Polymorphism and Inheritance (Early Binding/Compile Time Polymorphism)

1. C# recognizes the method by its parameters and not only by its name.
2. The return value/parameter type of a method is never the part of method signature if the names of the methods are same. So this is not polymorphism.
3. Modifiers such as static are not considered as part of method signature.
4. The signature of a method consists of its name, number and types of its formal parameters. The return type of a function is not part of the signature. Two methods can not have the same signature and also non-members cannot have the same name as members.
5. Parameter names should be unique. And also we can not have a parameter name and a declared variable name in the same function as same.
6. In case of pass by value, the value of the variable is passed and in the case of ref and out, the address of the reference is passed.
7. This params keyword can only be applied to the last argument of the method.So the n number of parameters can only be at the end.
8. C# is very smart to recognize if the penultimate argument and the params have the same data type.
9. Parameter array must be a single dimensional array.

<https://www.codeproject.com/Articles/771455/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Ear>

Polymorphism and Inheritance (Inheritance)

1) It is a mechanism of deriving new class from and old class that is pre-defined.

2) The old class is called the super class

3) The new class is called the sub class

4) Inheritance is used for code re-usability

5) It allows sub class to inherit the variables and methods of their super class

1. No one can stop a derived class to have a method with the same name already declared in its base class.
2. Derived classes get a first chance at execution, then the base class.
3. A reserved keyword named “base” can be used in derived class to call the base class method.
4. Inheritance does not work backwards.
5. Except constructors and destructors, a class inherits everything from its base class.
6. In inheritance in C#, custom classes cannot derive from special built in C# classes like System.ValueType, System.Enum, System.Delegate, System.Array, etc.
7. A class can only be derived from one class in C#. C# does not support multiple inheritance by means of class.
8. Circular dependency is not allowed in inheritance in C#. ClassX is derived from ClassW which was derived from ClassY and ClassY was again derived from ClassX, which caused circular dependency in three classes that is logically impossible.
9. We can equate an object of a base class to a derived class but not vice versa.
10. We cannot implicitly convert an int to char, but char can be converted to int.

# [Polymorphism and Inheritance (Dynamic Binding/Run Time Polymorphism)](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

1. In C#, we can equate an object of a base class to a derived class but not vice versa.
2. The override modifier is needed as the derived class methods will get first priority and be called upon.
3. These modifiers like new and override can only be used if the method in the base class is a virtual method. Virtual means that the base class is granting us permission to invoke the method from the derived class and not the base class. But, we have to add the modifier override if our derived class method has to be called.
4. If the base class object declared the method virtual and the derived class used the modifier override, the derived class method will get called. Otherwise, the base class method will get executed. Therefore for virtual methods, the data type created is decided at run time only.
5. All the methods not marked with virtual are non-virtual, and the method to be called is decided at compile time, depending upon the static data type of the object.
6. An override method is a method that has the override modifier included on it. This introduces a new implementation of a method. We can’t use the modifiers such as new, static or virtual along with override. But abstract is permitted.

# Polymorphism and Inheritance (All About Abstract Classes in C#)

1. We cannot create an object of abstract class using new keyword.
2. A class can be derived from an abstract class.
3. A object of a class derived from an abstract class can be created using new.
4. If we declare any method as abstract in our abstract class, then it’s the responsibility of the derived class to provide the body of that abstract method, unless a body is provided for that abstract method, we cannot create an object of that derived class.
5. When we override an abstract or a virtual method from a derived class, we cannot change the parameters passed to it or the return type of that overridden method .
6. An abstract class means that the class is incomplete and cannot be directly used. An abstract class can only be used as a base class for other classes to derive from.
7. If a class has even a single abstract method, then the class has to be declared abstract as well.
8. An abstract method also cannot use the modifiers such as static or virtual.
9. Virtual methods run slower that non virtual methods.
10. Abstract class cannot be sealed class.
11. Abstract class cannot be a static class.

# All About C# Access Modifiers (Public/Private/Protected/Internal/Sealed/Constants/Static and Readonly Fields)

1. The default access modifier is private for class members.
2. A class marked as internal can have its access limited to the current assembly only.
3. Namespaces as we see by default can have no accessibility specifiers at all. They are by default public and we cannot add any other access modifier including public again too.
4. A class can only be public or internal. It cannot be marked as protected or private. The default is internal for the class.
5. Members of a class can be marked with all the access modifiers, and the default access modifier is private.
6. Protected internal means that the derived class and the class within the same source code file can have access.
7. Between public and internal, public always allows greater access to its members.
8. Base class always allows more accessibility than the derived class.
9. The return values of a method must have greater accessibility than that of the method itself.
10. A class marked sealed can’t act as a base class to any other class.
11. Since we cannot derive from sealed classes, the code from the sealed classes cannot be overridden.
12. We need to initialize the const variable at the time we create it. We are not allowed to initialize it later in our code or program.
13. Like classes, const variables cannot be circular, i.e., they cannot depend on each other.
14. A const field of a reference type other than string can only be initialized with null.
15. One can only initialize a const variable to a compile time value, i.e., a value available to the compiler while it is executing.
16. A constant by default is static and we can’t use the instance reference, i.e., a name to reference a const. A const has to be static as no one will be allowed to make any changes to a const variable.
17. A const variable cannot be marked as static.
18. A variable in C# can never have an uninitialized value.
19. Static variables are always initialized when the class is loaded first. An int is given a default value of zero and a bool is given a default of False.
20. An instance variable is always initialized at the time of creation of its instance.
21. A static readonly field cannot be assigned to (except in a static constructor or a variable initializer).

# Understanding Enums in C# (A Practical Approach)

1. An enum represents for a constant number, and an enum type is known as a distinct type having named constants.
2. We can’t declare char as an underlying data type for enum objects because char stores Unicode characters, but enum objects data type can only be number.
3. An enum can’t be derived from any other type except that of type byte, sbyte, short, ushort, int, uint, long, or ulong.
4. By default, enum is a sealed class and therefore sticks to all the rules that a sealed class follows, so no class can derive from enum, i.e., a sealed type.
5. The enum type is implicitly derived from System.Enum and so we cannot explicitly derive it from System.Enum.
6. enum is also derived from three interfaces IComparable, IFormattable and IConvertible.
7. Numerous predefined conversion methods can be used to convert enum from one data type to another.
8. More than one enum members can be initialized a same constant value.
9. An enum acts as a constant, so its value cannot be changed once initialized.
10. The enumerator name 'value\_\_' is reserved and cannot be used.

# Properties in C# (A Practical Approach)

* The variable used for property should be of same data type as the data type of the property.
* A property cannot have a void return type.
* If one do not mark property defined in derived class as override, it will by default be considered as new.
* You cannot override an accessor that is not defined in a base class abstract property.
* Get accessor is only used to read a property value. A property having only get cannot be set with any value from the caller.

| **Point of difference** | **Variable** | **Property** |
| --- | --- | --- |
| Declaration | Single declaration statement | Series of statements in a code block |
| Implementation | Single storage location | Executable code (property procedures) |
| Storage | Directly associated with variable's value | Typically has internal storage not available outside the property's containing class or moduleProperty's value might or might not exist as a stored element 1 |
| Executable code | None | Must have at least one procedure |
| Read and write access | Read/write or read-only | Read/write, read-only, or write-only |
| Custom actions (in addition to accepting or returning value) | Not possible | Can be performed as part of setting or retrieving property value |

# Indexers in C# (A Practical Approach)

* Dictionary in C# largely uses indexers to have a staring parameter as an indexer argument.
* Classes like Array List and List use indexers internally to provide functionality of arrays for fetching and using the elements.

| **Property** | **Indexer** |
| --- | --- |
| Allows methods to be called as if they were public data members. | Allows elements of an internal collection of an object to be accessed by using array notation on the object itself. |
| Accessed through a simple name. | Accessed through an index. |
| Can be a static or an instance member. | Must be an instance member. |
| A [get](https://msdn.microsoft.com/en-us/library/ms228503.aspx) accessor of a property has no parameters. | A get accessor of an indexer has the same formal parameter list as the indexer. |
| A [set](https://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor of a property contains the implicit value parameter. | A set accessor of an indexer has the same formal parameter list as the indexer, and also to the [value](https://msdn.microsoft.com/en-us/library/a1khb4f8.aspx) parameter. |
| Supports shortened syntax with [Auto-Implemented Properties (C# Programming Guide)](https://msdn.microsoft.com/en-us/library/bb384054.aspx). | Does not support shortened syntax. |

# Understanding Events in C#

# <https://www.codeproject.com/Articles/1009930/Learning-Csharp-Day-Understanding-Events-in-Csharp>

# Delegates in C#

# <https://www.codeproject.com/Articles/1153164/Learning-Csharp-Day-Delegates-in-Csharp-A-Practica>

# Events in C#

# <https://www.codeproject.com/Articles/1164409/Learning-Csharp-Day-Events-in-Csharp-A-Practical#_Toc471402084>