Interface, Indexer & Delegates

Course Code: CSC 2210 Course Title: Object Oriented Programming 2

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**07**	**Week No:**	**08**	**Semester:**	
Lecturer:					

Lecture Outline

- Introduction and usage of Interface in OOP and in project management.
- Explicit Interface concepts.
- Implementation of Indexer and its use in library.
- Basic concept of delegates as function pointers.


AMERICAN INTERNATIONAL UNIVERSITY
PRAESIDIUM
DISCIPLINA
CIVITATIS
1994
BANGLADESH

Interface

Interface

- An interface looks like a class, but has no implementation.
- The only thing it contains are declarations of members.
- An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement.
- An interface defines a contract. Any class or struct that implements that contract must provide an implementation of the members defined in the interface.
- An interface may not declare instance data such as fields, auto-implemented properties, or property-like events.
- Beginning with C# 8.0, an interface may define a default implementation for members.
- An interface may define static methods, which must have an implementation.

```
public interface ITransaction
{
    bool Deposit(double amount);
}
```

Interface

- The name of an interface must be a valid C# identifier name.
- By convention, interface names begin with a capital I. such as *ITestInterface.*
- Interfaces can contain instance **methods, properties, events, indexers,** or any combination of those four member types.
- Interface is pure abstraction with no implementation.
- Interfaces may contain **static constructors, fields, constants** or **operators.**
- An interface can't contain **instance fields, instance constructors** or **finalizers**.
- Interface members are abstract and public by default.

Interface members are public by default.

```
public interface ILog
{
    void Write(string Message);
}
```

Interface

- Like abstract classes, interfaces cannot be used to create objects.
- An interface cannot contain a constructor (as it cannot be used to create objects).
- Interface methods do not have a body - the body is provided by the "implement" class.
- To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.
- When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface declares but doesn't provide a default implementation for.
- However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.
- An interface can be a member of a namespace or a class.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.


```
interface IPen
{
    string Color { get; set; }
    bool Open();
    bool Close();
    void Write(string text);
}
class Cello : IPen
{
    public string Color { get; set; }

    private bool isOpen = false;

    public bool Close()
    {
        isOpen = false;
        Console.WriteLine("Cello closed for
writing!");
return isOpen;
    }

    public bool Open()
    {
        isOpen = true;
        Console.WriteLine("Cello open for
writing!");

        return isOpen;
    }
    public void Write(string text)
    {
        //write text if open
        if(isOpen)
            Console.WriteLine("Cello: " +
text);
    }
}
```

Interface

Any class or struct that implements the IEquatable<T> interface must contain a definition for an Equals method that matches the signature that the interface specifies. As a result, you can count on a class that implements IEquatable<T> to contain an Equals method with which an instance of the class can determine whether it's equal to another instance of the same class.

No need to use the *override* keyword when implementing an interface:

Interface

IEquatable Explanation


```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T>
interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}
```

Interface

Why and When to use

- To achieve security - hide certain details and only show the important details of an object (interface).
- C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can implement multiple interfaces.
- To implement multiple interfaces, separate them with a comma.
- Inherited class must implement all members of interface otherwise it will become ***abstract*** also.
- Structure can also implement interfaces.

```
class DemoClass : IFirstInterface, ISecondInterface
{
  ……..……
}
```

Interface

Explicit Interface Implementation

- Explicit implementation is useful when class is implementing multiple interface thereby it is more readable and eliminates the confusion.
- It is also useful if interfaces have same method name coincidently.
- Do not use *public* modifier with an explicit implementation. It will give compile time error.
- There can be multiple classes or structs that implements the same interface.
- Implement interface explicitly using InterfaceName. with all the members.
- An interface can inherit one or more interfaces.
- See example: https://www.tutorialsteacher.com/csharp/csharp-interface

Interface

Interface Type Variables

- As we know you can't create or call constructor of interface but by implementing the interface one can create interface type variable.
- Only members from individual interface can be called using the variable.
- Consider the following code snippet where **Test** class implements ***ITest1*** & ***ITest2*** interface containing *Method1()* and *Method2()*

```
ITest1 t1 = new Test();
t1.Method1();

ITest2 t2 = new Test();
t2.Method2();
```


AMERICAN INTERNATIONAL UNIVERSITY
PRAESIDIUM
DISCIPLINA
CIVITATIS
1994
BANGLADESH

Indexer

Indexer

- Indexers enable objects to be indexed in a similar manner to arrays.
- An Indexer is a special type of property that allows a class or structure to be accessed like an array for its private collection.
- An indexer can be defined the same way as property with **this** keyword and square brackets []. Following is general syntax for indexer declaration.

```
<return type> this[<parameter type> index]
{
    get{
        // return the value from the specified index of an internal collection
    }
    set{
        // set values at the specified index in an internal collection
    }
}
```

Indexer

- The ***this*** keyword is used to define the indexer.
- A get accessor returns a value. A set accessor assigns a value.
- The value keyword is used to define the value being assigned by the set indexer.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.


```
// The following example defines a generic class with simple get
and set accessor methods to assign and retrieve values.
using System;
class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use []
notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Indexer

Indexer Overload

- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.
- Can be overloaded with the different data types for index or number of parameters can be different.

Indexer

Expression Body Defination

- Expression-bodied members provide a simplified syntax.
- Note that **=>** introduces the expression body.
- Starting with C# 7.0, both the get and set accessors can be an implemented as expression-bodied members.
- In this case, both get and set keywords must be used.

```
// Declare an array to store the data elements.
private T[] arr = new T[100];

// Define the indexer to allow client code to use [] notation.
public T this[int i]
{
   get => arr[i];
   set => arr[i] = value;
}
```

Indexer

Comparison Between Properties and Indexers

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 accessor of a property has no parameters.	A get accessor of an indexer has the same formal parameter list as the indexer.
A set 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 parameter.
Supports shortened syntax with Auto-Implemented Properties.	Supports expression bodied members for get only indexers.


AMERICAN INTERNATIONAL UNIVERSITY
PRAESIDIUM
DISCIPLINA
CIVITATIS
1994
BANGLADESH

Delegate

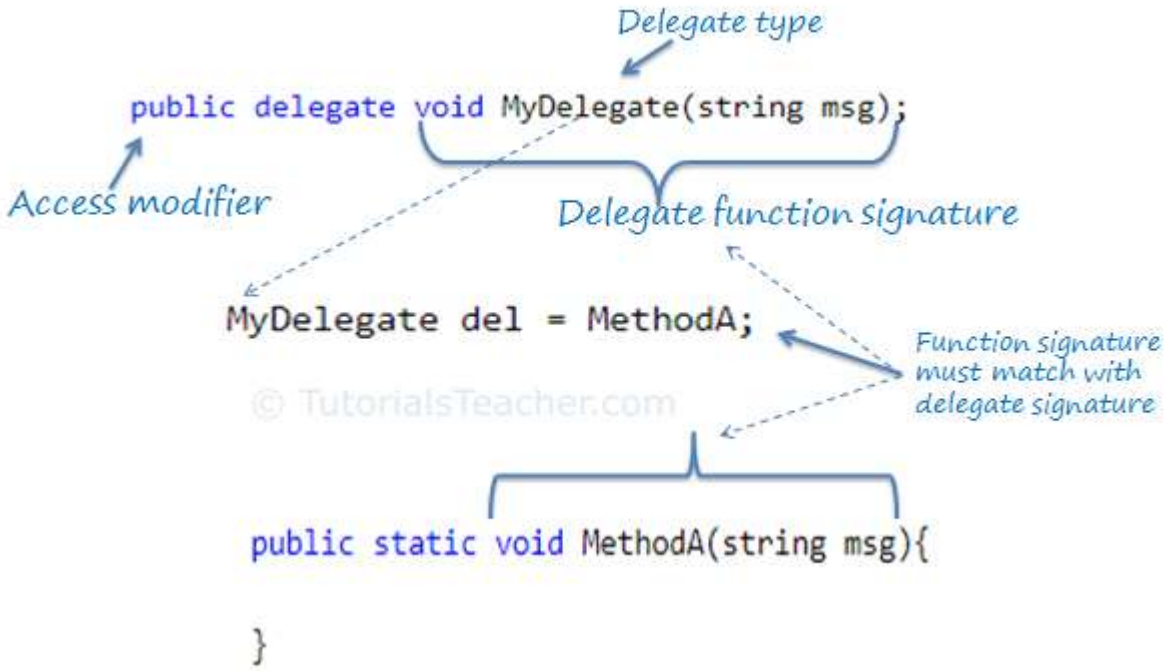
Delegate

- A delegate is an object which refers to a method or you can say it is a reference type variable that can hold a reference to the methods.
- A delegate is a type that represents references to methods with a particular parameter list and return type.
- Delegates are used to pass methods as arguments to other methods.
- When we instantiate a delegate, we can associate its instance with any method with a compatible signature and return type.
- One can invoke (or call) the method through the delegate instance.

```
public delegate int PerformCalculation(int x, int y);
public static void DelegateMethod(string message){
    Console.WriteLine(message);
}

Del handler = DelegateMethod; // Instantiate the delegate
handler("Hello World"); // Call the delegate.
```

Delegate


Delegate type
public delegate void MyDelegate(string msg);
Access modifier
Delegate function signature
MyDelegate del = MethodA;
Function signature must match with delegate signature
© TutorialsTeacher.com
public static void MethodA(string msg){
}

Delegate

- Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate.
- The method can be either static or an instance method.
- This makes it possible to programmatically change method calls, and also plug new code into existing classes.
- Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
- Delegates can be used to define callback methods.
- Delegates allow methods to be passed as parameters.
- **Event handlers are nothing more than methods that are invoked through delegates.**

Delegate

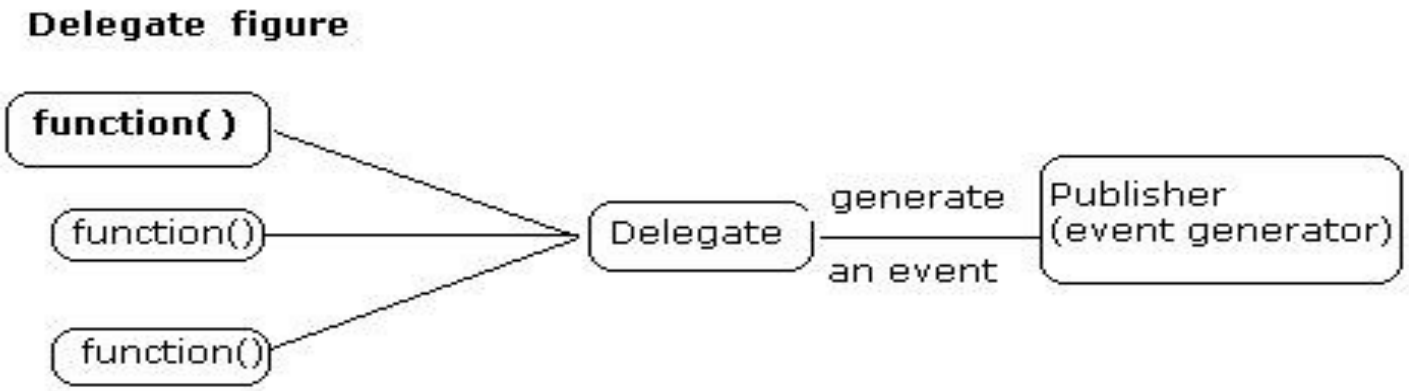
- Delegate types are derived from the Delegate class in the .NET Framework.
- **Delegate Type is also Reference type.**
- `Action` and `Func` are delegates that we can use instead of defining our own delegate types. Important to remember: `Action` and `Func` are delegates.
- `Func<>` to represent a method that returns something.
- `Action` and `Action<>` represent methods that return nothing.

Delegate

Delegate Multicasting

- Delegates can be chained together; for example, multiple methods can be called on a single event.
- When a delegate is wrapped with more than one method that is known as a multicast delegate.
- Multicast delegates are used extensively in event handling.
- The "+" or "+=" operator adds a function to the invocation list, and the "-" and "-=" operator removes it.

Delegate


Delegate figure
function( )
function()
function()
Delegate
generate
an event
Publisher
(event generator)


```
public delegate void MyDelegate(string msg); //declaring a delegate

class Program
{
    static void Main(string[] args)
    {
        MyDelegate del1 = ClassA.MethodA;
        MyDelegate del2 = ClassB.MethodB;

        MyDelegate del = del1 + del2; // combines del1 + del2
        del("Hello World");

        MyDelegate del3 = (string msg) => Console.WriteLine("Called
lambda expression: " + msg);
        del += del3; // combines del1 + del2 + del3
        del("Hello World");

        del = del - del2; // removes del2
        del("Hello World");

        del -= del1 // removes del1
        del("Hello World");
    }
}
```

Delegate

Usage of Delegate

- Delegate is used to declare an event and anonymous methods in C#.
- Need to pass a method as a parameter of other methods.
- it often serves as the basis for the event-handling model in C# and .NET.
- Delegates are object oriented and type-safe and very secure as they ensure that the signature of the method being called is correct.
- Delegate helps in code optimization.
- They are extensively used in threading.
- Delegates are also used for generic class libraries, which have generic functionality, defined.

Delegate

Anonymous Method

You can create a delegate, but there is no need to declare the method associated with it. You do not have to explicitly define a method prior to using the delegate. Such a method is referred to as ***anonymous***. In other words, if a delegate itself contains its method definition it is known as ***anonymous method***.
The code on is an example of an anonymous method:

```
public delegate void Print(int value);

static void Main(string[] args)
{
    Print print = delegate(int val) {
        Console.WriteLine("Inside Anonymous method. Value: {0}", val);
    };

    print(100);
}
```


```
using System;

public delegate void Test();

public class Program
{
        static int Main()
        {
                Test Display = delegate()
                {
                Console.WriteLine("Anonymous Delegate method");
                };

                Display();
                return 0;
        }
}
```

Delegate

Anonymous Method

- An anonymous method is a method without a name.
- Anonymous methods in C# can be defined using the ***delegate*** keyword and can be assigned to a variable of delegate type.
- Anonymous methods can access variables defined in an outer function.
- Anonymous methods can also be passed to a method that accepts the delegate as a parameter.
- Anonymous methods can be used as event handlers.

Thank You

Books

- C# 4.0 The Complete Reference; Herbert Schildt; McGraw-Hill Osborne Media; 2010
- Head First C# by Andrew Stellman
- Fundamentals of Computer Programming with CSharp – Nakov v2013

References

- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/
- https://www.c-sharpcorner.com/UploadFile/sekarbalag/Interface-In-CSharp/
- https://www.geeksforgeeks.org/c-sharp-interface/
- https://www.tutorialsteacher.com/csharp/csharp-interface
- https://www.tutorialspoint.com/csharp/csharp_interfaces.htm
- https://www.w3schools.com/cs/cs_interface.asp
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/
- https://www.tutorialsteacher.com/csharp/csharp-indexer
- https://www.infoworld.com/article/3018437/how-to-work-with-indexers-in-c.html
- https://www.c-sharpcorner.com/article/indexer-in-C-Sharp/
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/
- https://www.c-sharpcorner.com/UploadFile/puranindia/C-Sharp-net-delegates-and-events/
- https://www.pluralsight.com/guides/how-why-to-use-delegates-csharp

References

- https://www.tutorialsteacher.com/csharp/csharp-delegates
- https://www.tutorialspoint.com/csharp/csharp_delegates.htm
- https://www.geeksforgeeks.org/c-sharp-delegates/
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates
- https://www.tutorialsteacher.com/csharp/csharp-anonymous-method
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-functions