Beginning Level **C#**

Interface, Enums, Events and Delegates

Yong Zhang, Ph.D
Weber State University
2019

Interfaces

Interface

- An interface defines the behavior that a class has, but not how this behavior is implemented.
- Interfaces stand as separate constructs from classes, but they require a class to provide the working code to fulfill the interface.

Syntax

- Using the keyword interface.
- Can contain properties, methods, and events, just as classes can.
- No element of an interface can be given an access modifier.

Interfaces

- An interface declaration is like a class declaration, but it provides no implementation for its members, since all its members are implicitly abstract.
- These members will be implemented by the classes and structs that implement the interface.

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext() => count-- > 0;
  public object Current => count;
  public void Reset() { throw new NotSupportedException(); }
}
```

```
IEnumerator e = new Countdown();
while (e.MoveNext())
  Console.Write (e.Current);
```

Interface Declaration

```
public interface ISimpleInterface
{
    void ThisMethodRequiresImplementation();

    string ThisStringPropertyNeedsImplementingToo
    {
        get;
        set;
    }

    int ThisIntegerPropertyOnlyNeedsAGetter
    {
        get;
    }

    public event EventHandler<EventArgs> InterfacesCanContainEventsToo;
}
```

Interface Implementation

```
public class SimpleInterfaceImplementation : ISimpleInterface
{

    public void ThisMethodRequiresImplementation()
    {

    }

    public string ThisStringPropertyNeedsImplementingToo
    {
        get;
        set;
    }
    public int ThisIntegerPropertyOnlyNeedsAGetter
    {
        get
        {
             return this.encapsulatedInteger;
        }
        set
        {
            this.encapsulatedInteger = value;
        }
    }

    event EventHandler<EventArgs> InterfacesCanContainEventsToo = delegate { };

    private int encapsulatedInteger;
```

A class that implements multiple Interfaces

```
public interface IInterfaceOne
{
    void MethodOne();
}
// . . .
public interface IInterfaceTwo
{
    void MethodTwo();
}
// . . .
public class ImplementingMultipleInterfaces : IInterfaceOne, IInterfaceTwo
{
    public void MethodOne()
    {
    }

    public void MethodTwo()
    {
    }
}
```

Interface Explicit implementation

```
public interface IInterfaceOne
{
    void MethodOne();
}
```

```
public class ClassWithMethodSignatureClash
{
    public void MethodOne()
    {
    }
}
```

```
public class ClassWithMethodSignatureClash : IInterfaceOne
{
    public void MethodOne()
    {
    }
}
```

Interface Explicit implementation

```
public class ClassAvoidingMethodSignatureClash : IInterfaceOne
{
    public void MethodOne()
    {
        // original implementation
    }

    void IInterfaceOne.MethodOne()
    {
        // new implementation
    }
}
```

Interface Explicit implementation

```
public class ExplicitInterfaceImplementation : ISimpleInterface
{
    public ExplicitInterfaceImplementation()
    {
        this.encapsulatedInteger = 4;
    }

    void ISimpleInterface.ThisMethodRequiresImplementation()
    {
        encapsulatedEvent(this, EventArgs.Empty);
    }

    string ISimpleInterface.ThisStringPropertyNeedsImplementingToo
    {
        get;
        set;
    }

    int ISimpleInterface.ThisIntegerPropertyOnlyNeedsAGetter
    {
        get
        {
            return encapsulatedInteger;
        }
    }
```

Interface Explicit implementation

```
public class ExplicitInterfaceClient
{
    public ExplicitInterfaceClient(ExplicitInterfaceImplementation
        implementationReference, ISimpleInterface interfaceReference)
    {
        // Uncommenting this will cause compilation errors.
        //var instancePropertyValue =
        //implementationReference.ThisIntegerPropertyOnlyNeedsAGetter;
        //implementationReference.ThisMethodRequiresImplementation();
        //implementationReference.ThisStringPropertyNeedsImplementingToo = "Hello";
        //implementationReference.InterfacesCanContainEventsToo += EventHandler;

        var interfacePropertyValue =
            interfaceReference.ThisIntegerPropertyOnlyNeedsAGetter;
        interfaceReference.ThisMethodRequiresImplementation();
        interfaceReference.ThisStringPropertyNeedsImplementingToo = "Hello";
        interfaceReference.InterfacesCanContainEventsToo += EventHandler;
    }
```

Enums

- An enum is a special **value** type that lets you specify a group of named numeric constants.

```
public enum BorderSide { Left, Right, Top, Bottom }
```

- Each enum member has an underlying type **int** value: **0, 1, 2**... automatically assigned, in the declaration order of the members.
- You may specify an alternative integral type

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

- You may also specify an explicit underlying value for each member.

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```

Delegates

- A delegate is like a pointer to a method: a variable that references a method
- A delegate is a reference type and it holds the reference of a method.
- A delegate can be used to point to any method that has the same **return type** and **parameters** declared.
- A delegate can be declared using **delegate** keyword followed by a function signature as shown below.

<access modifier> **delegate** <return type> <delegate_name>(<parameters>)

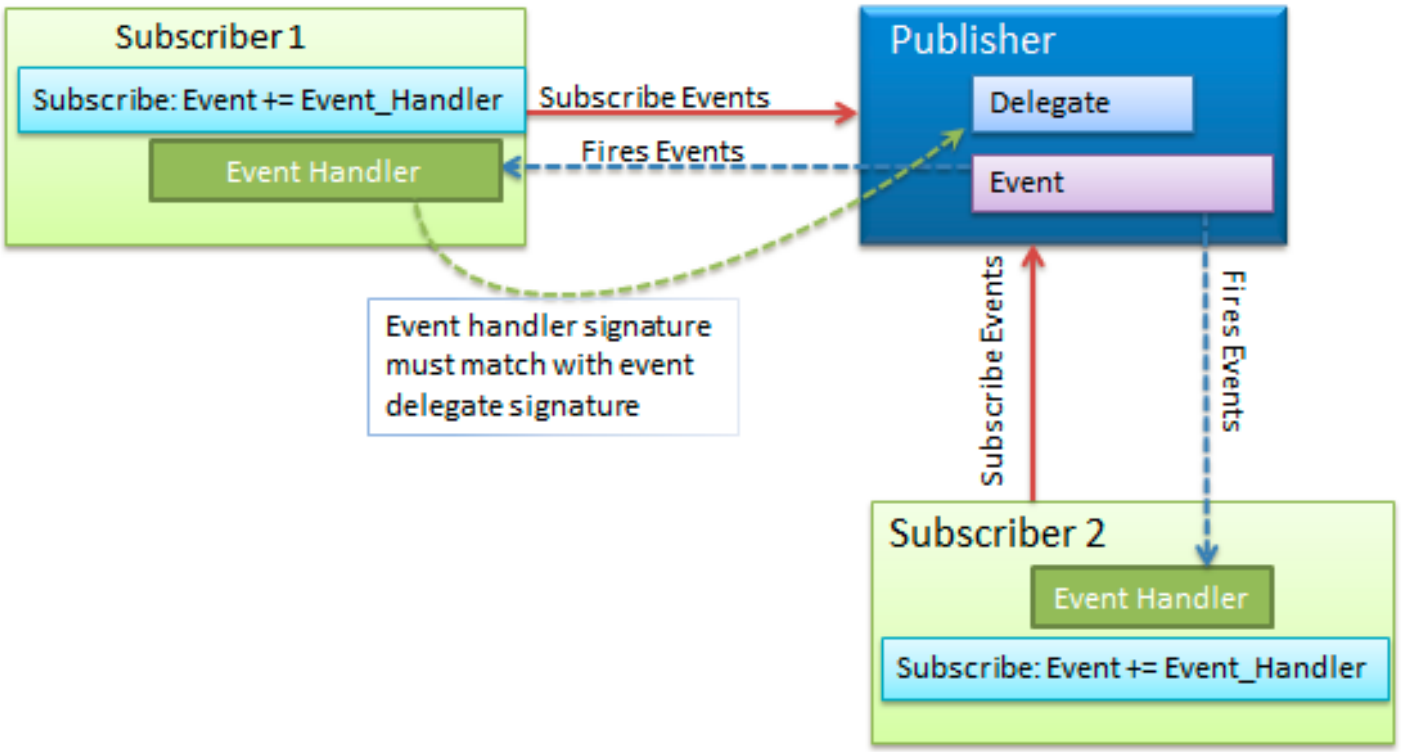
Events

- Events are a language feature that formalizes the broadcaster/subscriber pattern.
- An event is a construct that exposes just the subset of delegate features required for the broadcaster/subscriber model.
- The main purpose of events is to *prevent subscribers from interfering with one another*.
- Allows a class to send notification to other classes or objects
  - Publisher raises the event
  - One or more subscribers process the event

Events

- An event is nothing but an encapsulated **delegate**.
- The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class.
- The class containing the event is used to publish the event. This is called the **publisher** class.
- Some other class that accepts this event is called the **subscriber** class.
- Events use the **publisher-subscriber** model.

Events


Subscriber 1
Subscribe: Event += Event_Handler
Event Handler
Subscribe Events
Fires Events
Publisher
Delegate
Event
Event handler signature must match with event delegate signature
Subscribe Events
Fires Events
Subscriber 2
Event Handler
Subscribe: Event += Event_Handler