Known Interfaces

- In the .NET environment there are many Interfaces that expose helpful behaviours, like:
  - IComparable
  - IComparer
  - IEnumerable
  - IEnumerator
- Let's look at the Icomparable that has only one function for comparing objects and it's documentary explanation of the return value

```
public interface IComparable
{
        int CompareTo(object obj);
}
```

Returns:
A value that indicates the relative order of the objects being compared.
The return value has these meanings: Value Meaning Less than zero This instance is less than obj. Zero This instance is equal to obj. Greater than zero This instance is greater than obj.

IComparable

- The array class has a function called Array.Sort() . The intellisense tells us that this function sorts the array using the Icomparable implementation.
- If we sort the basic types like int, string… the Icomparable interface is already implemented
- but if we want to sort an array of persons???
- Then we only need to implement Icomparable which is one function in order to let the Array.Sort() function to do all the work
- Pay attention that the one function within the Icomparable interface deals with comparing only two objects and not a whole array

Icomparable example

```
public class Person : IComparable
{
    private int age;
    public int ID { get; set; }
    public static bool sortById = false;
…
        public int CompareTo(object obj)
        {
            Person p = (Person)obj;
            if (sortById)
            {
                if (ID< p.ID)
                    return -1;
                if (ID > p.ID  )
                    return 1;
                return 0;
            }
            else
            {
                if (Age < p.Age)
                    return -1;
                if (Age > p.Age)
                    return 1;
                return 0;
            }
        }
…
```

Icomparable example

```
Person[] pArr = new Person[3];
pArr[0] = new Person(5,100);
pArr[1] = new Person(2,20);
pArr[2] = new Person(12,15);

for (int i = 0; i < pArr.Length; i++)
    pArr[i].Print();

Console.WriteLine("-sort by AGE--------------------");
Array.Sort(pArr);

for (int i = 0; i < pArr.Length; i++)
    pArr[i].Print();


Console.WriteLine("-sort by ID--------------------");
Person.sortById = true;
Array.Sort(pArr);

for (int i = 0; i < pArr.Length; i++)
    pArr[i].Print();
```

-sort by AGE----
Age: 2, ID: 20
Age: 5, ID: 100
Age: 12, ID: 15

-sort by ID----
Age: 12, ID: 15
Age: 2, ID: 20
Age: 5, ID: 100

Sorting and Searching

- Many collections support sorting and searching
  - Algorithms pre-defined, you must provide the compare code
- `IComparable`
  - Implemented by types that compare themselves
- `IComparer`
  - Implemented by a type that compares other types

Can be written from outside the compared class. So if we get a sealed\exe and we can't\don't want to code the inside of the class we can create a separate class for the comparing operation. !!!

IComparable and IComparer

```
public class Person : IComparable {
   public int Age;
   public string Name;

   public int CompareTo( object other ) {
      Person p = other as Person;
      return Age.Compare( p.Age );
   }
}
```

Defines the natural sort sequence for elements of type Person

```
Array.Sort( people );
```

```
public class PersonNameComparer : IComparer

   public int Compare( object o1, object o2 )
   {
      Person p1 = o1 as Person;
      Person p2 = o2 as Person;
      return string.Compare( p1.Name, p2.Name );
   }
}
```

Defines an alternative sort sequence for elements of type Person

```
Array.Sort( people, new PersonNameComparer() );
```