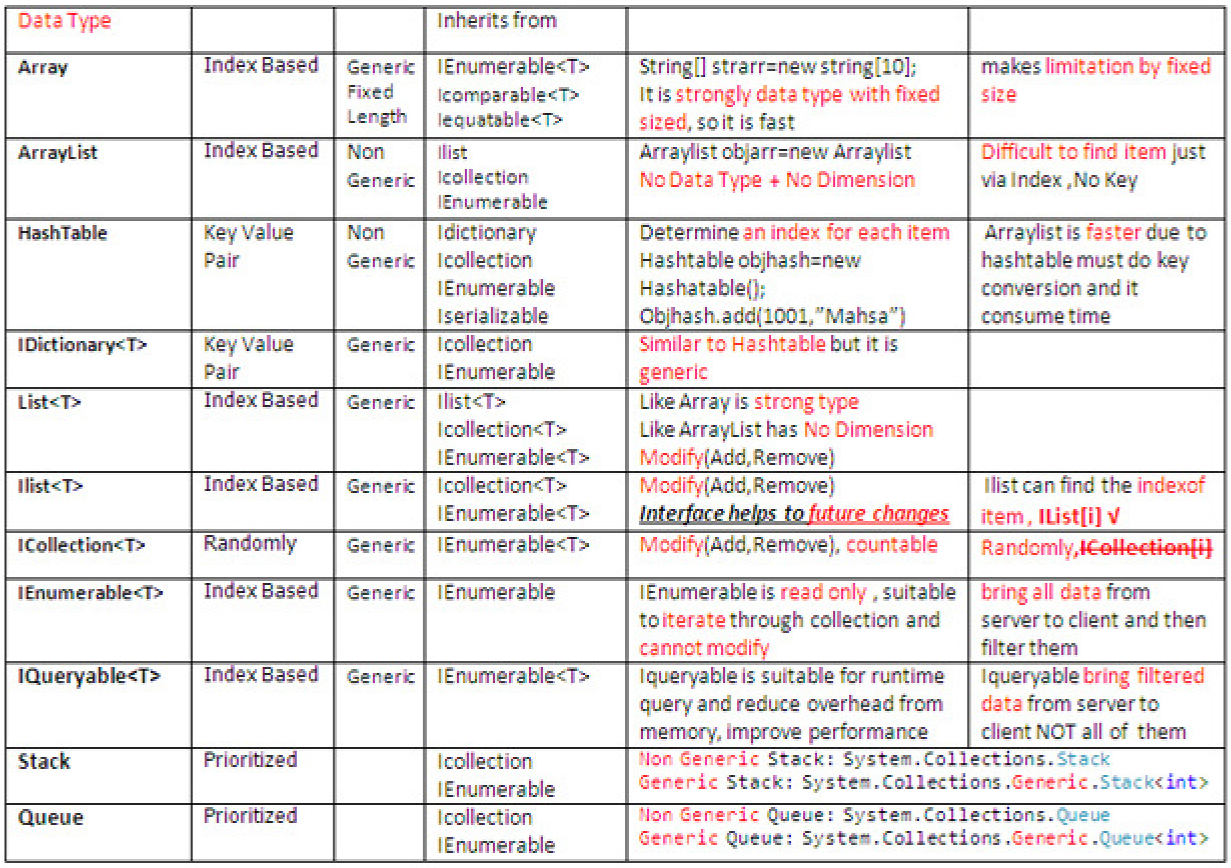
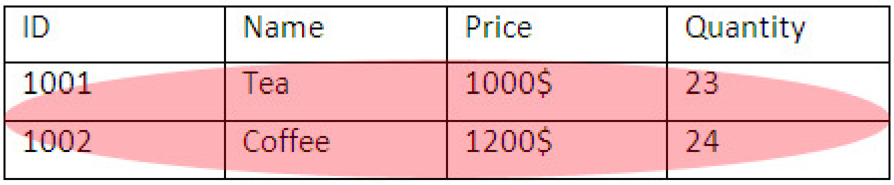
List vs IEnumerable vs IQueryable vs ICollection vs IDictionary



Data Type			Inherits from		
Array	Index Based	Generic Fixed Length	IEnumerable<T> Icomparable<T> Iequatable<T>	String[] strarr=new string[10]; It is strongly data type with fixed sized, so it is fast	makes limitation by fixed size
ArrayList	Index Based	Non Generic	Ilist Icollection IEnumerable	Arraylist objarr=new Arraylist No Data Type + No Dimension	Difficult to find item just via Index ,No Key
HashTable	Key Value Pair	Non Generic	Idictionary Icollection IEnumerable Iserializable	Determine an index for each item Hashtable objhash=new Hashatable(); Objhash.add(1001,"Mahsa")	Arraylist is faster due to hashtable must do key conversion and it consume time
IDictionary<T>	Key Value Pair	Generic	Icollection IEnumerable	Similar to Hashtable but it is generic	
List<T>	Index Based	Generic	Ilist<T> Icollection<T> IEnumerable<T>	Like Array is strong type Like ArrayList has No Dimension Modify(Add,Remove)	
Ilist<T>	Index Based	Generic	Icollection<T> IEnumerable<T>	Modify(Add,Remove) ***Interface helps to future changes***	Ilist can find the indexof item , **IList[i]** √
ICollection<T>	Randomly	Generic	IEnumerable<T>	Modify(Add,Remove), countable	Randomly,~~**ICollection[i]**~~
IEnumerable<T>	Index Based	Generic	IEnumerable	IEnumerable is read only , suitable to iterate through collection and cannot modify	bring all data from server to client and then filter them
IQueryable<T>	Index Based	Generic	IEnumerable<T>	Iqueryable is suitable for runtime query and reduce overhead from memory, improve performance	Iqueryable bring filtered data from server to client NOT all of them
Stack	Prioritized		Icollection IEnumerable	`Non Generic Stack: System.Collections.Stack` `Generic Stack: System.Collections.Generic.Stack<int>`	
Queue	Prioritized		Icollection IEnumerable	`Non Generic Queue: System.Collections.Queue` `Generic Queue: System.Collections.Generic.Queue<int>`	

Collection

Collection is set of related records. It includes a meaningful unit:
We should select appropriate container to store data temporarily for fetch and modification process.
Appropriate container depends on:
1. Our aim which we want to do on data (just reading, doing modification such as insert, delete, update)
2. Number of records which should be transferred



ID	Name	Price	Quantity
1001	Tea	1000$	23
1002	Coffee	1200$	24

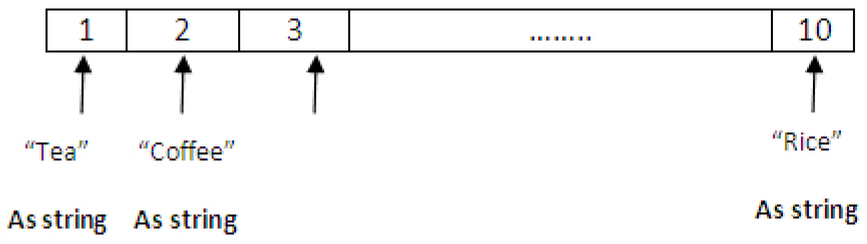
Array

1. Fixed Length -> Its size is not flexible. It is determined at instantiation time.

2. Strongly Typed -> Developers determine its type at the instantiation time instead of runtime. This feature makes to run fast at runtime, since it does not need to wait for type definition.

3. Developers use "foreach" keyword to fill and iterate through array.

Fixed Length and Strongly Typed consume less memory, therefore it has good performance.



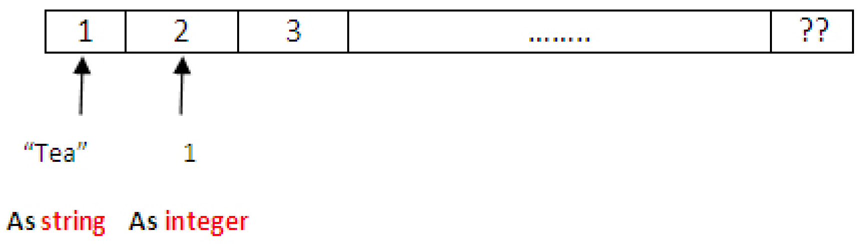
```
// It is obvious that strArray is
// 1. string   --> Strongly Type
// 2. Sized=10 --> Fixed Size

string[] strArray = new string[10];

for (int i = 0; i < 10; i++)
{
    if (strArray[i]==null)
    {
        strArray[i] = (i+1).ToString();
    }
}

this.ListBoxArray.DataSource = null;
this.ListBoxArray.Items.Clear();

this.ListBoxArray.DataSource = strArray;
this.ListBoxArray.DataBind();
```

ArrayList

1. Arraylist is NOT Fixed Length -> It is possible that data grows. On the one hand it is a good feature whenever developers are not sure about size of arraylist and on the other hand it might take long time for size definition.

2. Arraylist is NOT Strongly Typed -> Whenever developers are not sure about what is exactly type definition for input or output data and they should wait until runtime to appear its type. Its disadvantage is time consuming at the runtime for memory to determine type definition.

3. Developers use "foreach" keyword to fill and iterate through arraylist.



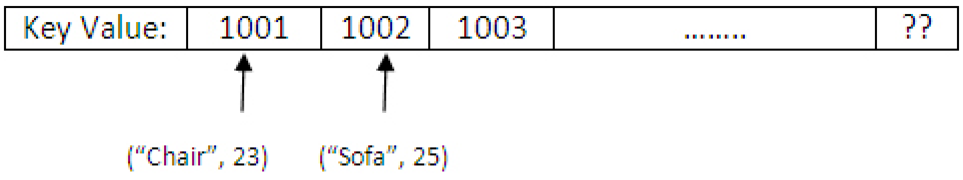
```
    public class Product
    {
        public Product()
        {
        }
        public Product(string Code, string Name)
        {
            _Code = Code;
            _Name = Name;
        }

        public string _Code {get; set;}
        public string _Name { get; set; }
    }
```

ArrayList can accept string, integer and decimal Simultaneously.

```
        // It is NOT obvious that strArrayList is 1. string? int? object? decimal?  --> NOT 
Strongly Type
        //                                       2. Sized=10? 20? 100?             -->NOT Fixed 
Size
        // Namespace: System.Collections

        System.Collections.ArrayList strArrayList = new System.Collections.ArrayList();
        //System.Linq.IQueryable  type of data is not specific runtime defered support
        strArrayList.Add("Mahsa");  //    "Mahsa": is string
        strArrayList.Add(1);        //         1 : is integer
        strArrayList.Add(0.89);     //      0.89: is decimal

        this.ListBoxArrayList.DataSource = null;
        this.ListBoxArrayList.Items.Clear();
        this.ListBoxArrayList.DataSource = strArrayList;
        this.ListBoxArrayList.DataBind();

        System.Text.StringBuilder str= new System.Text.StringBuilder();
        foreach (var item in strArrayList)
        {
            str.Append(" , "+item);
        }
        this.lblArrayList.Text = str.ToString();

       // Below is old way to fill obj from product , in Arraylist you need to create more than
one instance
       // Product objProduct = new Product();
       // objProduct.Code = "1001";
       // objProduct.Name = "Chair";

       // It is NOT obvious that strArrayList is
       // 1. string? int? object? decimal? OR OBJECT??  --> NOT Strongly Type
       // 2. Sized=10? 20? 100?                         -->NOT Fixed Size
       // Namespace: System.Collections

        System.Collections.ArrayList objArrayList = new System.Collections.ArrayList();
        objArrayList.Add(new Product("1001", "Chair"));
        objArrayList.Add(new Product("1002", "Sofa"));
        objArrayList.Add(new Product("1003", "Carpet"));

        this.DropDownListArrayListObject.DataSource = null;
        this.DropDownListArrayListObject.Items.Clear();
        this.DropDownListArrayListObject.DataSource = objArrayList;

        // Finding among Object of Array List is difficult, you have to find your specific item
by index
        Product objTemp = (Product)objArrayList[0];
        objArrayList.Remove(objTemp);

        this.DropDownListArrayListObject.DataTextField = "_Name";
        this.DropDownListArrayListObject.DataValueField = "_Code";
        this.DropDownListArrayListObject.DataBind();
        this.GridViewArrayListObject.DataSource = objArrayList;
        this.GridViewArrayListObject.DataBind();
```

HashTable

HashTable is another kind of data structure that defines key value for each data section. Therefore finding data is easy just by point out to its key. It is NOT strongly typed and NOT fixed size.

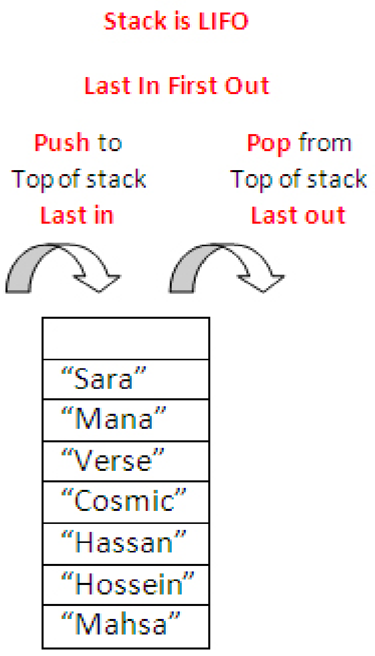


```
        // It is NOT obvious that strArrayList is
        // 1. string? int? object? decimal? OR OBJECT??  --> NOT Strongly Type
        // 2. Sized=10? 20? 100?                         -->NOT Fixed Size
        // Namespace: System.Collections
        // Hashtable solve the problem in Arraylist when we are looking for specific item
        // Hashtable dedicate a key for each item, then finding item is easier and faster

        System.Collections.Hashtable objHashTable = new System.Collections.Hashtable();

        objHashTable.Add("1001","Chair");
        objHashTable.Add("1002", "Sofa");
        objHashTable.Add("1003", "Carpet");


        this.DropDownListHashTable.DataSource = null;
        this.DropDownListHashTable.Items.Clear();
        this.DropDownListHashTable.DataSource = objHashTable;

        // finding item is easier you just need to point to it by call its key
        objHashTable.Remove("1002");
        //

        this.DropDownListHashTable.DataTextField = "Value";
        this.DropDownListHashTable.DataValueField = "Key";
        this.DropDownListHashTable.
```

Stack

We have different data structure and stack is one of them. Stack is subset of data structure. Stack is prioritized data structure (such as List is indexed base). Stack defines priority for each item, it means stack behavior force its items to put (push) inside stack prioritized form. So stack put later item on the top of items and this behavior is "defining priority for each item". Therefore whenever you want to insert item, you should add (PUSH) it at the top of the stack and whenever you want to remove (POP) item from stack you should remove it from top of the stack. As you got it, item that comes last one will be selected to POP for first one and its expression in computer science is equal to "Last in First out" == "LIFO".

It is NOT strongly typed and NOT fixed size.



Stack - Push

```
        // Stack is LIFO: Last in First Out
        System.Collections.Stack objStackPush = new System.Collections.Stack();

        // By Push method you can insert item at the top of the stack
        objStackPush.Push("Mahsa");
        objStackPush.Push("Hassankashi");
        this.lblPop.Text = "";
        this.ListBoxStack.DataSource = objStackPush.ToArray();
        this.ListBoxStack.DataBind();
```

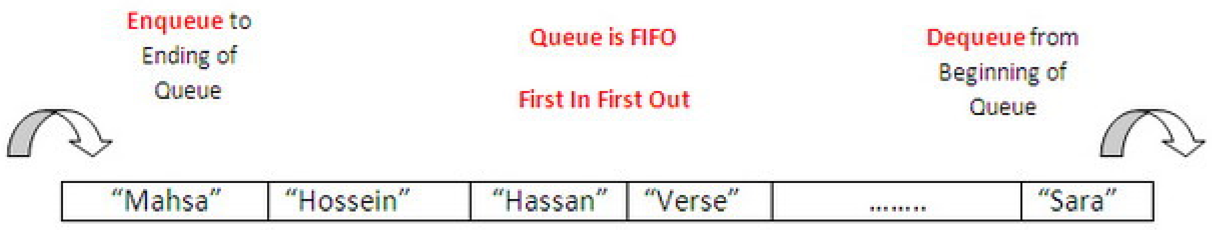
Stack - Pop

```
        System.Collections.Stack objStackPop = new System.Collections.Stack();
        objStackPop.Push("Mahsa");
        objStackPop.Push("Hassankashi");

        // By Pop method you can remove item from the top of the stack --> Last in First in
        this.lblPop.Text = objStackPop.Pop().ToString();
        this.ListBoxStack.DataSource = objStackPop.ToArray();
        this.ListBoxStack.DataBind();
```

Queue

Queue is another kind of data structure that defines priority for each item in other form. Therefore whenever you want to insert item, you should add (Enqueue) it at the head of the queue and whenever you want to remove (Dequeue) item from queue you should remove it from bottom of the queue. As you got it, item that comes first one will be selected to Dequeue for first one and its expression in computer science is equal to "First in First out" == "FIFO".

It is NOT strongly typed and NOT fixed size.



Queue - Enqueue

```
        // Queue is FIFO: First in First Out
        System.Collections.Queue objQueue = new System.Collections.Queue();

        // By Enqueue method you can insert item at the END of the Queue
        objQueue.Enqueue("Mahsa");
        objQueue.Enqueue("Hassankashi");
        objQueue.Enqueue("Cosmic");
        objQueue.Enqueue("Verse");

        this.lblQueue.Text = "";
        this.ListBoxQueue.DataSource = objQueue.ToArray();
        this.ListBoxQueue.DataBind();
```

Queue - Dequeue

```
        System.Collections.Queue objQueue = new System.Collections.Queue();

        objQueue.Enqueue("Mahsa");
        objQueue.Enqueue("Hassankashi");
        objQueue.Enqueue("Cosmic");
        objQueue.Enqueue("Verse");

        // By Dequeue method you can remove item from the BEGINING of the Queue --> First in 
First out FIFO
        this.lblQueue.Text=objQueue.Dequeue().ToString();
        this.ListBoxQueue.DataSource = objQueue.ToArray();
        this.ListBoxQueue.DataBind();
```

List

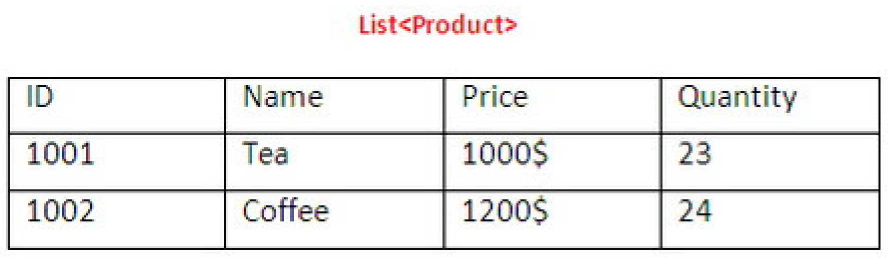
Why we need List?

1. List is NOT Fixed Length -> It is possible that data grows. On the one hand it is a good feature whenever developers are not sure about size of arraylist and on the other hand it might take long time for size definition.

2. List is Strongly Typed when it is defined "Generic" -> Whenever developers are sure about what is exactly type definition for input or output data and they do not wait until runtime to appear its type. This feature makes to run fast at runtime, since it does not need to wait for type definition.

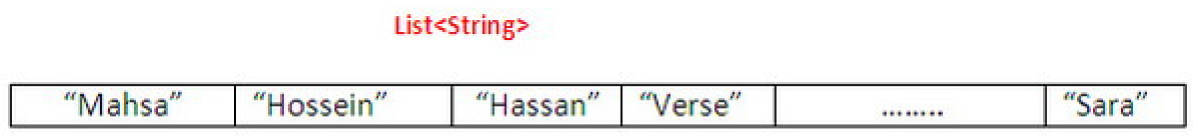
3. Developers use "foreach" keyword to fill and iterate through array.

Since List is not Fixed Length makes developers feel flexible to use it, and because of it is Strongly Typed when it is defined "Generic" so our code runs fast at runtime because it does not need to wait for type definition.



List<Product>

ID	Name	Price	Quantity
1001	Tea	1000$	23
1002	Coffee	1200$	24



List<String>

“Mahsa”	“Hossein”	“Hassan”	“Verse”	……..	“Sara”

```
        // Like Array is Strong Type
        // Like ArrayList with No Dimension
        System.Collections.Generic.List<string> strList = new List<string>();


        strList.Add("Mahsa");
        strList.Add("Hassankashi");
        strList.Add("Cosmic");
        strList.Add("Verse");

        this.ListBoxListGeneric.DataSource = strList;
        this.ListBoxListGeneric.DataBind();

        System.Text.StringBuilder str = new System.Text.StringBuilder();

        foreach (var item in strList)
        {
            str.Append(" , " + item);
        }
        this.lblList.Text = str.ToString();
```

IList

Why we need IList? IList is implemented by List, Ilist is an interface and implements methods. Whenever you estimate probability that your code would be changed in future you have to use IList because interface reduces dependency and with the little modification your code runs. Therefore you should observe polymorphism in order to decouple your app and control on adding or removing method that might be changed. Everything else is similar. Whenever we want to change on some operation, so “IList” allow us to do that easily with at least changing in the whole of codes.

Interfaces can not be instantiated, so it should be instantiate from List

```
System.Collections.Generic.IList<string> strIList = new List<string>();
```

Difference between Concrete Class and Interface

1. Concrete Class inherits from just ONE class but it can implement one or MORE than one interfaces

2. You can write inside concrete class full version of your function while you have to define just signature inside the interface.

3. You can define variable and value Inside Concrete Class while you are not allowed to define variable inside interfaces.

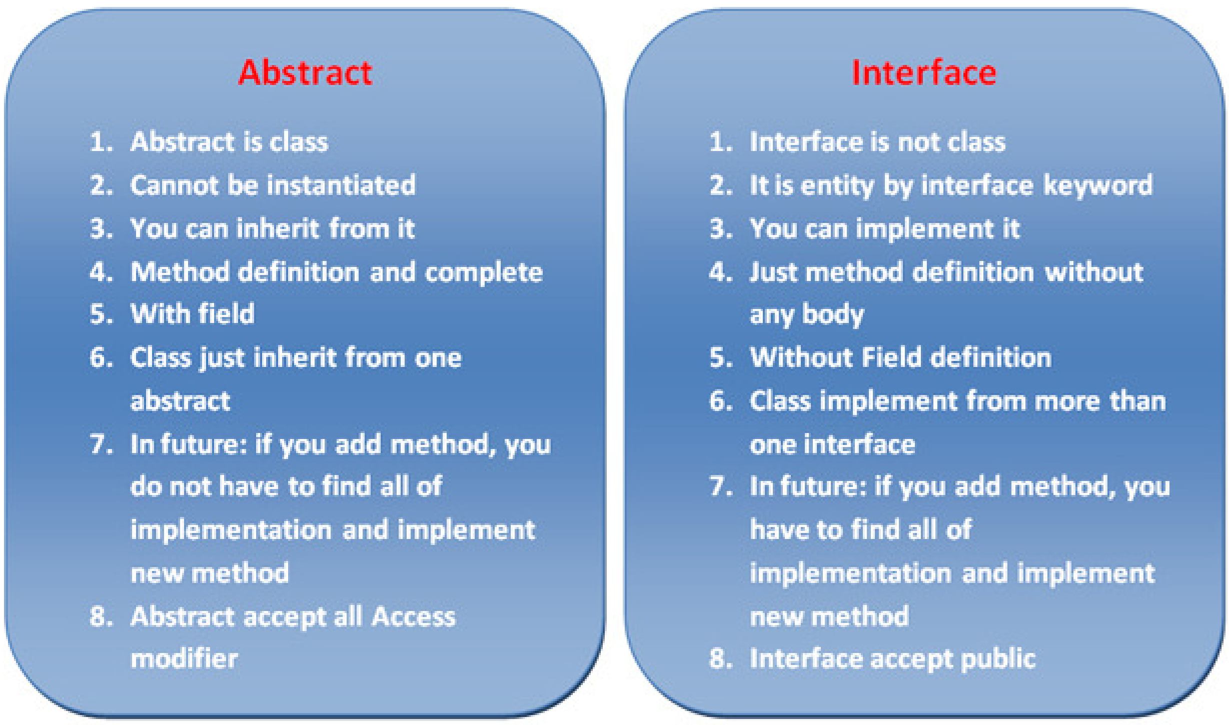
4. An Concrete class can have constructor while you are not allowed to define constructor inside interfaces.

5. Abstract class can contain access modifier while interfaces does not.

As I mentioned that how a **class cannot be driven from two class** it just can be driven from only one class, so whenever you want to drive from two class it is not possible to inherit from two abstract class but **it is possible to drive from more than one class by interfaces**.

Later in future if developers decide to add some features on their class and inherit it from another class, developers always prefer to use interface of collection so that if you want to change your code and enhance its abilities, choose interfaces.

On the other hand interface keeps your program **extensible** and **Decouple**: The classes are independent from each other, so error, exception and failure will happen rarely in future changing code by interfaces.



Abstract

1. Abstract is class
2. Cannot be instantiated
3. You can inherit from it
4. Method definition and complete
5. With field
6. Class just inherit from one abstract
7. In future: if you add method, you do not have to find all of implementation and implement new method
8. Abstract accept all Access modifier

Interface

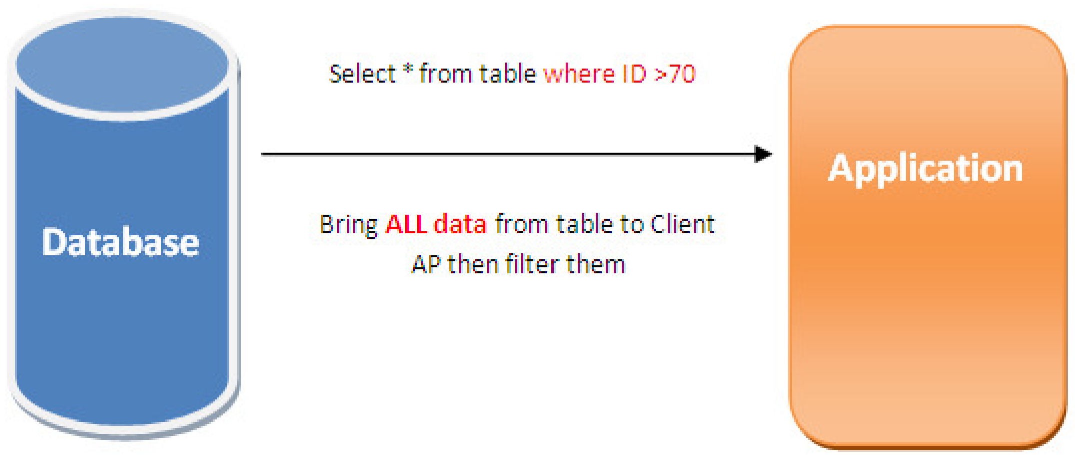
1. Interface is not class
2. It is entity by interface keyword
3. You can implement it
4. Just method definition without any body
5. Without Field definition
6. Class implement from more than one interface
7. In future: if you add method, you have to find all of implementation and implement new method
8. Interface accept public

Polymorphism: When you are using interface you absolutely observe and do polymorphism and oop. It means encapsulate your designer. Interface means a point or a node to join two part to each other it means making **low dependency** from two part and make a joint section to make flexible change in future.

```
// Ilist can not be instantiated from Ilist, so it should be instantiate from List
System.Collections.Generic.IList<string> strIList = new List<string>();

strIList.Add("Mahsa");
strIList.Add("Hassankashi");
strIList.Add("Cosmic");
strIList.Add("Verse");

this.ListBoxListGeneric.DataSource = strIList;
this.ListBoxListGeneric.DataBind();

System.Text.StringBuilder str = new System.Text.StringBuilder();

foreach (var item in strIList)
{
    str.Append(" , " + item);
}
this.lblList.Text = str.ToString();
```

IEnumerable

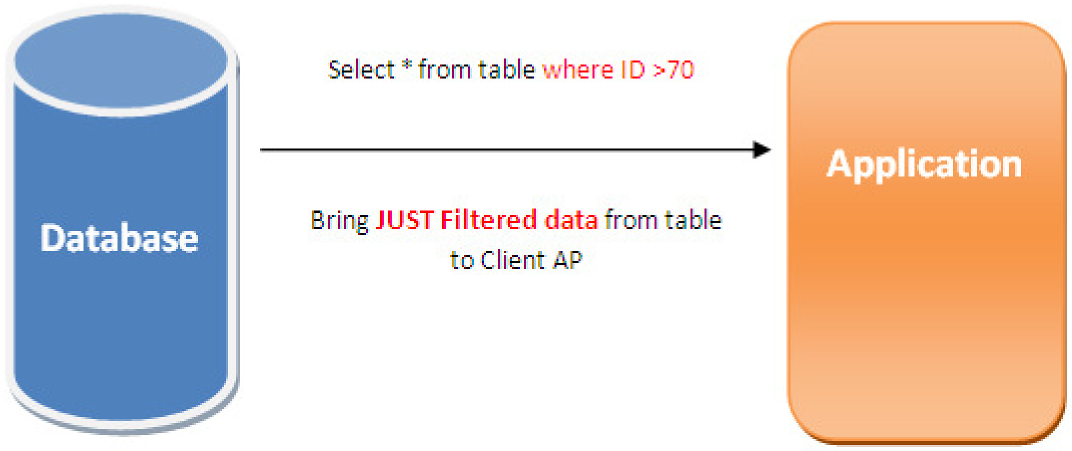
IEnumerable is suitable just for iterate through collection and you can not modify (Add or Remove) data IEnumerable bring ALL data from server to client then filter them, assume that you have a lot of records so IEnumerable puts overhead on your memory.


Database
Select * from table where ID >70
Bring ALL data from table to Client AP then filter them
Application

```
        // IEnumerable cannot be instantiated from Enumerable, so it should be instantiate from
List
        System.Collections.Generic.IEnumerable<Employee> empIEnumerable = new List<Employee>
        {   new Employee { ID = 1001, Name="Mahsa"},
            new Employee { ID = 1002, Name = "Hassankashi" },
            new Employee { ID = 1003, Name = "CosmicVerse" },
            new Employee { ID = 1004, Name = "Technical" }
        };

        this.GridViewIEnumerable.DataSource = empIEnumerable;
        this.GridViewIEnumerable.DataBind();

        System.Text.StringBuilder str = new System.Text.StringBuilder();

        foreach (Employee item in empIEnumerable)
        {
            str.Append(" , " + item.ID +"-"+item.Name);
        }

        this.lblIEnumerable.Text = str.ToString();
```

IQueryable

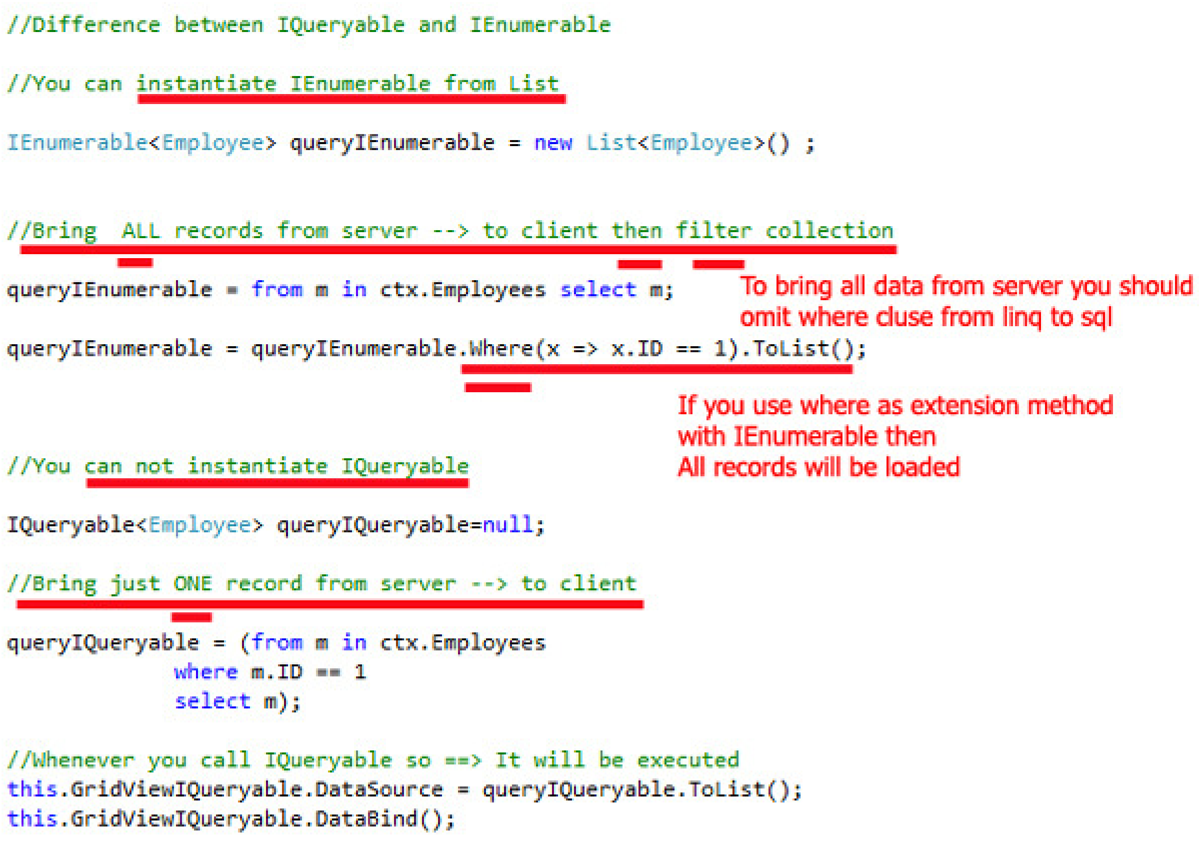
Whenever we encounter to huge data with so many records so we have to reduce overhead from application. IQueryable prepares high performance in such situations (huge data) by filtering data firstly and then sending filtered data to client.


Database
Select * from table where ID >70
Bring JUST Filtered data from table to Client AP
Application

Follow "Entity Framework DatabaseFirst" from this article for making DBContext

```
DataAccessEntities ctx = new DataAccessEntities();
var ctx = new DataAccessEntities();
```

Data Access



```
//Difference between IQueryable and IEnumerable

//You can instantiate IEnumerable from List

IEnumerable<Employee> queryIEnumerable = new List<Employee>() ;


//Bring  ALL records from server --> to client then filter collection

queryIEnumerable = from m in ctx.Employees select m;

queryIEnumerable = queryIEnumerable.Where(x => x.ID == 1).ToList();


//You can not instantiate IQueryable

IQueryable<Employee> queryIQueryable=null;

//Bring just ONE record from server --> to client

queryIQueryable = (from m in ctx.Employees
             where m.ID == 1
             select m);

//Whenever you call IQueryable so ==> It will be executed
this.GridViewIQueryable.DataSource = queryIQueryable.ToList();
this.GridViewIQueryable.DataBind();
```

To bring all data from server you should omit where cluse from linq to sql

If you use where as extension method with IEnumerable then All records will be loaded

```
     //Difference between IQueryable and IEnumerable

    //You can instantiate IEnumerable from List

    IEnumerable<employee> queryIEnumerable = new List<employee>() ;


    //Bring  ALL records from server --> to client then filter collection
    //To bring all data from server you should omit where cluse from linq to sql
    queryIEnumerable = from m in ctx.Employees select m;

    //If you use where as extension method with IEnumerable then All records will be loaded
    queryIEnumerable = queryIEnumerable.Where(x => x.ID == 1).ToList();



    //You can not instantiate IQueryable

    IQueryable<employee> queryIQueryable=null;

    //Bring just ONE record from server --> to client

    queryIQueryable = (from m in ctx.Employees
                       where m.ID == 1
                       select m);

    //Whenever you call IQueryable so ==> It will be executed
    this.GridViewIQueryable.DataSource = queryIQueryable.ToList();
    this.GridViewIQueryable.DataBind();
```