

# Interfaces. C# Collections.

*Net Core. 2017*

- Interface declaration
- Interface implementation
- Built-in .Net interfaces
- Task1
- C# Collections
- Task 2

# Interface declaration

An interface contains definitions for a group of related functionalities that a **class** or a **struct** can implement

```
modifieropt interface INameOfInterface: listOfInterfacesopt  
{  
    //declaration of interface members  
}
```

- ❑ **Interface** Includes only declaration of **method, properties, events, indexers**
- ❑ **Interface** can't contain **constants, fields, operators, instance constructors, destructors, or types.**
- ❑ **Interface members** are automatically **public**, and they can't include any access modifiers.
- ❑ **Interface members** **can't be static.**

# Interface declaration

method

indexer

property

event

```
interface IMyInterface
{
    void Process(int arg1, double
arg2);

    float this [int index] { get; set;
}

    string Name { get; set; }

event MouseEventHandler Mouse;
}
```

# Interface Implementation

```
interface IFighter
{
    void Punch(int side);
    void Kick (int side);
    void Block();
}
```

```
IFighter f = new Soldier();
f.Punch(Left);
f.Kick(Right);
f.Block();
```

```
IFighter s= new Dragon();
s.Punch(Left);
s.Kick(Right);
s.Block();
...
```

```
class Soldier : IFighter
{
    public void Punch(int side)
    {
        Move(arms[side], Forward);
    }
    public void Kick (int side)
    {
        Move(arms[side], Forward);
    }
}
```

```
class Dragon : IFighter
{
    public void Punch(int side)
    {
        Move(tail, side);
    }
    public void Kick (int side)
    {
        Move(legs[side],
        ... Forward);
    }
    public void Block()
    {
        ... Move(legs[Left ], Up);
        Move(tail, Up);
    }
}
```

# Interface Implementation

- Any class or struct that implements the interface *must implement all its members*.
- By using interfaces, we may include *behavior from multiple sources in a class*.
- It is important in C# because the language *doesn't support multiple inheritance of classes*.
- We must use an interface for simulating *inheritance for structs*, because they can't actually inherit from another struct or class.

# Interface Implementation

```
interface IFighter
{
    void Punch(int side);
    void Kick (int side);
    void Block();
}
```

```
interface IPerson
{
    string Name { get; set; }
    string Introduce();
}
```

IFighter  
methods

IPerson  
methods

```
class Soldier : IFighter, IPerson
{ private string name;
  public void Punch(int side) { ... }
  public void Kick (int side) { ... }
  public void Block() { ... }

  public string Name {get{return name;} set{name = value;}}
  public string Introduce(){return "My name
is"+this.name;}

  ...
}
```

## **IEnumerable:**

The IEnumerable interface allows foreach-loops on collections. It is often used in LINQ.

```
public interface IEnumerable {  
    IEnumerator GetEnumerator();  
}
```

## **IDisposable:**

Provides a mechanism for releasing unmanaged resources

```
public interface IDisposable {  
    void Dispose();  
}
```

## **ICollection:**

Defines methods to manipulate generic collections.

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable {  
    void Add(T item);  
    void Clear();  
    Boolean Contains(T item);  
    void CopyTo(T[] array, Int32 arrayIndex);  
    Boolean Remove(T item);  
    Int32 Count { get; } // Read-only property  
    Boolean IsReadOnly { get; } // Read-only property  
}
```



```
public interface IComparable<T>
{
    Int32 CompareTo (T other) ;
}
```

**Value**

Less than zero

Zero

Greater than zero

**Meaning**

This object is less than the other parameter.

This object is equal to other.

This object is greater than other.

```
class Doctor: IComparable<Doctor>
{
    int CompareTo(Doctor other)
    {
        return salary-other.salary;
    }
    ...
}
```

```
public static void Main()
{
    Doctor [] doctors= new Doctor [5];
    //... input doctors

    Array.Sort(doctors);
}
```

# Task 5-1.

- Develop interface **IFlyable** with method **Fly()**.
- Create two classes **Bird** (with fields: name and canFly) and **Plane** (with fields: mark and highFly) , which implement interface IFlyable.
- Create **List** of **IFlyable** objects and add some Birds and Planes to it. Call Fly() method for every item from the list of it.

- .NET framework provides specialized classes for data storage and retrieval.
- There are two distinct collection types in C#:
  - The ***standard collections*** from the `System.Collections` namespace
  - The ***generic collections*** from `System.Collections.Generic`.
- **Generic collections** are more flexible and safe, and are the preferred way to work with data.

## System.Collections.Generic

List<T>

Dictionary<K,T>

SortedList<K,T>, SortedDictionary<K,T>

Stack<T>

Queue<T>

LinkedList<T> O(1)

ICollection<T>

IDictionary<K,T>

ICollection<T>

IEnumerator<T>

IEnumerable<T>

IComparer<T>

IComparable<T>

## System.Collections

ArrayList

HashTable

SortedList

Stack

Queue

-

ICollection

IDictionary

ICollection

IEnumerator

IEnumerable

IComparer

IComparable

# ArrayList

- **ArrayList** is a special array that provides us with some functionality over and above that of the standard Array.
- Unlike arrays, an ArrayList *can hold data of multiple data types*.
- We can *dynamically resize* it by simply *adding* and *removing* elements.

**create ArrayList** →

```
using System.Collections;

class Department
{
    ArrayList employees = new
ArrayList();
    ...
}
```

# ArrayList

**add new elements** →

**remove** →

**containment testing** →

**read/write existing element** →

**control of memory  
in underlying array** →

```
public class ArrayList : IList, ICloneable
{
    int Add (object value) // at the end
    void Insert(int index, object value) ...

    void Remove (object value) ...
    void RemoveAt(int index) ...
    void Clear () ...

    bool Contains(object value) ...
    int IndexOf (object value) ...

    object this[int index] { get... set.. }

    int Capacity { get... set... }
    void TrimToSize() //minimize memory
    ...
}
```

```
using System.Collections;
```

```
ArrayList da = new ArrayList();  
da.Add("Visual Studio");  
da.Add(344);  
da.Add(55);  
da.Add(new Empty());  
  
da.Remove(55);  
  
foreach(object el in da)  
{  
    Console.WriteLine(el);  
}
```

```
class Empty {}
```



# List<T>

- **List<T>** is a strongly typed list of objects that can be accessed by index.
- It can be found under **System.Collections.Generic** namespace

```
using System.Collections.Generic;
```

```
static void Main()  
{  
    List<string> langs = new List<string>();  
    langs.Add("Java");  
    langs.Add("C#");  
    langs.Add("C++");  
    langs.Add("Javascript");  
  
    Console.WriteLine(langs.Contains("C#"));  
    Console.WriteLine(langs[1]);  
  
    langs.Remove("C#");  
    Console.WriteLine(langs.Contains("C#"));  
  
    langs.Insert(2, "Haskell");  
    langs.Sort();  
    foreach(string lang in langs)  
        { Console.WriteLine(lang); }  
}
```

```
static void Display(IEnumerable<int> values)
{
    foreach (int value in values)
    {
        Console.WriteLine(value);
    }
}
```

```
static void Main()
{
    int[] values = { 1, 2, 3 };
    List<int> values2 = new List<int>() { 1, 2, 3 };

    // Pass to a method that receives IEnumerable.
    Display(values);
    Display(values2);
}
```

- A **Dictionary**, also called an associative array, is a *collection of unique keys* and a *collection of values*
- Each key is associated with one value.
- Retrieving and adding values is very fast.

- Dictionary where we map domain names to their country names:

```
Dictionary<string, string> domains = new Dictionary<string, string>();  
domains.Add("de", "Germany");  
domains.Add("sk", "Slovakia");  
domains.Add("us", "United States");
```

- Retrieve values by their keys and print the number of items:

```
Console.WriteLine(domains["sk"]);  
Console.WriteLine(domains["de"]);  
Console.WriteLine("Dictionary has {0} items", domains.Count);
```

- Print both keys and values of the dictionary:

```
foreach(KeyValuePair<string, string> kvp in domains)  
{  
    Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);  
}
```

- A **Queue** is a *First-In-First-Out* (FIFO) data structure.
- The first element added to the queue will be the first one to be removed.
- Queues may be used to process messages as they appear or serve customers as they come.
- Methods:
  - **Clear()**; removes all elements from the Queue.
  - **Contains**(object obj); determines whether an element is in the Queue.
  - **Dequeue**(); removes and returns the object at the beginning of the Queue.
  - **Enqueue**(object obj); adds an object to the end of the Queue.
  - **ToArray**(); Copies the Queue to a new array.

```
Queue<string> msgs = new Queue<string>();
```

```
    msgs.Enqueue("Message 1");  
    msgs.Enqueue("Message 2");  
    msgs.Enqueue("Message 3");
```

```
    Console.WriteLine(msgs.Dequeue());  
    Console.WriteLine(msgs.Peek());  
    Console.WriteLine(msgs.Peek());
```

```
    Console.WriteLine();
```

```
    foreach(string msg in msgs)  
    {  
        Console.WriteLine(msg);  
    }
```

- A **stack** is a *Last-In-First-Out* (LIFO) data structure.
- The last element added to the queue will be the first one to be removed.
- The C language uses a stack to store local data in a function. The stack is also used when implementing calculators.



```
Stack<int> stc = new Stack<int>();

    stc.Push(1);
    stc.Push(4);
    stc.Push(3);
    stc.Push(6);

    Console.WriteLine(stc.Pop());
    Console.WriteLine(stc.Peek());
    Console.WriteLine(stc.Peek());

    Console.WriteLine();

    foreach(int item in stc)
    {
        Console.WriteLine(item);
    }
```

# Task 5-2. Collections

- Declare **myColl** of 10 integers and fill it from Console.
  - 1) Find and print all positions of element -10 in the collection
  - 2) Remove from collection elements, which are greater than 20.  
Print collection
  - 3) Insert elements 1,-3,-4 in positions 2, 8, 5. Print collection
  - 4) Sort and print collection
- Use next Collections for this tasks: List and ArrayList

1. Create interface **IDeveloper** with property **Tool**, methods **Create()** and **Destroy()**  
Create two classes **Programmer** (with field **language**) and **Builder** (with field **tool**), which implement this interface.  
Create **List** of **IDeveloper** and add some Programmers and Builders to it. Call **Create()** and **Destroy()** methods, property **Tool** for all of it
2. Create Console Application project in VS. In the **Main()** method declare **Dictionary<uint,string>**. Add to Dictionary from Console seven pairs (ID, Name) of some persons. Ask user to enter ID, then find and write corresponding Name from your Dictionary. If you can't find this ID - say about it to user.

