

# .Net Training

Intermediate Basics of .Net (C#)  
Part II

# Agenda

---

- Inheritance Review
- Class Modifiers (New, Virtual and Sealed keywords)
- Abstract Classes
- Interfaces
- Delegates
- Events



# Agenda

---

- Inheritance Review

# Inheritance Review

---

- C# allows us to inherit behavior from parent classes in child classes (subclasses)
- To inherit from another class we use the ":" notation (class B : A)
- When we inherit from a parent class we automatically get all the parent class' behavior and implementation by default
- But C# also allows us to control what can be inherited and how



# Agenda

---

- Class modifiers

# Inheritance – virtual keyword

---

- In order to allow overriding in child classes, methods and properties must be marked with the keyword **virtual**

```
class A
{
    virtual void DoWork()
    {...}
}
class B : A
{
    override void DoWork()
    {...}
}
```

- The child class that wants to override a virtual method or property must use the keyword **override**

# Inheritance – base keyword

---

- In order to access methods and properties from the base/parent class from within the child class use the base keyword:

```
class A
{
    virtual void DoWork()
    {...}
}
class B : A
{
    override void DoWork()
    {
        base.DoWork()
        ...
    }
}
```

# Inheritance – new keyword

---

- By default, when you inherit from a class you get all of its behavior and cannot override anything that is not **virtual**

```
class A
{
    void DoWork()
    {...}
}
class B : A
{
    new void DoWork()
    {...}
}
```

- In order to force the overriding in class B's implementation of DoWork() you have to use the **new** keyword

## Inheritance – sealed keyword

---

- In order to prevent a class from being used as a base class you can use the keyword **sealed**
- Sealed classes are primarily used to prevent derivation
- **Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster**

```
public sealed class D
{
    // Class members here.
}
```

## Inheritance – sealed keyword

---

- A class member, method, field, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed
- This negates the virtual aspect of the member for any further derived class
- This is accomplished by putting the **sealed** keyword before the override keyword in the class member declaration

## Inheritance – sealed keyword

---

```
public class C
{
    public virtual void DoWork() { }
}
public class D : C
{
    public sealed override void DoWork() { }
}
```

# Agenda

---

- Abstract Classes

# Abstract Classes

---

- Classes can be declared as abstract.
- This is accomplished by putting the keyword **abstract** before the keyword **class** in the class definition.

```
public abstract class A
{
    // Class members here.
}
```

- An abstract class cannot be instantiated.
- The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

# Abstract Classes

---

- Abstract classes may also define abstract methods. This is accomplished by adding the keyword **abstract** before the return type of the method.

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

- Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block.

# Abstract Classes

---

- Derived classes of the abstract class **must implement** all abstract methods.
- When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method.

# Abstract Classes

---

```
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}
public abstract class E : D
{
    public abstract override void DoWork(int i);
}
public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

# Abstract Classes

---

- If a virtual method is declared abstract, it is still virtual to any class inheriting from the abstract class.
- A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, DoWork on class F cannot call DoWork on class D.
- In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

# Abstract Classes

---

- In summary, an abstract class is a good way to defer implementation to child classes
- An abstract class can contain implementation, whereas an Interface cannot
- The .Net Framework uses many abstract classes
  - For example, .Net provides an abstract XmlReader class.
    - Implementation is available on XmlTextReader, XmlNodeReader and XmlValidationReader
    - Each work slightly differently, but all inherit from XmlReader so they have a common interface
  - Stream is also an abstract class
    - There are various implementations of Stream (FileStream, MemoryStream, NetworkStream, etc.)

# Equivalency

---

- You can think of an abstract class as a “white box” item
- Some manufacturer makes a generic “white box” computer, for example
- It has the basics – memory, hard-drive, motherboard, processor – but it leaves open slots for optical drives, video cards, monitor, mouse keyboard
- Then you can buy these in bulk and customize them
- Add in your optical drive of choice screen, mouse, keyboard
- Stick your company label on it and you have a custom PC to rival Dell



# Agenda

---

- Interfaces

# Interfaces

---

- An interface contains only the signatures of [methods](#), [delegates](#) or [events](#).
- The implementation of the methods is done in the class that implements the interface
- An interface can be a member of a namespace or a class and can contain signatures of the following members:
  - [Methods](#)
  - [Properties](#)
  - [Indexers](#)
  - [Events](#)
- An interface can inherit from one or more base interfaces



# Interfaces

---

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    public void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

# Equivalency

---

- The concept of an interface can be translated to the modern world
- Take a modern tap
- Taps turn clockwise and counter-clockwise to open or close
- Some move up or down
- These are well established interfaces
- People all over the world can interact with taps without special training
- Different manufacturers all over the world can come up with different styles, shapes, sizes, but because they follow well established interfaces for taps, people can still interact with them
- The first time somebody decided to make a tap that you didn't have to turn, instead reacted to the presence of your hand via lasers or optical beams, threw people off
- You had to stop, think, wonder where the tap was
- Then play around with the thing until you figured out it had a sensor that triggered the water
- Imagine your grandfather, without the concept of sensors, wondering at what demonic contraption the devil has spawned this time...

# Why?

---

- Why would you need Interfaces or Abstract Classes?
- You can write code that is more generic and more reusable
- You can take actions on different kinds of objects that have areas of commonality
- Use them and you'll understand



# Agenda

---

- Delegates

# Delegates

---

- A delegate is a type that references a method
- Once a delegate is assigned a method, it behaves exactly like that method
- The delegate method can be used like any other method, with parameters and a return value  
`public delegate int PerformCalculation(int x, int y);`
- Any method that matches the delegate's signature can be assigned to the delegate

# Delegates

---

- This makes it possible to programmatically change method calls
- Also plug new code into existing classes
- The ability to refer to a method as a parameter makes delegates ideal for defining callback methods
- For example, a sort algorithm could be passed a reference to the method that compares two objects
- Separating the comparison code allows the algorithm to be written in a more general way

# Delegates

---

- The following example declares a delegate named Del that can encapsulate a method that takes a string as an argument and returns void:

```
public delegate void Del(string message);
```

- Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate to that method

```
// Create a method for a delegate.
```

```
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
```

```
Del handler = DelegateMethod;
```

```
// Call the delegate.
```

```
handler("Hello World");
```

# Delegates

---

- Delegates are similar to C++ function pointers, but unlike C function pointers, delegates are object-oriented, type safe, and secure
- Delegates allow methods to be passed as parameters
- Delegates can be used to define callback methods
- Delegates can be chained together; for example, multiple methods can be called on a single event

# Delegates

---

- Delegate types are derived from the Delegate class in the .NET Framework
- Delegate types are sealed—they cannot be derived from— and it is not possible to derive custom classes from **Delegate**
- Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property
- This allows a method to accept a delegate as a parameter, and call the delegate at some later time

# Delegates

---

- This is known as an asynchronous callback, and is a common method of notifying a caller when a long process has completed
- When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used
- The functionality is similar to the encapsulation interfaces provide

# Delegates

---

```
public void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

- You can then pass the delegate created above to that method:

```
MethodWithCallback(1, 2, handler);
```

- and receive the following output to the console:

```
The number is: 3
```

# Delegates

---

- Using the delegate as an abstraction, *MethodWithCallback* does not need to call the console directly—it does not have to be designed with a console in mind
- What *MethodWithCallback* does is simply prepare a string and pass the string to another method
- This is especially powerful since a delegated method can use any number of parameters
- This concept is called **DECOUPLING**

# Delegates

---

- When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method
- A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature
- When a delegate is constructed to wrap a static method, it only references the method

# Multicast Delegates

---

- A delegate can call more than one method when invoked
  - This is referred to as multicasting
- To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+=')

```
Del allMethodsDelegate = Method1 + Method2;  
allMethodsDelegate += Method3;
```

- To remove a method from the invocation list, use the decrement or decrement assignment operator ('-' or '-=')

```
allMethodsDelegate -= Method2;
```

# Multicast Delegates

---

- Because delegate types are derived from **System.Delegate**, the methods and properties defined by that class can be called on the delegate
- For example, to find the number of methods in a delegate's invocation list, you may write:

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

# Delegates and Events

---

- Multicast delegates are used extensively in event handling
- Event source objects send event notifications to recipient objects that have registered to receive that event
- To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source
- The source calls the delegate when the event occurs
- The delegate then calls the event handling method on the recipient, delivering the event data.
- The delegate type for a given event is defined by the event source

# When to Use Delegates Instead of Interfaces

---

- Both delegates and interfaces allow a class designer to separate type declarations and implementation
- A given interface can be inherited and implemented by any class or struct;
- a delegate can be created for a method on any class, as long as the method fits the method signature for the delegate
- An interface reference or a delegate can be used by an object with no knowledge of the class that implements the interface or delegate method
- Given these similarities, when should a class designer use a delegate and when should they use an interface?

# When to Use Delegates Instead of Interfaces

---

- Use a delegate when:
  - An eventing design pattern is used
  - It is desirable to encapsulate a static method
  - The caller has no need access other properties, methods, or interfaces on the object implementing the method
  - Easy composition is desired
  - A class may need more than one implementation of the method

# When to Use Delegates Instead of Interfaces

---

- Use an interface when:
  - There are a group of related methods that may be called
  - A class only needs one implementation of the method
  - The class using the interface will want to cast that interface to other interface or class types
  - The method being implemented is linked to the type or identity of the class: for example, comparison methods

# When to Use Delegates Instead of Interfaces

---

- One good example of using a single-method interface instead of a delegate is `IComparable`
- **`IComparable`** declares the `CompareTo` method, which returns an integer specifying a less than, equal to, or greater than relationship between two objects of the same type
- **`IComparable`** can be used as the basis of a sort algorithm, and while using a delegate comparison method as the basis of a sort algorithm would be valid, it is not ideal
- Because the ability to compare belongs to the class, and the comparison algorithm doesn't change at run-time, a single-method interface is ideal

# Delegate Summary

---

- Delegates are a feature of Abstraction / Encapsulation
- Delegates are used for DECOUPLING type declarations and implementations
- Delegates are like pointers to methods
- Delegates are used in Events

# Command Pattern

---

- One of the most commonly used design patterns is called the Command Pattern
- The Command Pattern dictates that you separate the User Interface (UI) objects from the actions they must perform
- You see this pattern implemented in user controls, like Buttons

# Command Pattern

---

- The Button knows how to render itself and exposes events like Click
- When you use a button you provide your own “Click event handler”
- You do this by supplying your own method, to the event delegate of the Button
- The Button, when it detects a user Click will call the delegate you supplied, which can make the Button do different things depending on your custom program

# Command Pattern

- It is this separation of code (DECOUPLING) that makes delegates so powerful and objects so reusable in so many different scenarios





# Agenda

---

- Events

# Events

---

- An event is a way for a class to provide notifications when something of interest happens
- For example, a class that encapsulates a user interface control might define an event to occur when the user clicks on the control
- The control class does not care what happens when the button is clicked, but it does need to tell derived classes that the click event has occurred
- The derived classes can then choose how to respond
- Events use delegates to provide type-safe encapsulation of the methods that will be called when triggered

# How to create an Event

---

```
// Declare the handler delegate for the event
public delegate void ButtonEventHandler();
class TestButton
{
    // OnClick is an event, implemented by a delegate ButtonEventHandler.
    public event ButtonEventHandler OnClick;
    // A method that triggers the event:
    public void Click()
    {
        OnClick();
    }
}

// Create an instance of the TestButton class.
TestButton mb = new TestButton();
// Specify the method that will be triggered by the OnClick event.
mb.OnClick += new ButtonEventHandler(TestHandler);
// Trigger the event
mb.Click();

private void TestHandler()
{System.Console.WriteLine("Hello, World!"); }
```

# Events

---

- An event is a way for a class to notify objects that they need to perform an action of some kind
- The most common use for events is in graphical user interfaces, although events can be useful at other times, such as signaling state changes
- Events are usually declared using [delegate](#) event handlers