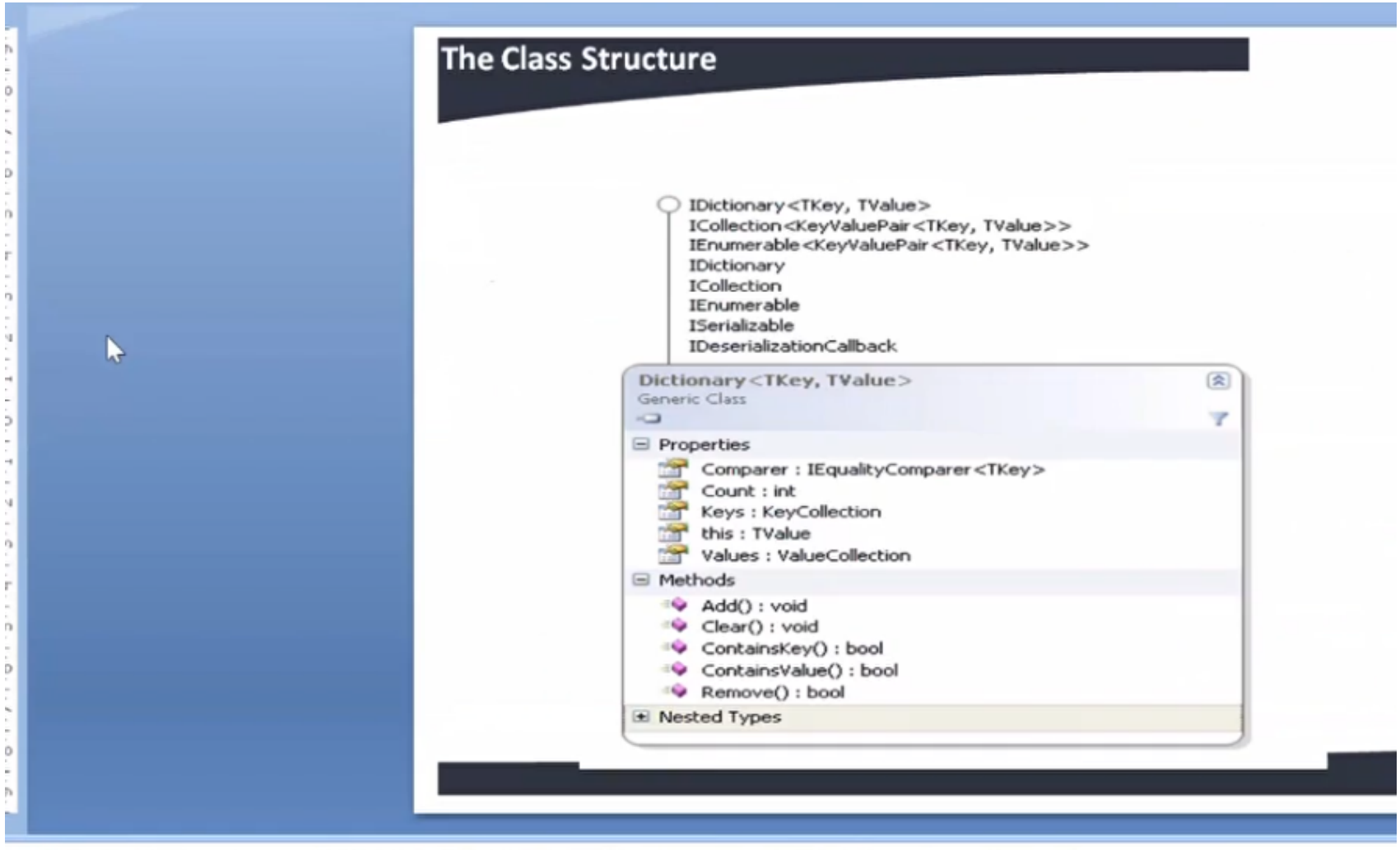
C#-13



Dictionaries represent a sophisticated data structure that allows you to access an element based on a key. Dictionaries are also known as hash tables or maps. The main feature of dictionaries is fast lookup based on keys. You can also add and remove items freely, a bit like a List < T > , but without the

Important Members

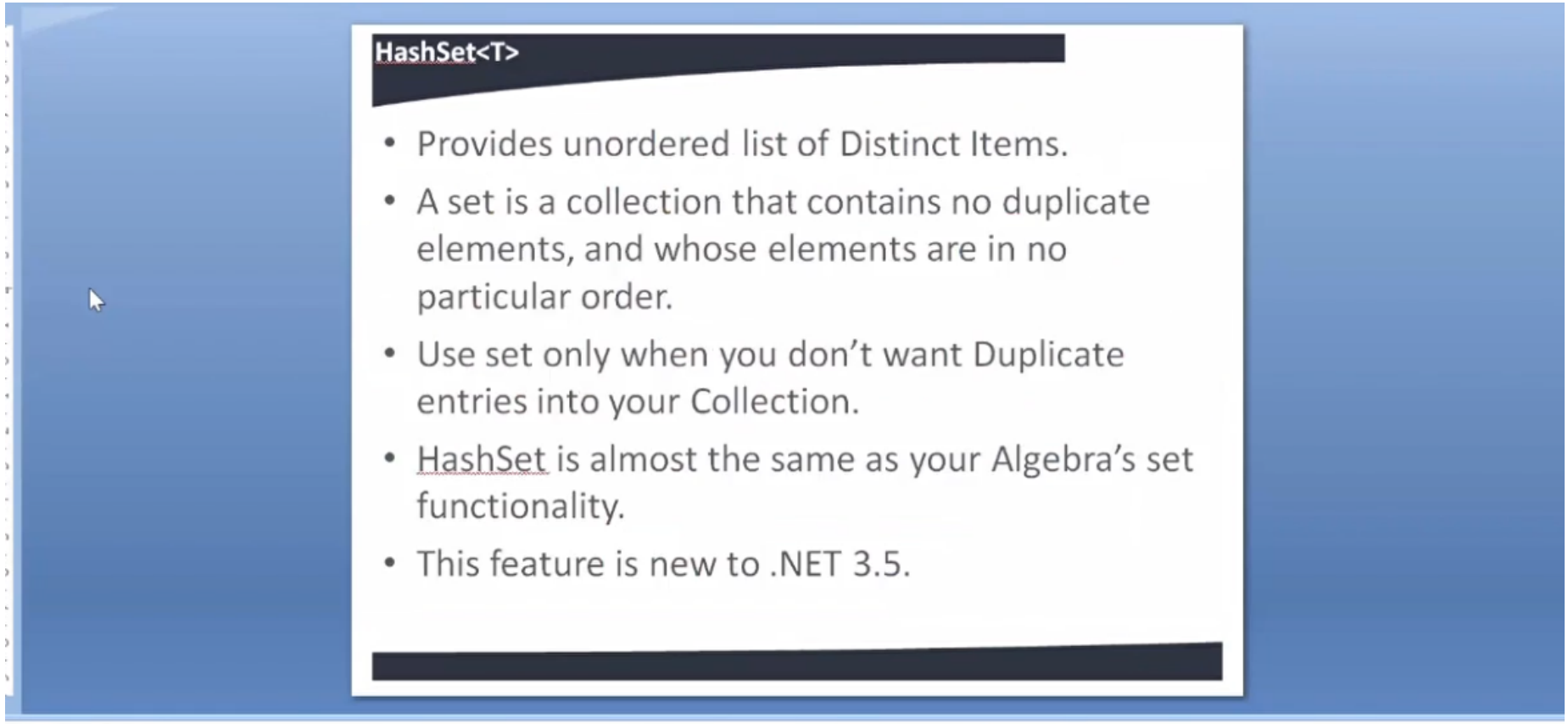
Method	Description
Add(TK k, TV v)	Adds the key/value pair specified by k and v to the dictionary. If the k is already in the dictionary, then its value is unchanged and an ArgumentException is thrown.
ContainsKey(TK k)	Returns true if k is a key in the invoking dictionary. Returns false otherwise.
ContainsValue(TV v)	Returns true if v is a value in the invoking dictionary. Returns false otherwise.
GetEnumerator( )	Returns an enumerator for the invoking dictionary.
Remove(TK k)	Removes k from the dictionary. Returns true if successful. Returns false if k was not in the dictionary.

Code snippet.

```
Dictionary<string, string> Users = new Dictionary<string, string>();
/*Here the users is a collection which stores the
 * userid and password pairs of Each Employee*/
Users.Add("userID1", "pwd2");
Users.Add("userID2", "pwd12");
Users.Add("userID3", "pwd123");
Users.Add("userID3", "pwd123");
/*Throws arguement exception saying that
the key called userID3 already exists in the Collection.
 This Implies Only Unique Keys should be available.*/

if (Users.ContainsKey("userID3"))//Check if the Key already exists....
{
    //Code to tell that the User already exists...
}
else
    Users.Add("userID3", "pwd123");
//Similarly U can check for Value also....
```

Points to Remember…

- A type that is used as a key in the dictionary must override the method GetHashCode() of the Object class.
- Whenever a dictionary class needs to work out where an item should be located, it calls the GetHashCode() method.
- On how to implement GetHashCode() refer the MSDN…..



HashSet<T> implements a set in which all elements are unique. In other words, duplicates are not allowed. The order of the elements is not specified. **HashSet<T>** defines a full complement of set operations, such as intersection, union, and symmetric difference. This makes **HashSet<T>** the perfect

Important Members

Method	Description
Add(T element)	Adds the specified element to a set. Returns false if element already exists.
UnionWith	Modifies the current HashSet<T> object to contain all elements that are present in itself, the specified collection, or both.
IntersectWith	Changes the set to include only elements that are part of both the collection that is passed and the set
RemoveWhere	This method removes all elements on the condition that it match. The Condition is given as function thro a Delegate object called Predicate.
ExceptWith	Receives a collection as argument and removes all the elements from this collection from the set.

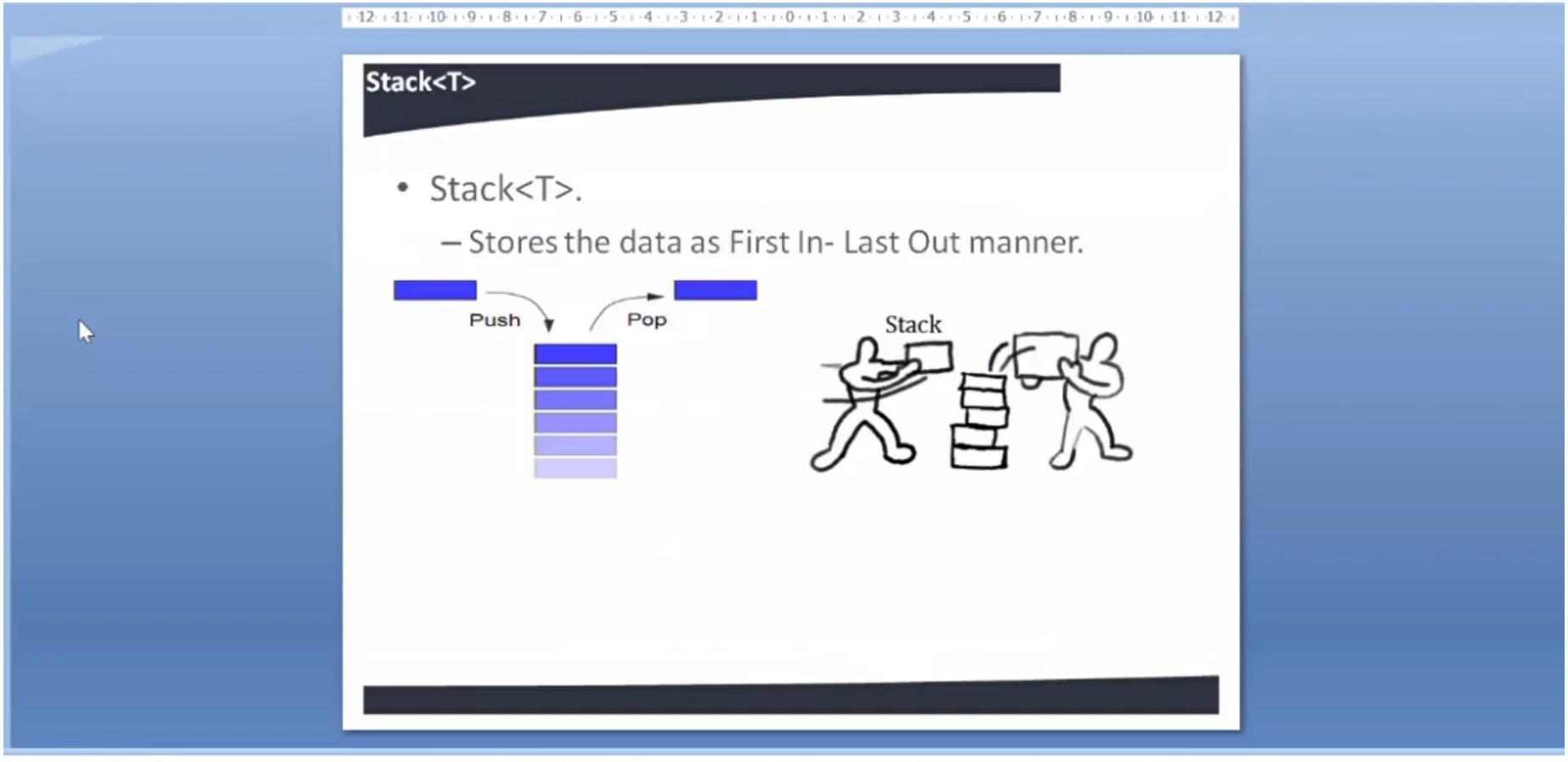
Example

```
HashSet<string> IPLTeams = new HashSet<string>
{
    "Kolkatta Knight Riders",
    "Mumbai Indians",
    "Royal Chalengers",
    "Deccan Chargers",
    "Chennai Super Kings",
    "KingsXI Punjab"
};

HashSet<string> NewTeams = new HashSet<string>
{
    "Pune Warriors",
    "Kochi Tigers",
    "Mysore Kings",
    "Gujarat Patels"
};

if (NewTeams.Add("Oddisi Devils"))
    Console.WriteLine("New Team Added");
if (!NewTeams.Add("Kochi Tigers"))
    Console.WriteLine("They are already Included");
```

Working with Multiple sets

```
HashSet<string> CompleteTeams = new HashSet<string>();
CompleteTeams.UnionWith(IPLTeams);
CompleteTeams.UnionWith(NewTeams);
foreach (var team in CompleteTeams)
    Console.WriteLine(team);//Displays all the Teams with no Duplicates.
```

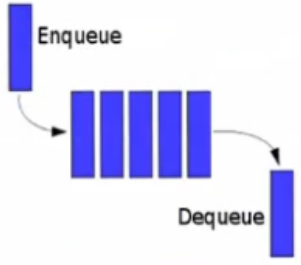
```
CompleteTeams.ExceptWith(NewTeams);
//Removes all NewTeams from the Current Set.
CompleteTeams.IntersectWith(NewTeams);
//Removes all Other Teams that are not the part of NewTeams
```



tack: To visualize a stack, imagine a stack of plates on a table. The first plate put down is the last one to be picked up. The stack is one of the most important ata structures in computing. It is frequently used in system software, compilers, and AI-based backtracking routines, to name just a few.

Queue<T>

- Queue<T>.
  - Stores the Data as First-In- First Out manner.



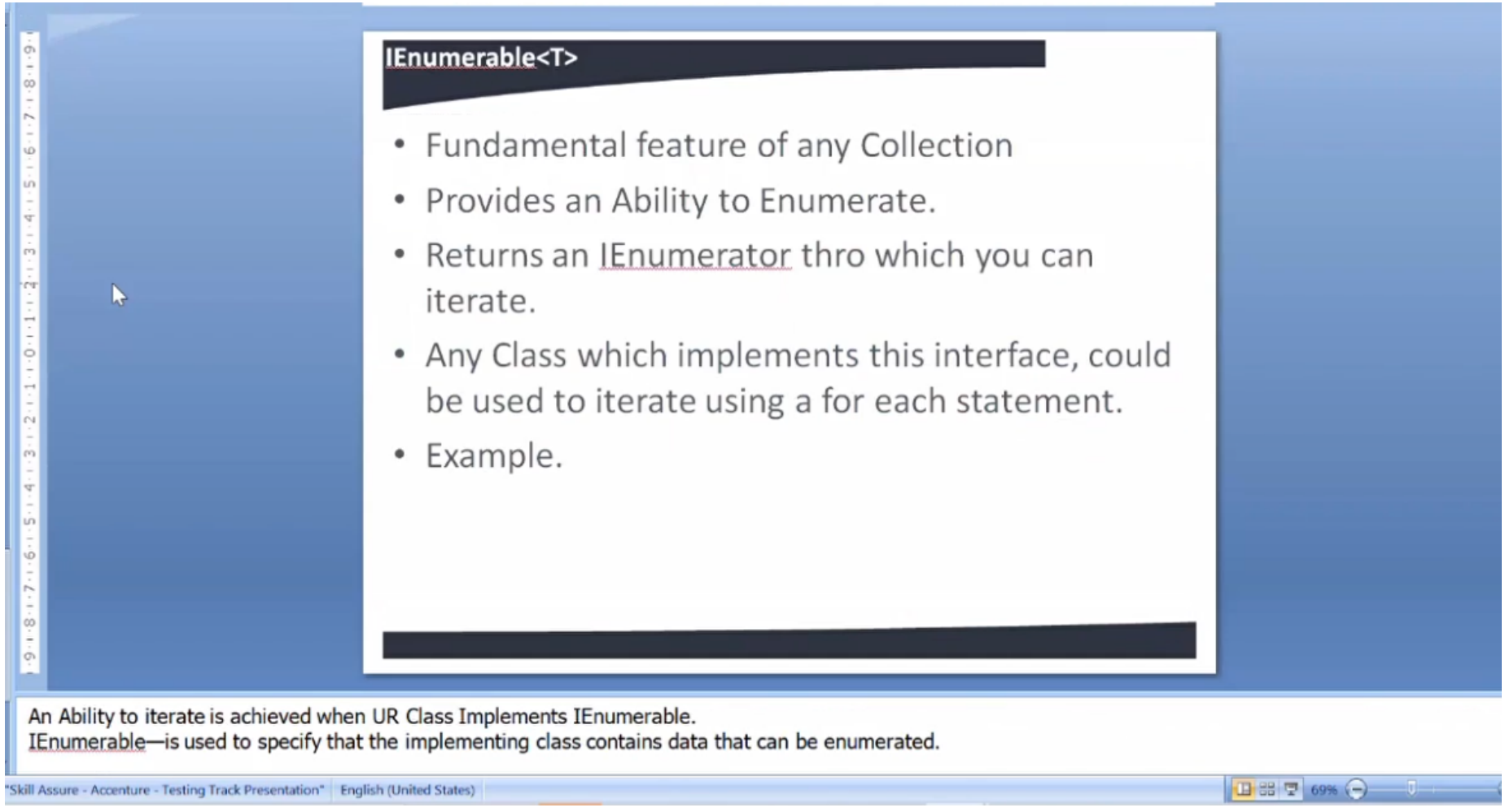
Queue: That is, the first item put in a queue is the first item retrieved. Queues are common in real life. For example, lines at a bank or fast-food restaurant are queues. In programming, queues are used to hold such things as the currently executing processes in the system, a list of pending database transactions, or data packets received over the Internet. They are also often used in simulations.

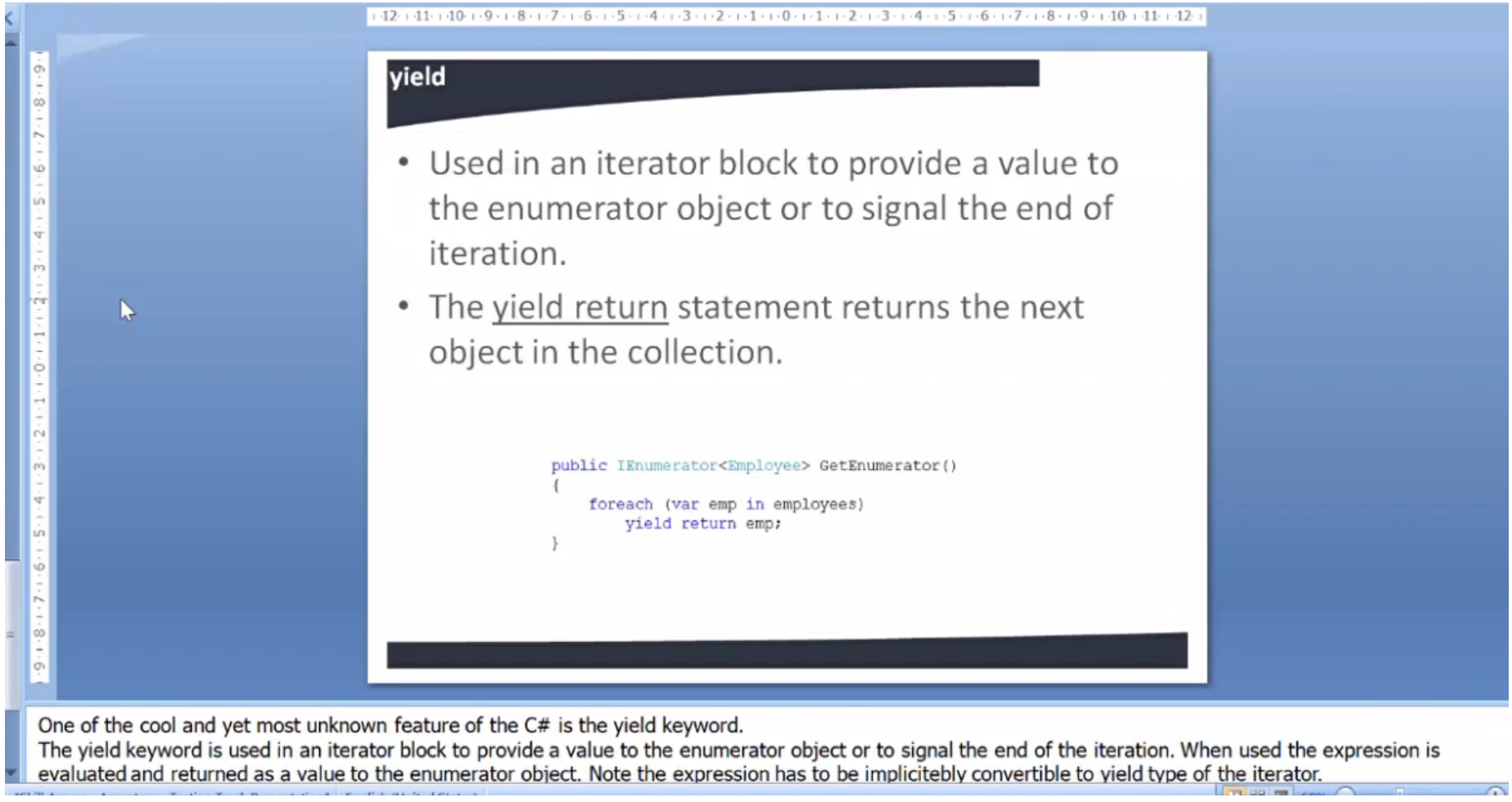
Custom Collections

- What?
  - Classes created to make its object usable in a simple iteration like a for each statement.
- Why?
  - Creating a Collection that suits by my business needs without unnecessary functions.
- How?
  - Implement the interfaces of the Collections to suit your business needs.

Interfaces of Generics

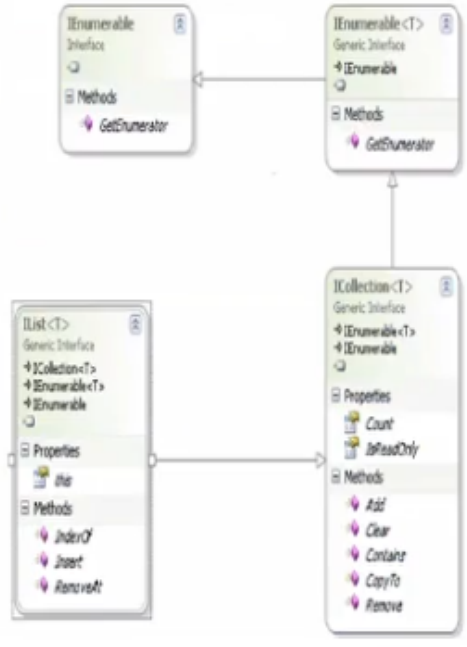
Interfaces	Decription
IEnumerable<T>	Exposes the enumerator, which supports a simple iteration over a collection of a specified type.
ICollection<T>	Defines methods to manipulate generic collections
IList<T>	Represents a collection of objects that can be individually accessed by index
IEnumerator<T>	Supports a simple iteration over a generic collection
IComparer<T>	Defines a method that a type implements to compare two objects
IDictionary<TK,TV>	Represents a generic collection of key/value pairs.
IEqualityComparer<T>	Defines methods to support the comparison of objects for equality


IEnumerable<T>
• Fundamental feature of any Collection
• Provides an Ability to Enumerate.
• Returns an IEnumerator thro which you can iterate.
• Any Class which implements this interface, could be used to iterate using a for each statement.
• Example.
An Ability to iterate is achieved when UR Class Implements IEnumerable.
IEnumerable—is used to specify that the implementing class contains data that can be enumerated.
"Skill Assure - Accenture - Testing Track Presentation"
English (United States)
69%



One of the cool and yet most unknown feature of the C# is the yield keyword.
The yield keyword is used in an iterator block to provide a value to the enumerator object or to signal the end of the iteration. When used the expression is evaluated and returned as a value to the enumerator object. Note the expression has to be implicitebly convertible to yield type of the iterator.

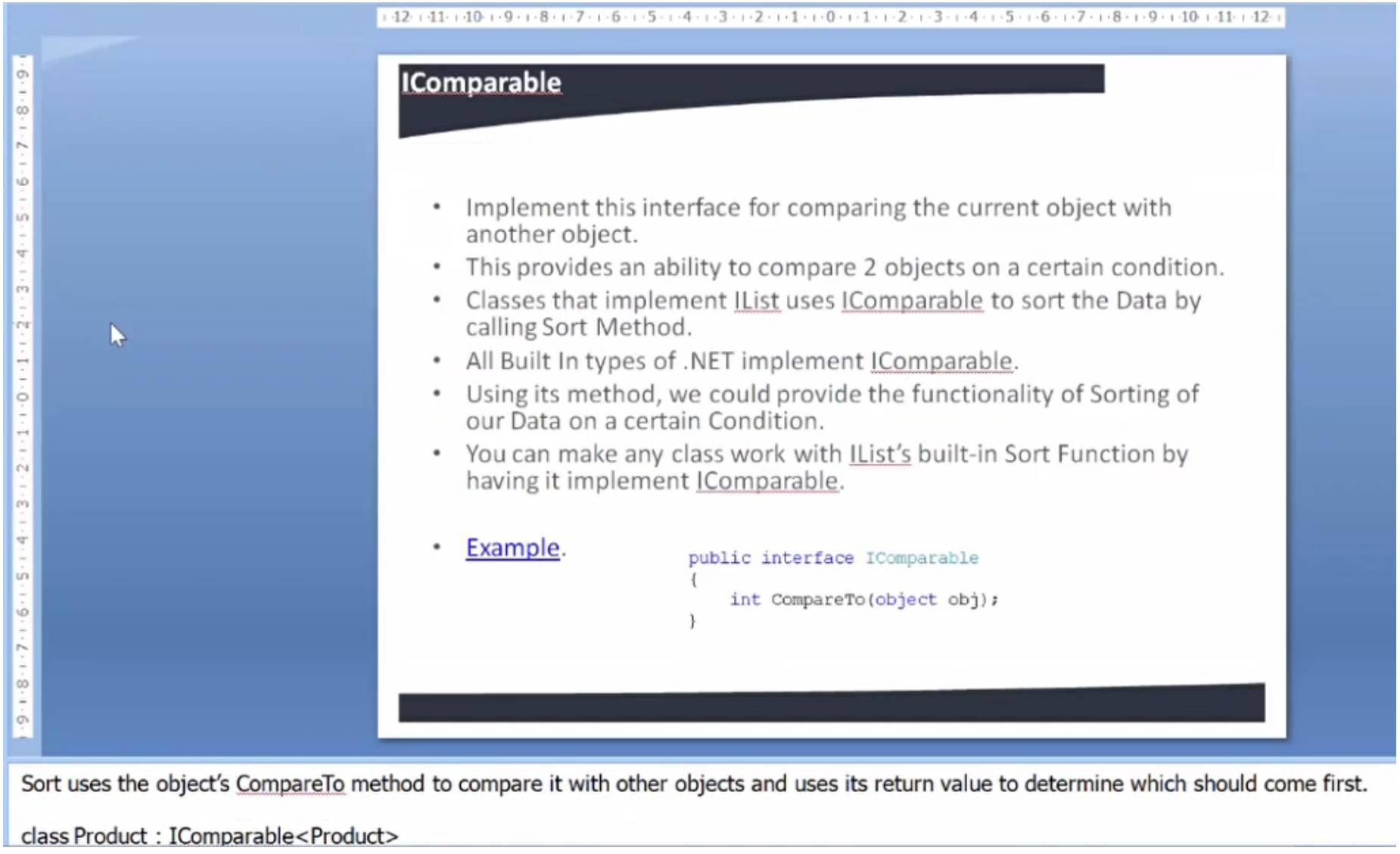
IList<T>



- Represents a collection of objects that can be individually accessed by index.
- Functions
  - Insert
  - RemoveAt

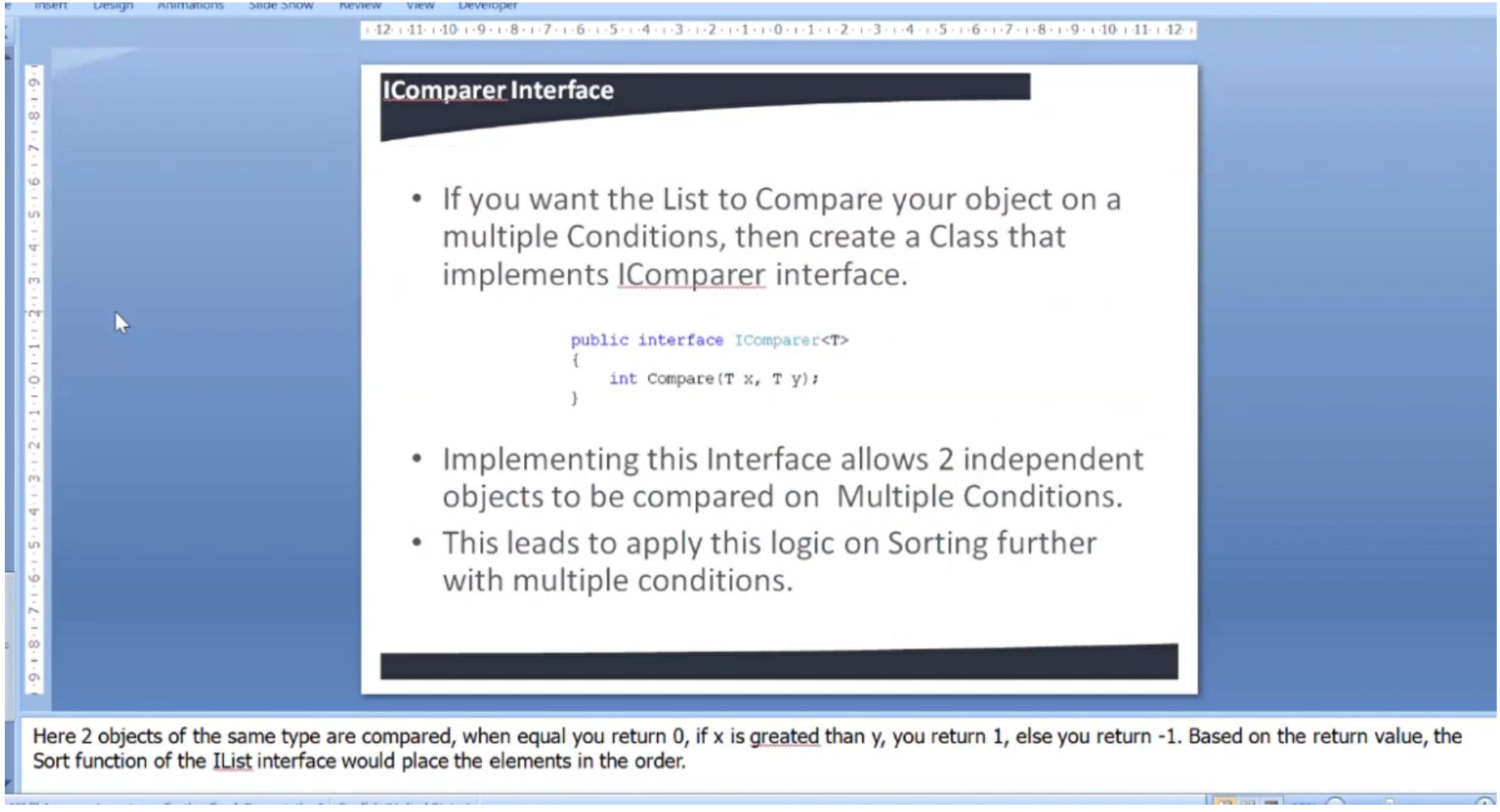
The IList is the Descendent of ICollection which is the base interface for all generic Lists of .NET.

- Insert method inserts the element at the specified Index.
- RemoveAt removes an Element from the specified Index.

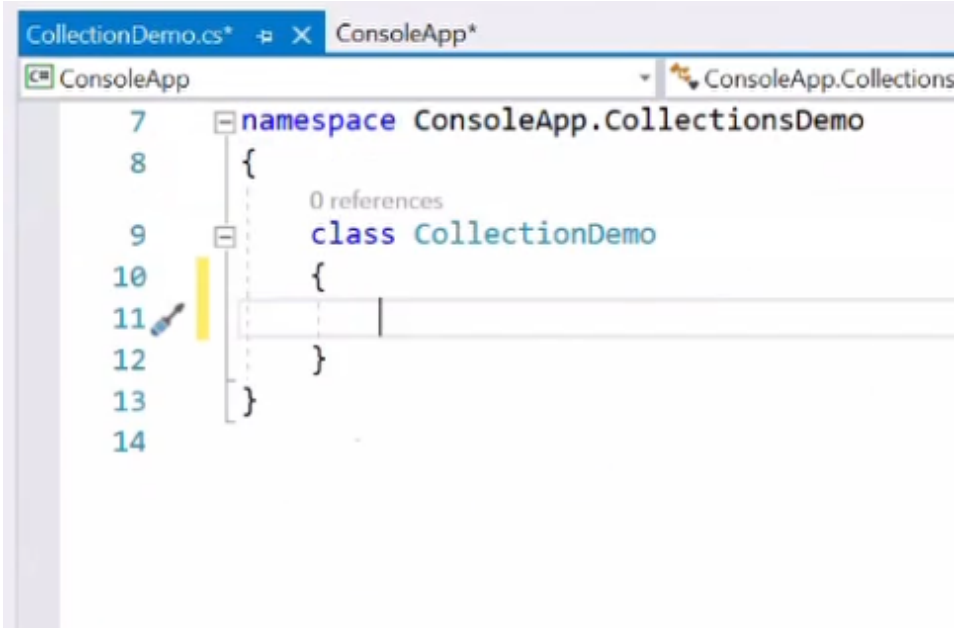


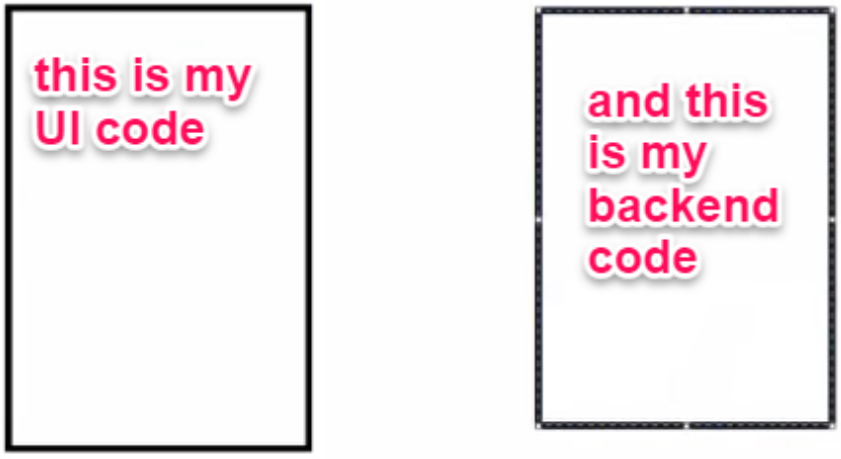
Sort uses the object's CompareTo method to compare it with other objects and uses its return value to determine which should come first.

class Product : IComparable<Product>



There are lot of things which are associated with the collections
imagine that this is my class, which will going to give me some details


CollectionDemo.cs*
ConsoleApp*
ConsoleApp
ConsoleApp.Collections
namespace ConsoleApp.CollectionsDemo
{
0 references
class CollectionDemo
{
}
}

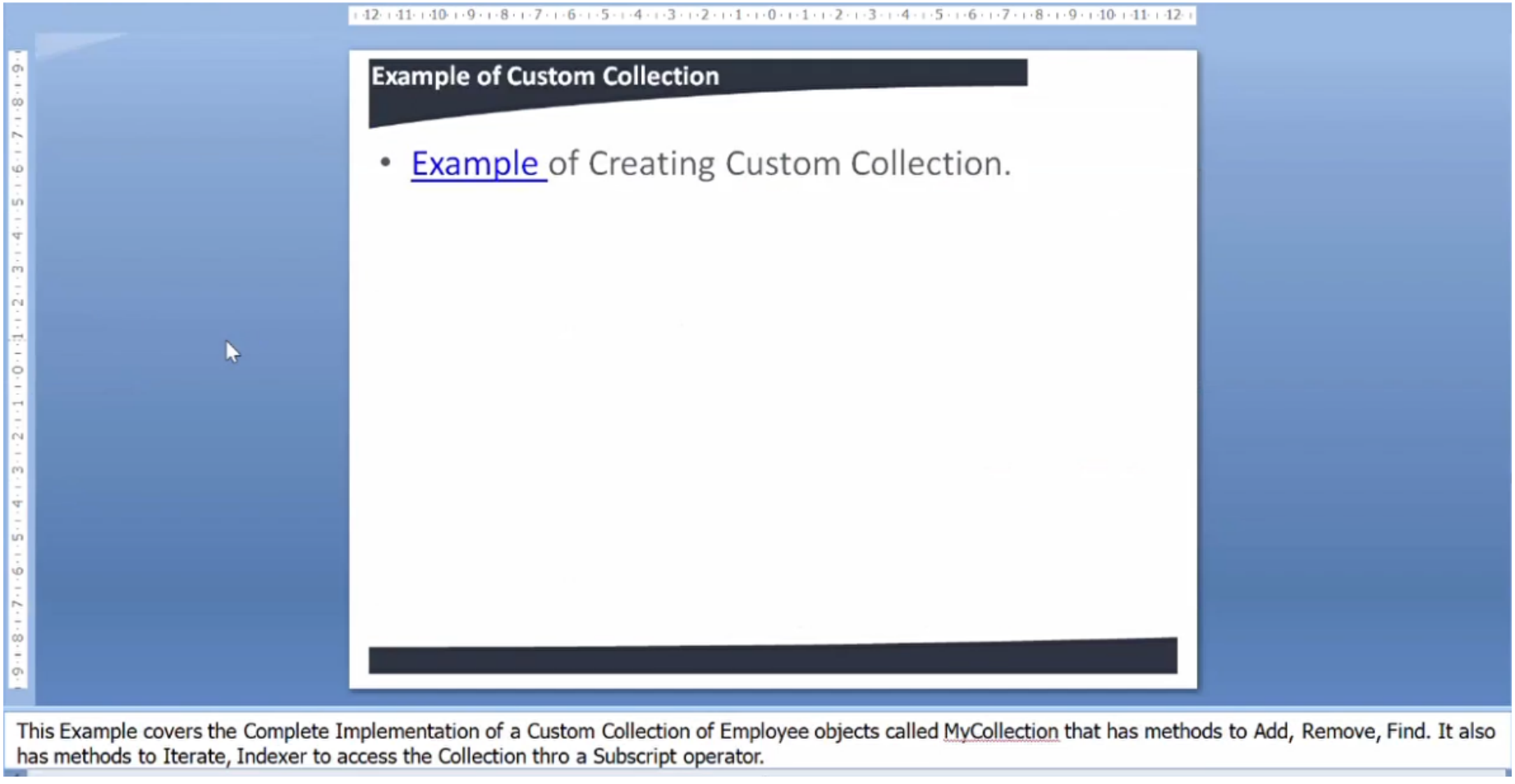


now my backend code have to push some data to my ui code, will my ui code know what sort of data this guy will push
obviously he should know ,
take a example that I'm asking this guy to give me list of products

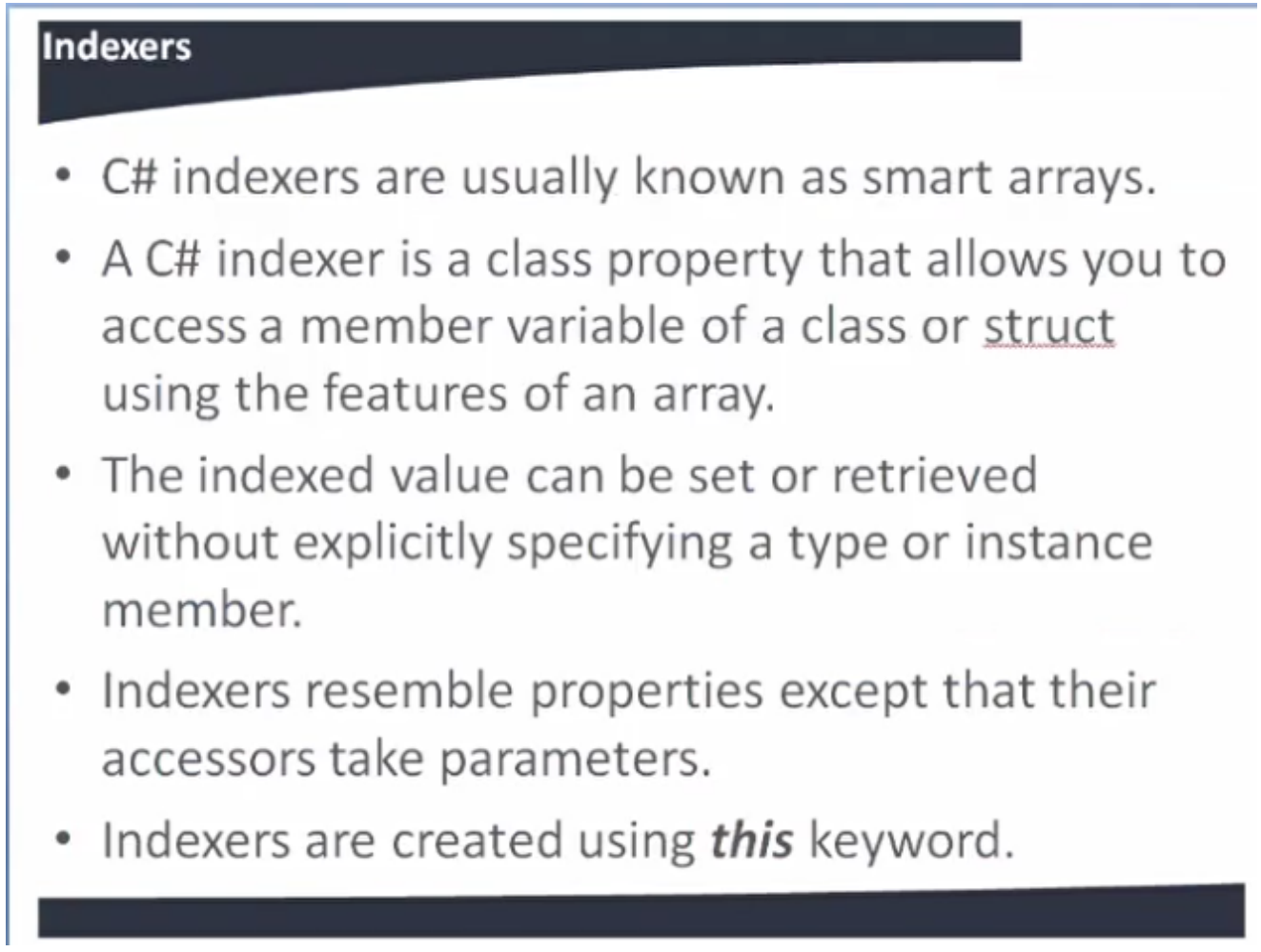
So UI can push a product and get product

So how do we do this

1st thing you should think like what are the products which are available
So while doing shopon we'll see that



next we have **Indexer**



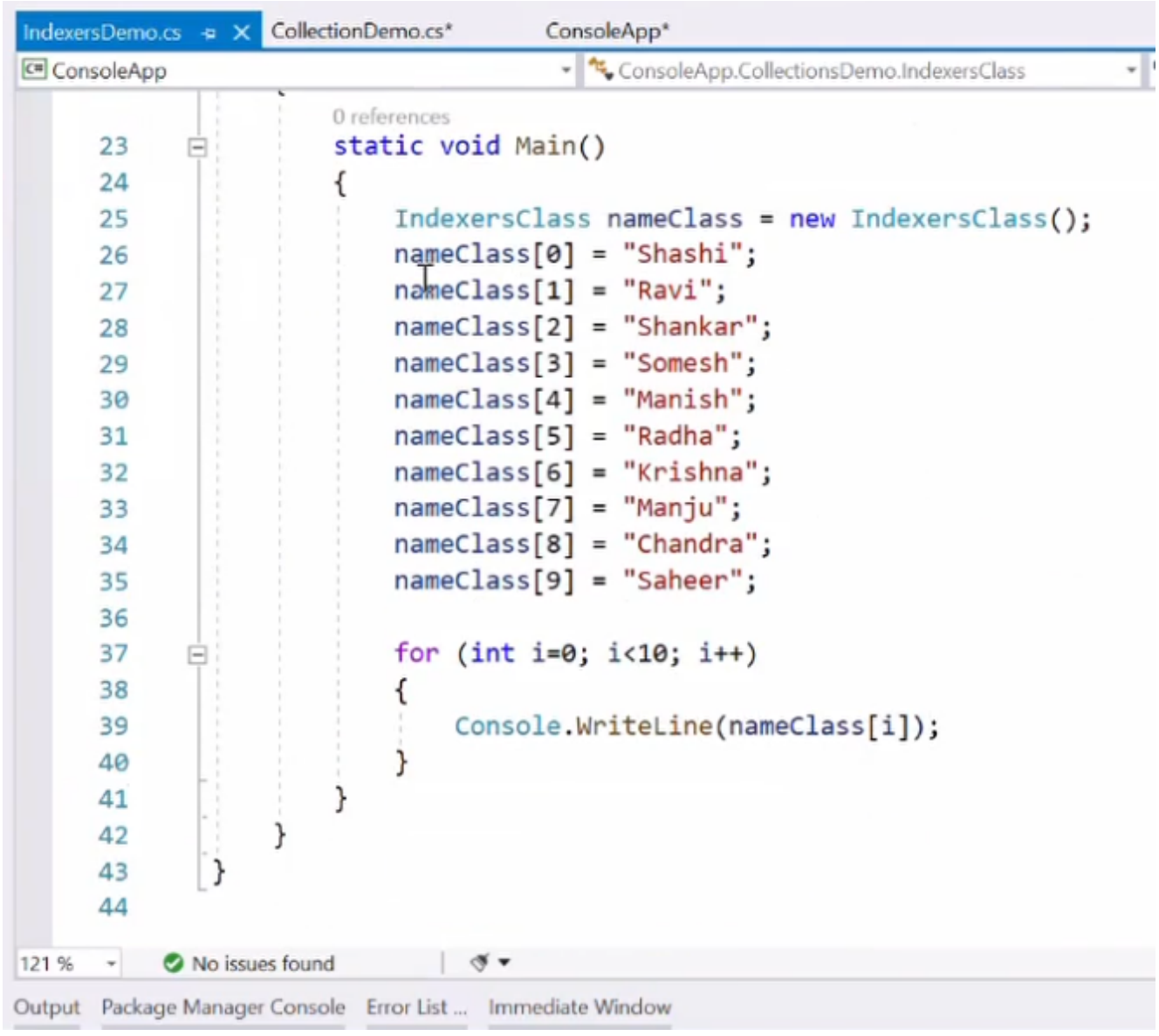
We call indexer as a smart array,

why we call it as a smart array?

usually what happens is, in a array we can store only a string type or numeric type may be i can create a object type, i can create a student array

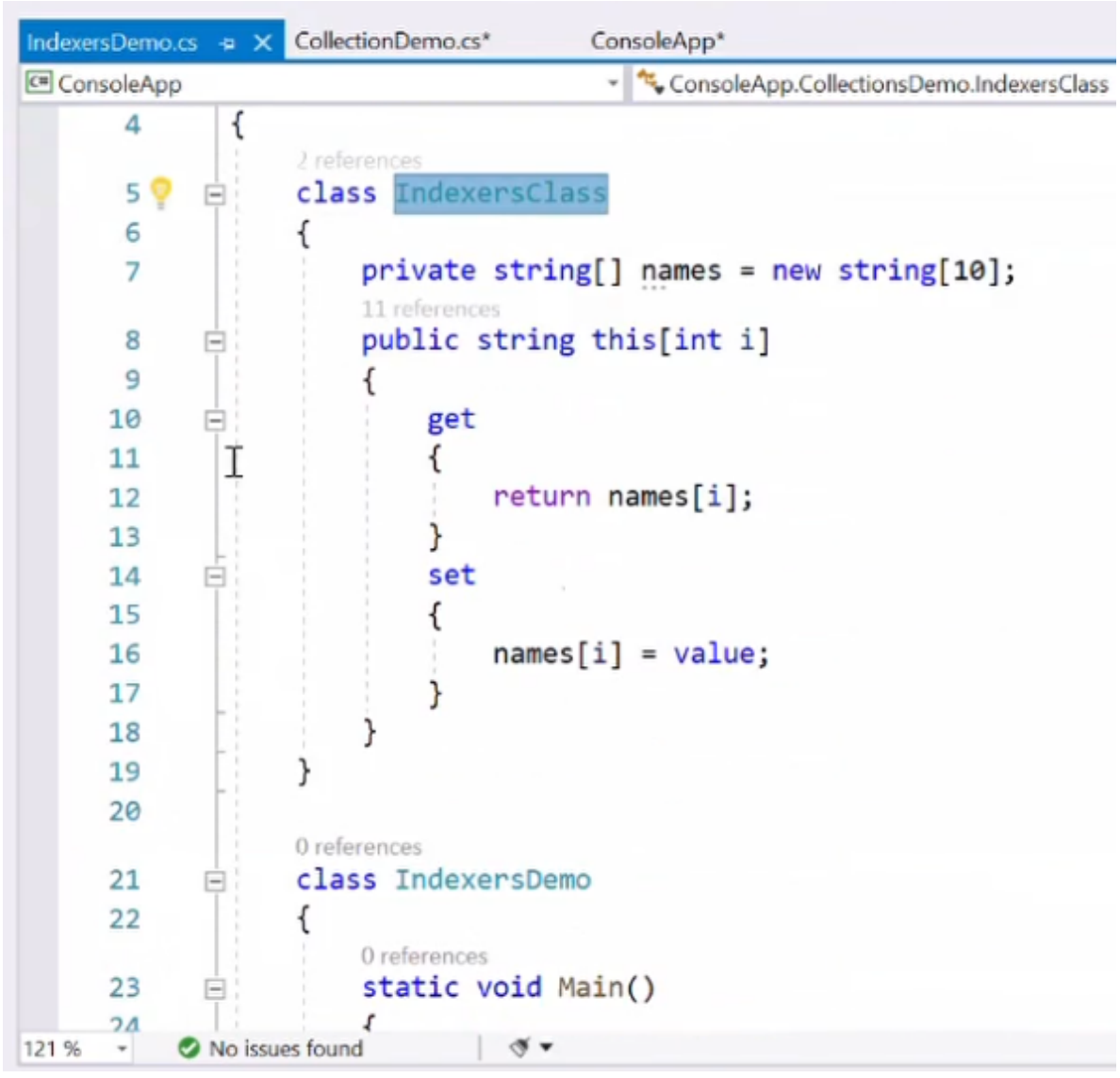
So when we are working in a complex senario, where i want to use the complete object or i want to make the object itself a array thats a time where you can use a indexer

I want to store like this

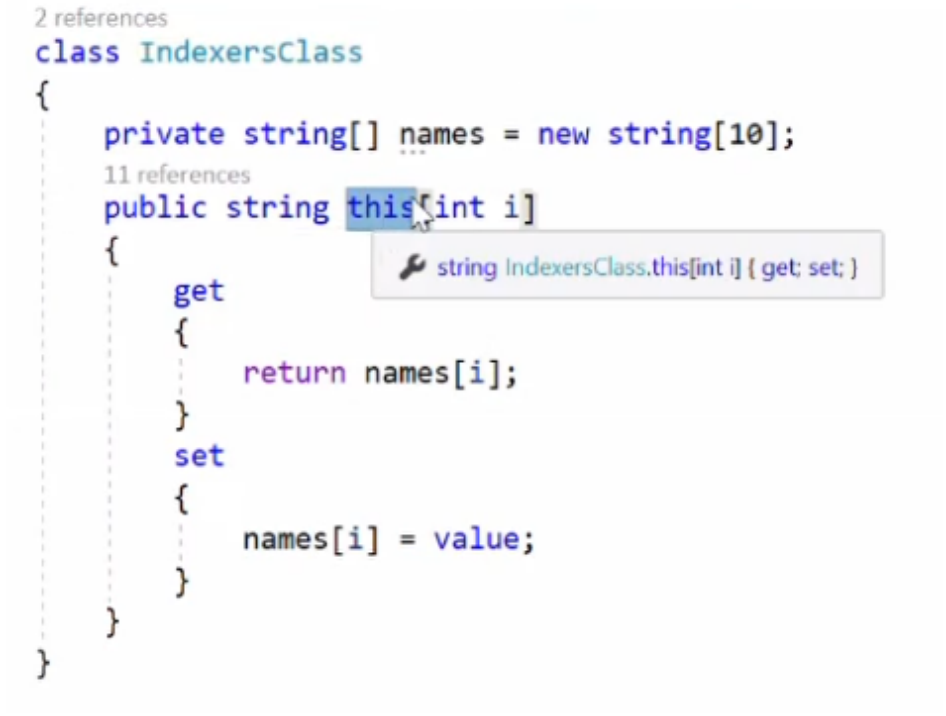

IndexersDemo.cs
CollectionDemo.cs*
ConsoleApp*
ConsoleApp
ConsoleApp.CollectionsDemo.IndexersClass
0 references
static void Main()
{
IndexersClass nameClass = new IndexersClass();
nameClass[0] = "Shashi";
nameClass[1] = "Ravi";
nameClass[2] = "Shankar";
nameClass[3] = "Somesh";
nameClass[4] = "Manish";
nameClass[5] = "Radha";
nameClass[6] = "Krishna";
nameClass[7] = "Manju";
nameClass[8] = "Chandra";
nameClass[9] = "Saheer";
for (int i=0; i<10; i++)
{
Console.WriteLine(nameClass[i]);
}
}
}
}
121 %
No issues found
Output
Package Manager Console
Error List ...
Immediate Window

this is just a array right
but look at this my nameClass is this a array here, no this is not a array, this is instance of my IndexersClass
Remember, indexersclass is a class to this class when i'm creating a instance, to this instance i'm using it as [0], [1], [2],.....

now to fetch the data you can simply use the for loop and you can get the data
you cannot use foreach loop with the indexer because we have to pass particular index i here
now to do this
this is my indexersclass



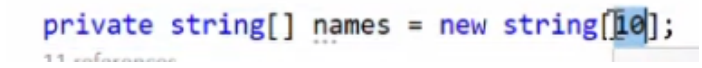
it's a normal class, but it has a speciality of using **this** here



this is where the trick is

now what we are doing here

I'm creating a string array which will store me some kind of static number, i'creating string array of 10

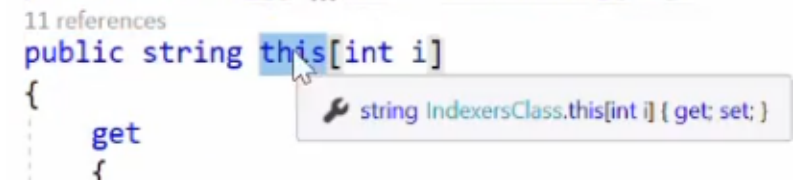


and my indexer here is a property

```
public string this[int i]
{
    get
    {
        return names[i];
    }
    set
    {
        names[i] = value;
    }
}
```

now what this particular property will do ?

look here, my property here is of type this



this means what?

----------current object

So whenever you use this meaning, whenever you are passing any object here

So this particular data what you are passing

```
nameClass[0] = "Shashi";
```

will be stored in the variable called value

```
set
{
    names[i] = value;
}
```
(parameter) string value

that's how it works in property, isn't it
that time i'm going to set it with my ---- name of i
because when we are passing this 0, look at this i'm saying , of 0

```
nameClass[0] = "Shashi";
```

so 0 will be stored in variable i, that becomes my index and to that index i'm storing the value ,

in the same way when i say return name of i

```
get
{
    return names[i];
}
```

so it will return you the value of that particular valriable at that particular location

Creating Indexers

```
<modifier> <return type> this [argument list]
{
get
{
// your get block code
}
set
{
// your set block code
}
}
```

<modifier>
can be private, public, protected or internal.

<return type>
can be any valid C# types.

this
this is a special keyword in C# to indicate the object of the current class.

[argument list]
The formal-argument-list specifies the parameters of the indexer.

Points to remember

- Indexers are always created with **this** keyword.
- Parameterized property are called indexer.
- Indexers are implemented through get and set accessors for the [ ] operator.
- ref and out parameter modifiers are not permitted in indexer.
- The formal parameter list of an indexer corresponds to that of a method and at least one parameter should be specified.
- Indexer is an instance member so can't be static but property can be static.
- Indexers are used on group of elements.
- Indexer is identified by its signature where as a property is identified by it's name.
- Indexers are accessed using indexes where as properties are accessed by names.
- Indexer can be overloaded.

Indexers Example

```
class IndexersClass
{
    private string[] names = new string[10];
    11 references
    public string this[int i]
    {
        get
        {
            return names[i];
        }
        set
        {
            names[i] = value;
        }
    }
}
```

```
class IndexersDemo
{
    0 references
    static void Main()
    {
        IndexersClass nameClass = new IndexersClass();
        nameClass[0] = "Shashi";
        nameClass[1] = "Ravi";
        nameClass[2] = "Shankar";
        nameClass[3] = "Somesh";
        nameClass[4] = "Manish";
        nameClass[5] = "Radha";
        nameClass[6] = "Krishna";
        nameClass[7] = "Manju";
        nameClass[8] = "Chandra";
        nameClass[9] = "Saheer";

        for (int i=0; i<10; i++)
        {
            Console.WriteLine(nameClass[i]);
        }
    }
}
```

there are 2 things which we have to do
1st we have to see how to use the collections
2nd how to use inheritance , the power of inheritance

that we'll write with Company order