* You can write C# in any editor you want.
* C# is open source now
* C# runs on Windows, Mac, and Linux
* C# can be used to build Windows client apps, Windows Store apps, iOS apps, and Android aps and can also be used to build backend and middle-tier frameworks and libraries.
* C# (via Roslyn, the C# engine):
* Supports all IDEs and editors
* All the linters and analysis tools
* All the fixing and refactoring and code generation tools
* All the scripting and all the REPLs
* C# 7 comes with new features including tuples, record types, and pattern matching.
* C# is young and evolving.
* Unlike other programming languages, C# is still young and evolving. Now being open sourced, C# is getting community involvement and new features are being decided by community.

**Abstraction in C#:**

The word abstract means a concept or an idea not associated with any specific instance. In programming we apply the same meaning of abstraction by making classes not associated with any specific instance. The abstraction is done when we need to only inherit from a certain class, but do not need to instantiate objects of that class. In such case the base class can be regarded as "Incomplete". Such classes are known as an "Abstract Base Class".  
  
**Abstract Base Class**  
  
There are some important points about Abstract Base Class

1. An Abstract Base class can not be instantiated; it means the object of that class can not be created.
2. Class having abstract keyword and having abstract keyword with some of its methods (not all) is known as an Abstract Base Class.
3. Class having Abstract keyword and having abstract keyword with all of its methods is known as pure Abstract Base Class.
4. The method of abstract class that has no implementation is known as "operation". It can be defined as abstract void method ();
5. An abstract class holds the methods but the actual implementation of those methods is made in derived class.

**Encapsulation in C#:**

The object oriented programming will give the impression very unnatural to a programmer with a lot of procedural programming experience. In Object Oriented programming Encapsulation is the first place. Encapsulation is the procedure of covering up of data and functions into a single unit (called class). An encapsulated object is often called an abstract data type. In this article let us see about it in a detailed manner.

**NEED FOR ENCAPSULATION**The need of encapsulation is to protect or prevent the code (data) from accidental corruption due to the silly little errors that we are all prone to make. In Object oriented programming data is treated as a critical element in the program development and data is packed closely to the functions that operate on it and protects it from accidental modification from outside functions.  
  
Encapsulation provides a way to protect data from accidental corruption. Rather than defining the data in the form of public, we can declare those fields as private. The Private data are manipulated indirectly by two ways. Let us see some example programs in C# to demonstrate Encapsulation by those two methods. The first method is using a pair of conventional accessor and mutator methods. Another one method is using a named property. Whatever be the method our aim is to use the data without any damage or change.

**ENCAPSULATION USING PROPERTIES**

Properties are a new language feature introduced with C#. Only a few languages support this property. Properties in C# helps in protect a field in a class by reading and writing to it. The first method itself is good but Encapsulation can be accomplished much smoother with properties.

**Inheritence in C#:**

Inheritance is one of the three foundational principles of Object-Oriented Programming (OOP) because it allows the creation of hierarchical classifications. Using inheritance you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.

In the language of C#, a class that is inherited is called a base class. The class that does the inheriting is called the derived class. Therefore a derived class is a specialized version of a base class. It inherits all of the variables, methods, properties, and indexers defined by the base class and adds its own unique elements.

**Polymorphism in C#:**

Polymorphism means the same operation may behave differently on different classes.

* Example of Compile Time Polymorphism: Method Overloading
* Example of Run Time Polymorphism: Method Overriding
* Example of Compile Time Polymorphism
* Method Overloading: Method with same name but with different arguments is called method overloading.
* Method Overloading forms compile-time polymorphism.

Note - By default functions are not virtual in C# and so you need to write “virtual” explicitly. While by default in Java each function are virtual.

**Delegates in C#:**

Delegate is one of the base types in .NET. Delegate is a class, which is used to create delegate at runtime.  
  
Delegate in C# is similar to a function pointer in C or C++. It's a new type of object in C#. Delegate is very special type of object as earlier the entire the object we used to defined contained data but delegate just contains the details of a method.  
  
**Need of delegate**  
  
There might be a situation in which you want to pass methods around to other methods. For this purpose we create delegate.  
  
A delegate is a class that encapsulates a method signature. Although it can be used in any context, it often serves as the basis for the event-handling model in C# but can be used in a context removed from event handling (e.g. passing a method to a method through a delegate parameter).  
  
One good way of understanding delegates is by thinking of a delegate as something that gives a name to a method signature.

**Delegate magic**  
  
In class we create its object, which is instance, but in delegate when we create instance that is also referred to as delegate (means whatever you do you will get delegate).  
  
Delegate does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.  
  
**Benefit of delegates**  
  
In simple words delegates are object oriented and type-safe and very secure as they ensure that the signature of the method being called is correct. Delegate helps in code optimization.  
  
**Types of delegates**

1. Singlecast delegates
2. Multiplecast delegates

Delegate is a class. Any delegate is inherited from base delegate class of .NET class library when it is declared. This can be from either of the two classes from System.Delegate or System.MulticastDelegate.

**Singlecast delegate**  
  
Singlecast delegate point to single method at a time. In this the delegate is assigned to a single method at a time. They are derived from System.Delegate class.  
  
**Multicast Delegate**  
  
When a delegate is wrapped with more than one method that is known as a multicast delegate.  
  
In C#, delegates are multicast, which means that they can point to more than one function at a time. They are derived from System.MulticastDelegate class.

**Collections in C#:**

“.NET” offers a variety of collections, such as ArrayList, Hashtable, queues, Dictionaries. Collections are abstractions of data algorithms. An ArrayList is an abstract dynamic array, a Hashtable collection abstracts a lookup table, a Queues collection abstracts queues and so on. In addition to that, collections implement the ICollection, IEnumerable and IClonable interfaces. The detailed specification for each collection is found under the System.Collection namespace.

**Exception Handling in C#:**

Exception handling is a built-in mechanism in .NET framework to detect and handle run time errors. The .NET framework contains many standard exceptions. The exceptions are anomalies that occur during the execution of a program. They can be because of user, logic or system errors. If a user (programmer) does not provide a mechanism to handle these anomalies, the .NET run time environment provides a default mechanism that terminates the program execution.  
  
C# provides the three keywords try, catch and finally to do exception handling. The try block encloses the statements that might throw an exception whereas catch handles an exception if one exists. The finally can be used for doing any clean-up process.

**File Handling in C#:**

The System.IO namespace provides four classes that allow you to manipulate individual files, as well as interact with a machine directory structure. The Directory and File directly extends System.Object and supports the creation, copying, moving and deletion of files using various static methods. They only contain static methods and are never instantiated. The FileInfo and DirecotryInfo types are derived from the abstract class FileSystemInfo type and they are typically, employed for obtaining the full details of a file or directory because their members tend to return strongly typed objects. They implement roughly the same public methods as a Directory and a File but they are stateful and the members of these classes are not static.

In the .NET framework, the System.IO namespace is the region of the base class libraries devoted to file based input and output services. Like any namespace, the System.IO namespace defines a set of classes, interfaces, enumerations, structures and delegates. The following table outlines the core members of this namespace.

**Reading and Writing to Files**  
  
Reading and writing operations are done using a File object.

**Stream**  
  
The .NET provides many objects such as FileStream, StreamReader/Writer, BinaryReader/Writer to read from and write data to a file. A stream basically represents a chunk of data flowing between a source and a destination. Stream provides a common way to interact with a sequence of bytes regardless of what kind of devices store or display the bytes.

**FileStream**A FileStream instance is used to read or write data to or from a file. In order to construct a FileStream, first we need a file that we want to access. Second, the mode that indicates how we want to open the file. Third, the access that indicates how we want to access a file. And finally, the share access that specifies whether you want exclusive access to the file.

The FileStream can read or write only a single byte or an array of bytes. You will be required to encode the System.String type into a corresponding byte array. The System.Text namespace defines a type named encoding that provides members that encode and decode strings to an array of bytes. Once encoded, the byte array is persisted to a file with the FileStream.Write() method. To read the bytes back into memory, you must reset the internal position of the stream and call the ReadByte() method. Finally, you display the raw byte array and the decoded string to the console.

**BinaryReader and BinaryWriter**  
The BinaryReader and Writer class allows you to read and write discrete data types to an underlying stream in a compact binary format. The BinaryWriter class defines a highly overloaded Write method to place a data type in the underlying stream.