| Feature | Interface | Abstract class |
| --- | --- | --- |
| Implemenation | No idea about implementation | Partial Implementation is known |
| Method | Abstract methods | Abstract and concrete methods |
| Access Modifiers | Only be public but no need to specify | Can use any access modifier |
| Adding functionality (Versioning) | If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method. | If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly. |
| Fields and Constants | No fields can be defined in interfaces | An abstract class can have fields and constrants defined |

.

the difference is that Dictionary is concrete implementation while IDictionary is just a contract, abstraction.  
  
It is recommended for example to expect as argument an IDictionary rather that concrete Dictionary, or to expose property of IDictionary rather that Dictionary, because this promotes loose coupling.

Strongly typed – need to specify type of variable before using it.

type safe – means operations allowed on that type are only executed. Two bool variables cannot be summed.

|  |  |
| --- | --- |
| Dispose | Finalize |
| It is used to free unmanaged resources like files, database connections etc. at any time. | It can be used to free unmanaged resources (when you implement it) like files, database connections etc. held by an object before that object is destroyed. |
| Explicitly, it is called by user code and the class which is implementing dispose method, must has to implement IDisposable interface. | Internally, it is called by Garbage Collector and cannot be called by user code. |
| It belongs to IDisposable interface. | It belongs to Object class |
| It's implemented by implementing IDisposable interface Dispose() method. | It's implemented with the help of destructor in C++ & C#. |
| There is no performance costs associated with Dispose method. | There is performance costs associated with Finalize method since it doesn't clean the memory immediately and called by GC automatically. |

| **Array** | **ArrayList** |
| --- | --- |
| Array is strongly typed. This means that an array can store only specific type of items\elements. | ArrayList can store any type of items\elements. |
| Array stores fixed number of elements. Size of an Array must be specified at the time of initialization. | ArrayList grows automatically and you don't need to specify size. |
| No need to cast elements of an array while retriving because it is strongly type and stores specific type of items only. | Items of ArrayList need to be cast to appropriate data type  while retriving. |
| Use static helper class Array to perform different tasks on the array. | ArrayList itself includes various utility methods for various tasks. |

|  |  |
| --- | --- |
|  | [typeof](http://msdn.microsoft.com/en-us/library/58918ffs.aspx) is an operator to obtain a type known at *compile-time* |

[GetType()](http://msdn.microsoft.com/en-us/library/system.object.gettype.aspx) is a method you call on individual objects, to get the execution-time type of the object. Both return result derived from System.Type

Int32.Parse(string s) method converts the string representation of a number to its 32-bit signed integer equivalent.

* When s is a **null reference**, it will **throw ArgumentNullException**.
* If s is **other than integer** value, it will **throw FormatException**.
* When s represents **a number less than MinValue or greater than MaxValue**, it will **throw OverflowException**.

Convert.ToInt32(string s) method converts the specified string representation of 32-bit signed integer equivalent. This calls in turn Int32.Parse() method.

* When s is a **null reference**, it will **return 0** rather than throw ArgumentNullException.
* If s is **other than integer** value, it will **throw FormatException**.
* When s represents **a number less than MinValue or greater than MaxValue**, it will **throw OverflowException**.

**Int32.TryParse(string, out int) -> Recommended Method**

Int32.Parse(string, out int) method converts the specified string representation of 32-bit signed integer equivalent to out variable, and returns true if it is parsed successfully, false otherwise. This method is available in C# 2.0.

* When s is a **null reference**, it will **return 0** rather than throw ArgumentNullException.
* If s is **other than an integer** value, the out variable will **have 0** rather than FormatException.
* When s represents **a number less than MinValue or greater than MaxValue**, the out variable will **have 0** rather than OverflowException.

Well, there are two types of Polymorphism as stated below:   
1. Static Polymorphism (Early binding)   
2. Dynamic Polymorphism (Late binding)

**1. Static Polymorphism(Early Binding):**

Static Polymorphism is also know as Early Binding and Compile time Polymorphism. **Method Overloading** and **Operator Overloading** are examples of the same.

    It is known as Early Binding because the compiler is aware of the functions with same name and also which overloaded function is tobe called is known at compile time.

For exa:

public class Test()

{

    public Test()

   {

   }

  public int add(int no1, int no2)

     {

     }

public int add(int no1, int no2, int no3)

     {

     }

}

class Program  
    {  
        static void Main(string[] args)  
        {  
            Test tst = new Test ();  
             int sum = an.add(10,20);

**// here in above statement compiler is aware at compile time that need to call function add(int no1, int no2), hence it is called early binding and it is fixed so called static binding.**  
        }  
    }  
  
**2. Dynamic Polymorphism(Late Binding):**

 public class Animal  
    {  
        public virtual void MakeSound()  
        {  
            Console.WriteLine("Animal sound");  
        }  
    }  
    public class Dog:Animal  
    {  
        public override void MakeSound()  
        {  
            Console.WriteLine("Dog sound");  
        }  
          
    }  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Animal an = new Dog();  
            an.MakeSound();             
            Console.ReadLine();  
        }  
    }

  As in the above code , as any other call to a virtual method, will be compiled to a callvirt IL instruction. This means that the actual method that gets called is determined at run-time (unless the JIT can optimize some special case), but the compiler checked that the method exists, it chose the most appropriate overload (if any) and it has the guarantee that the function pointer will exist at a well-defined location in the vtable of the type (even though that is an implementation detail). The process of resolving the virtual call is extremely fast (you only need to dereference a few pointers), so it doesn't make much of a difference.

Iterators

An iterator can be used to step through collections such as lists and arrays.  An iterator method uses the [yield return](https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx) statement to return each element one at a time. When a yield return statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called. The return type of an iterator method or get accessor can be [IEnumerable](https://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx), [IEnumerable<T>](https://msdn.microsoft.com/en-us/library/9eekhta0.aspx), [IEnumerator](https://msdn.microsoft.com/en-us/library/system.collections.ienumerator.aspx), or [IEnumerator<T>](https://msdn.microsoft.com/en-us/library/78dfe2yb.aspx).

1. IEnumerable<**string**> iEnumerableOfString = (IEnumerable<**string**>)Month;
3. //If we want to retrieve all the items from **this** IEnumerable **object**, we can use a **foreach** loop.
5. **foreach**(**string** AllMonths **in** iEnumerableOfString)
6. {
7. Console.WriteLine(AllMonths);
8. }

IEnumerable uses IEnumerator internally.

1. IEnumerator<**string**> iEnumeratorOfString = Month.GetEnumerator();//to convert list into IEnumerator we can invoke the GetEnumerator method
3. //To retrieve all the items from the above IEnumerator **object**, we cannot use **foreach** loop instead of that we need to invoke MoveNext() Boolean method.
4. **while**(iEnumeratorOfString.MoveNext())
5. {
6. Console.WriteLine(iEnumeratorOfString.Current);
7. }

So, if you want to loop sequentially through the collection, use an IEnumerable interface else if you want to retain the cursor position and want to pass it from one function to another function then use an IEnumerator interface.

| **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](https://msdn.microsoft.com/en-us/library/bb384054.aspx). | Does not support shortened syntax. |

|  |  |
| --- | --- |
| **Shadowing** | **Overriding** |
| Shadowing is a VB.Net concept. It also known as method hiding in C#. Using this concept we can provide a new implementation for the base class method without overriding it. | Overriding allows us to re-write a base class function with a different definition. |
| Using the “new” keyword we can do the shadowing or method hiding. | C# uses the virtual/abstract and override keyword for method overriding. |
| Shadowing redefines an entire method or function. | Overriding redefines only the implementation of a method or function. |
| Showing is used to protect against subsequent base class modification. | Overriding does polymorphism by defining a different implementation. |
| We can change the access modifier. | We cannot change the access modifier. The access modifier must be the same as in the base class method or function. |
| There is no control of a base class on shadowing. In other words, a base class element cannot enforce or stop shadowing. | The base class has some control over the overriding. Using the keyword abstract, the base class forces the child (derived) class to implement the function or method. |
| Shadowing an element (function method or property) can be inherited further in a child (derived) class. The shadowed element is still hidden. | The same as shadowing, overriding an element is inherited further in a derived class and the overridden element is still overridden. |
| In shadowing, the signature of an element could be different. | In overriding, the signature of the element must be the same. |
| In shadowing, the base class cannot access the newly created child (derived) class method. This is because the base class has the same name of the element. | In concept, the base class can be accessed using the child object's overridden method. |

Reflection Dynamic

Inspect (meta-data) Yes No

Invoke public members Yes Yes

Invoke private members Yes No

Caching No Yes

Static class Yes No

#### Dictionary:

* It returns error if we try to find a key which does not exist.
* It is faster than a Hashtable because there is no boxing and unboxing.
* Only public static members are thread safe.
* Dictionary is a generic type which means we can use it with any data type.

#### Hashtable:

* It returns null if we try to find a key which does not exist.
* It is slower than dictionary because it requires boxing and unboxing.
* All the members in a Hashtable are thread safe,
* Hashtable is not a generic type,

The information stored in a type can include the following:

* The storage space that a variable of the type requires.
* The maximum and minimum values that it can represent.
* The members (methods, fields, events, and so on) that it contains.
* The base type it inherits from.
* The location where the memory for variables will be allocated at run time.
* The kinds of operations that are permitted.

System.Object

Ref types – classes,arrays,delegates,string

int i = 123;

// The following line boxes i.

object o = i;

o = 123;

i = (int)o;

value type on stack and reference type on heap.

the as operator is more efficient because it actually returns the cast value if the cast can be made successfully. The is operator returns only a Boolean value.

void UseIsOperator(Animal a)

{

if (a is Mammal)

{

Mammal m = (Mammal)a;

m.Eat();

}

}

void UseAsOperator(object o)

{

Mammal m = o as Mammal;

if (m != null)

{

Console.WriteLine(m.ToString());

}

else

{

Console.WriteLine("{0} is not a Mammal", o.GetType().Name);

}

}

Array is object (Array.Length)

A string is an object of type [String](https://msdn.microsoft.com/en-us/library/system.string.aspx) whose value is text. Internally, the text is stored as a sequential read-only collection of [Char](https://msdn.microsoft.com/en-us/library/system.char.aspx) objects.

String objects are immutable: they cannot be changed after they have been created.

All of the [String](https://msdn.microsoft.com/en-us/library/system.string.aspx) methods and C# operators that appear to modify a string actually return the results in a new string object.

In the following example, when the contents of s1 and s2 are concatenated to form a single string, the two original strings are unmodified

string s1 = "Hello ";

string s2 = s1;

s1 += "World";

System.Console.WriteLine(s2);

Verbatime string filePath = @"C:\Users\scoleridge\Documents\";

Nullable types are instances of the [System.Nullable<T>](https://msdn.microsoft.com/en-us/library/b3h38hb0.aspx) struct.

Nullable types have the following characteristics:

* Nullable types represent value-type variables that can be assigned the value of null. You cannot create a nullable type based on a reference type. (Reference types already support the null value.)
* The syntax T? is shorthand for [Nullable<T>](https://msdn.microsoft.com/en-us/library/b3h38hb0.aspx), where T is a value type. The two forms are interchangeable.
* Assign a value to a nullable type just as you would for an ordinary value type, for example int? x = 10; or double? d = 4.108. A nullable type can also be assigned the value null: int? x = null.
* Use the [Nullable<T>.GetValueOrDefault](https://msdn.microsoft.com/en-us/library/72cec0e0.aspx) method to return either the assigned value, or the default value for the underlying type if the value is null, for example int j = x.GetValueOrDefault();
* Use the [HasValue](https://msdn.microsoft.com/en-us/library/sksw8094.aspx) and [Value](https://msdn.microsoft.com/en-us/library/ydkbatt6.aspx) read-only properties to test for null and retrieve the value, as shown in the following example: if(x.HasValue) j = x.Value;
  + The HasValue property returns true if the variable contains a value, or false if it is null.
  + The Value property returns a value if one is assigned. Otherwise, a [System.InvalidOperationException](https://msdn.microsoft.com/en-us/library/system.invalidoperationexception.aspx) is thrown.
  + The default value for HasValue is false. The Value property has no default value.
  + You can also use the == and != operators with a nullable type, as shown in the following example: if (x != null) y = x;
* Use the ?? operator to assign a default value that will be applied when a nullable type whose current value is null is assigned to a non-nullable type, for example int? x = null; int y = x ?? -1;
* Nested nullable types are not allowed. The following line will not compile: Nullable<Nullable<int>> n;

T? variable

T is the underlying type of the nullable type. T can be any value type including struct; it cannot be a reference type because reference type already support nullable types.

String objects are immutable: they cannot be changed after they have been created.

Console.WriteLine(typeof(string).IsClass); // true

String is a reference type.

It can't be a value-type, as value-types need a known size for the stack etc. As a reference-type, the size of the **reference** is known in advance, even if the size of the string isn't.

When you perform comparisons with nullable types, if the value of one of the nullable types is null and the other is not, all comparisons evaluate to false except for != (not equal).

int? num1 = 10;

int? num2 = null;

num1 >= num2 false value cannot be compared to null

num1 != num2 true because num1 is int type and num2 is null value with int type.

The ??(Null Colaescing) operator defines a default value that is returned when a nullable type is assigned to a non-nullable type.

int? c = null;

// d = c, unless c is null, in which case d = -1.

int d = c ?? -1;

Important  
To use a bool? in a conditional statement, first check its [HasValue](https://msdn.microsoft.com/en-us/library/sksw8094.aspx) property to ensure that its value is not null, and then cast it to bool.

Value Types from System.ValueType and stored on stack.

* [Structs](https://msdn.microsoft.com/en-us/library/ah19swz4.aspx)
* [Enumerations](https://msdn.microsoft.com/en-us/library/sbbt4032.aspx)

class ImplDerivedClass: BaseClass, IFace1 { }

Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or used to store values of another type unless that type is convertible to the variable's type.

int i;

i = "Hello"; // Error: "Cannot implicitly convert type 'string' to 'int'"

// Implicit conversion. num long can

// hold any value an int can hold, and more!

int num = 2147483647;

long bigNum = num;

if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a cast.

double x = 1234.7;

int a;

// Cast double to int.

a = (int)x;

Anonymous types are [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) types that derive directly from [object](https://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx), and that cannot be cast to any type except [object](https://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx).

two instances of the same anonymous type are equal only if all their properties are equal.

To concatenate string variables, you can use the + or += operators, or the [String.Concat](https://msdn.microsoft.com/en-us/library/system.string.concat.aspx), [String.Format](https://msdn.microsoft.com/en-us/library/system.string.format.aspx) or [StringBuilder.Append](https://msdn.microsoft.com/en-us/library/system.text.stringbuilder.append.aspx) methods.

Decimal pricePerOunce = 17.36m;

String s = String.Format("The current price is {0} per ounce.",

pricePerOunce);

The [StringBuilder](https://msdn.microsoft.com/en-us/library/system.text.stringbuilder.aspx) class is found in the [System.Text](https://msdn.microsoft.com/en-us/library/system.text.aspx) namespace.

If you set the **Length** property to a value that is greater than the **Capacity** property, the **Capacity** property is automatically changed to the same value as the **Length** property. Setting the **Length** property to a value that is less than the length of the string within the current **StringBuilder** shortens the string.

Cw(default(int)); //0

Test a = new Test() { Num = 1, Str = "Hi" };

Test b = new Test() { Num = 1, Str = "Hi" };

bool areEqual = System.Object.ReferenceEquals(a, b);

// False:

b = a;

// Repeat calls with different results.

areEqual = System.Object.ReferenceEquals(a, b);

// True:

The explicit keyword declares a user-defined type conversion operator that must be invoked with a cast.

public static explicit operator Fahrenheit(Celsius c)

{

return new Fahrenheit((9.0f / 5.0f) \* c.degrees + 32);

}

Fahrenheit fahr = new Fahrenheit(100.0f);

Console.Write("{0} Fahrenheit", fahr.Degrees);

Celsius c = (Celsius)fahr;

The implicit keyword is used to declare an implicit user-defined type conversion operator.

public static implicit operator double(Digit d)

{

return d.val;

}

Digit dig = new Digit(7);

//This call invokes the implicit "double" operator

double num = dig;

|  |  |
| --- | --- |
| Const | Readonly |
| Constants can be assigned values only at the time of declaration | Read only variables can be assigned values either at runtime or at the time of instance initialization via constructor |
| Constant variables have to be accessed using "Classname.VariableName" | Read only variables have to be accessed using the "InstanceName.VariableName" |
| Constants are known at compile time | Read only variables are known at run time. |
|  |  |

The is operator in C# is used to check the object type and it returns a bool value: **true** if the object is the same type and **false** if not.

For null objects, it returns false.

The as operator does the same job of is operator but the difference is instead of bool, it returns the **object** if they are compatible to that type, else it returns null.

“==” compares if the object references are the same while “.Equals()” compares if the contents are the same.

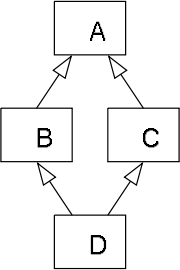
| **var** | **dynamic** |
| --- | --- |
| Introduced in **C# 3.0** | Introduced in **C# 4.0** |
| **Statically typed** – This means the type of variable declared is decided by the compiler at compile time. | **Dynamically typed** - This means the type of variable declared is decided by the compiler at runtime time. |
| **Need** to initialize at the time of declaration.  e.g., var str=”I am a string”;  Looking at the value assigned to the variable str, the compiler will treat the variable str as string. | **No** **need** to initialize at the time of declaration.  e.g., dynamic str;  str=”I am a string”; //Works fine and compiles  str=2; //Works fine and compiles |
| **Errors are caught at compile time.**  Since the compiler knows about the type and the methods and properties of the type at the compile time itself | **Errors are caught at runtime**  Since the compiler comes to about the type and the methods and properties of the type at the run time. |
| **Visual Studio shows intellisense** since the type of variable assigned is known to compiler. | **Intellisense is not available** since the type and its related methods and properties can be known at run time only |
| **e.g., var obj1;**  **will  throw a compile error since the variable is not initialized. The compiler needs that this variable should be initialized so that it can infer a type from the value.** | **e.g., dynamic obj1;**  **will compile;** |
| **e.g. var obj1=1;**  **will compile**  **var obj1=” I am a string”;**  **will throw error since the compiler has already decided that the type of obj1 is System.Int32 when the value 1 was assigned to it. Now assigning a string value to it violates the type safety.** | **e.g. dynamic obj1=1;**  **will compile and run**  **dynamic obj1=” I am a string”;**  **will compile and run since the compiler creates the type for obj1 as System.Int32 and then recreates the type as string when the value “I am a string” was assigned to it.**  **This code will work fine.** |

|  |
| --- |
|  |
| **System.Array.CopyTo()** | **System.Array.Clone()** |
| It performs a deep copy of the array. | It performs a shallow copy of the array. |
| It copies the elements of one array to another array. | It creates a copy of an array as an object. |
| It requires a destination array. | It does not require destination array. |

|  | **IEnumerable** | **IQueryable** |
| --- | --- | --- |
| *Namespace* | System.Collections Namespace | System.Linq Namespace |
| *Derives from* | No base interface | Derives from IEnumerable |
| [*Deferred Execution*](http://synvistech.com/blogs/deferred-execution-vs-lazy-loading-vs-eager-loading-vs-explicitly-loading/) | Supported | Supported |
| [*Lazy Loading*](http://synvistech.com/blogs/deferred-execution-vs-lazy-loading-vs-eager-loading-vs-explicitly-loading/) | Not Supported | Supported |
| *How does it work* | While querying data from database, IEnumerable executes select query on server side, load data in-memory on client side and then filter data. Hence does more work and becomes slow. | While querying data from database, IQueryable executes select query on server side with all filters. Hence does less work and becomes fast. |
| *Suitable for* | LINQ to Object and LINQ to XML queries | LINQ to SQL queries |
| *Custom Query* | Doesn’t support | Supports using CreateQuery and Executemethods |
| *Extension method* *parameter* | Extension methods supported in IEnumerable takes functional objects. | Extension methods supported in IEnumerable takes expression objects, i.e., expression tree. |
| *When to use* | When querying data from in-memory collections like List, Array, etc. | When querying data from out-memory (like remote database, service) collections. |
| *Best Uses* | In-memory traversal | Paging |

|  |  |
| --- | --- |
| **var** | **IEnumerable** |
| Use var type when you want to make a "custom" type on the fly. | Use IEnumerable when you already know the type of query result. |
| var is also good for remote collection since var is an IQueryable type that executes query in SQL server with all filters. | IEnumerable is good for in-memory collection. |
| **IEnumerable** | **IList** |
| Move forward only over a collection, it can’t move backward and between the items. | Used to access an element in a specific position/index in a list. |
| Doesn’t support add or remove items from the list. | Useful when you want to Add or remove items from the list. |
| Find out the no of elements in the collection after iterating the collection. | Find out the no of elements in the collection without iterating the collection. |
| Supports further filtering. | Doesn’t support further filtering. |

The "diamond problem" (sometimes referred to as the "deadly diamond of death"[[4]](https://en.wikipedia.org/wiki/Multiple_inheritance#cite_note-4)) is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have [overridden](https://en.wikipedia.org/wiki/Method_overriding_(programming)), and D does not override it, then which version of the method does D inherit: that of B, or that of C?



// This is not allowed

class A { void A() {} }

class B { void B() {} }

class C : A, B {}

// This is allowed

interface IA { void A(); }

interface IB { void B(); }

class A : IA, IB

{

public void A() {}

public void B() {}

}

Inteface forces any class to provide implementation of the method declare in them.

Object Pooling is something that tries to keep a pool of objects in memory to be re-used later and hence it will reduce the load of object creation to a great extent.

Connection Pooling

connectionStrings>

<add name="TrackingConnectionString" connectionString="Data Source=localhost;Min Pool Size=0;Max Pool Size=100;Pooling=true;  
 Initial Catalog=Tracking;Integrated Security=True;" providerName="System.Data.SqlClient"/>

</connectionStrings>

“ToDictionary” conversion operator is the instance of Dictionary (k, T). The “keySelector” predicate recognizes the key of each item, while “elementSelector”, is used to extract each single item, if it is given.

Extension method on “IEnumerable” is.AsEnumerable. AsEnumerable simply returns the source sequence as an object of type IEnumerable <T>.

The **volatile** keyword indicates that a field might be modified by multiple threads that are executing at the same time. Fields that are declared **volatile** are not subject to compiler optimizations that assume access by a single thread.

N Layer - Different LAyer on same computer

N tier - Different ties on differebt computers.