# Agenda

- Introduction to Abstract Classes
- Introduction to Interfaces
- Abstract Classes vs. Interfaces
- Abstract Classes vs. Sealed Classes

# Abstract Class

- Abstract Class:
  - An abstract class cannot be instantiated directly.
  - A non-abstract class is derived from an abstract class, the non abstract class must include actual implementations of inherited abstract members, thereby overriding the abstract members.
  - An abstract modifier indicates that the thing being modified has missing or incomplete implementation.
  - An abstract modifier can be used with classes, methods, properties, indexes, and events.
  - Abstract modifiers are used in class declarations to indicate that a class is intended only to be a base class of other classes.

# Rules on Abstract Classes

- Abstract classes have the following features:
  - An Abstract class cannot be instantiated.
  - An Abstract class can contain abstract methods and assessors.
  - Abstract modifiers are used in a method or property declaration to indicate that the method or property does not contain implementation.
  - They cannot have a sealed access modifier, as this would make inheritance impossible.

# Rules on Abstract Classes (cont.)

- An abstract modifier can be used in a method or property declaration to indicate that the method or property does not contain implementation.
- An abstract method:
  - Is implicitly a virtual method.
  - Declaration is only permitted in abstract classes because abstract method declarations provide no actual implementation.
  - It is an error to use the static or virtual modifiers in an abstract method declaration.

# Abstract Class - Example

```
using System;
// Abstract class
abstract class MyBaseC
{
      protected int x = 100;
      protected int y = 150;
      // Abstract method
      public abstract void MyMethod();

      // Abstract property
      public abstract int GetX
      {
         get;
      }
}
```

```
x = 111
```

```
class MyDerivedC: MyBaseC
{
      public override void MyMethod()
      {
          x++;
      }
      // overriding property
      public override int GetX
      {
            get
            {
            return x + 10;
            }
      }
      public static void Main()
      {
        MyDerivedC mC = new MyDerivedC();
        mC.MyMethod();
        Console.WriteLine("x = {0}”,
         mC.GetX);
      }
}
```

# Interface

- An Interface:
  - Describes a group of related functionalities that can belong to any class or struct.
  - Consists of methods, properties, events, indexers, or any combination of these four member types.
  - Provides no functionality by itself that a class or struct can inherit. The class or struct provides an implementation for all the members defined by the interface.

# Rules on Interface (1 of 3)

- An Interface:
  - Can inherit from multiple base interfaces.
  - Can be implemented by any class.
  - Cannot implement any data type.
  - Cannot be instantiated.
  - Can contain methods, properties, events, and indexers
  - Itself does not provide implementations for the members that it defines.
  - Merely specifies that members must supply classes or structs to implement the interface.
- A class or struct can implement several interfaces, thus enabling multiple inheritance.

# Rules on Interface (2 of 3)

- A class that implements an interface partially must be declared abstract.
- Valid interface modifiers include new, public, protected, internal and private.
- The explicit base interfaces of an interface must be at least as accessible as the interface itself.
- An interface cannot contain constants, fields, operators, instance constructors, destructors, types, and static members of any kind.

# Rules on Interface (3 of 3)

- The base interfaces of an interface are the explicit base interfaces and their base interfaces in turn.

```
interface IControl
{
            void Paint();
}
interface ITextBox: IControl
{
            void SetText(string text);
}
interface IListBox: IControl
{
            void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {
}
```

- In this example the base interfaces of IComboBox are IControl, ITextBox, and IListBox.
- In other words, the IComboBox interface above inherits members SetText and SetItems as well as Paint.

- All interface members implicitly have public access.

# Implementing an Interface

- Interfaces are implemented using a “:”.
- Rules for implementing interface methods are:
  - They must have the same method signature and return type.
  - They cannot narrow the method accessibility.
- Interface methods are implicitly public.

# Interface - Example

```
public abstract class AccountInfo
{
     private double balance;
     public double Balance
         {
             get
             {
                 return balance;
             }
             set
             {
                 balance = value;
             }
         }
     public abstract void PrintAccountInfo();
}
public class Savings : AccountInfo, BankOperations
{
       public void Withdraw(double amount)
       {
           Balance = Balance - amount;
       }
       public void Deposit(double amount)
       {
           Balance = Balance + amount;
       }
       public double BalanceInquiry()
       {   return Balance;            }
      public void PrintAccountInfo()
      {
           Console.WriteLine("Account Balance: " + Balance);
       }
}
```

```
public interface BankOperations
{
    public void Withdraw(double amount);
    public void Deposit(double amount);
    public double BalanceInquiry();
}
```

```
public class BankApp
{
 public static void Main(String[] args)
 {
    Savings pesoAcct = new Savings();
    pesoAcct.Balance = 500;
    pesoAcct.PrintAccountInfo();
    pesoAcct.Deposit(300);
    pesoAcct.Withdraw(50);
    Console.WriteLine("Updated Balance: " +
    pesoAcct.BalanceInquiry());
  }
}
```

```
Account Balance: 500.0
Updated Balance: 750.0
```

# Uses of Interface: Multiple Inheritance

- A core reason for using an interface is to be able to upcast to more than one base type.
- Several interfaces can be implemented by any class, thus resulting in multiple inheritances.

```
interface ICanFight
{
    void Fight();
}
interface ICanSwim
{
    void Swim();
}
interface ICanJump
{
    void Jump();
}
```

```
class ActionStar : ICanFight
 {
   public void Fight()
    {
       //fight here
    }
}
```

```
class Hero: ActionStar, ICanSwim, ICanJump
{
    public void Swim()
    {
            //swim here
    }
    public void Jump()
    {
            //jump here
    }
}
```

# Abstract Class vs. Interface

- Considerations in using either an interface or an abstract class:
  - An interface, by contract, is a totally abstract set of members that can be thought of as defining a contract.
  - Both interfaces and abstract classes are useful for component interactions.
  - An interface is useful because any class can implement it. However, if another method is added to an interface, all classes that implement that interface will be broken.
  - A good implementation for both an abstract class and an interface is to create an interface and let the abstract class implement it. Therefore, when there is a need for adding methods, they can be safely added to the abstract class itself rather than the interface.
  - Use interfaces when a certain method needs to be forcibly overridden or enforced by a class.

# Sealed Class

- A sealed class:
  - Is considered complete; it cannot be improved or specialized.
  - Ensures that its state and behavior cannot be changed for safety and security.
  - Can be re-used by utilizing composition / aggregation, instead of inheritance.

# Rules on Sealed Classes

- A sealed class cannot be extended or sub-classed.
- A sealed class cannot also be an abstract class. In other words, all methods of a sealed class have implementations.
- All methods of a sealed class are implicitly sealed methods.

# Sealed Class Example

```
sealed class Formula
{
    static double Speed(double distance,
    double time)
    {
            return distance/time;
    }
    static double Acceleration(double s2,
    double s1, double t)
    {
            return (s2-s1)/t;
    }
    static double Force(double mass, double
    acc)
    {
            return mass * acc;
    }
    static double Pressure(double force,
    double area)
    {
            return force / area;
    }
    static double Work(double force,
    distance)
    {
            return force * distance;
    }
}
```

```
  public static void Main(String[] args)
  {
    double d1 = 50, t1 = 10;
    double d2 = 80, t2 = 10;
    double mass = 50;
    double s1 = Formula.Speed(d1, t1);
    double s2 = Formula.Speed(d2, t2);
    double acc = Formula.Acceleration(s2, s1, 60);
    double force = Formula.Force(mass, acc);
    double pressure = Formula.Work(force,10);
    Console.WriteLine("If I weigh " + mass + " kg,
            and");
    Console.WriteLine("\t my initial speed is " +
s1 +                  " m/s, and");
    Console.WriteLine("\t my final speed is " + s2
+ "        m/s.");
    Console.WriteLine("Then, my acceleration in 1
            minute is " + acc + " m/s2, and");
    Console.WriteLine("\t I'm exerting a force of "
```

```
If I weigh 50.0 kg, and
 my initial speed is 5.0 m/s, and
 my final speed is 8.0 m/s.
Then, my acceleration in 1 minute is 0.05 m/s2, and
 I'm exerting a force of 2.5 newton, and
 a pressure of 25.0 joules for 10 meters!
Ahem ;-)
```

# Abstract Class vs. Sealed Class

- Abstract classes provide templates of behaviors for subclasses to follow without actually dictating to the subclasses the exact implementation of those behaviors.
- Sealed classes, since they cannot be extended, encourage composition, while abstract classes encourage inheritance.
- Inheritance through abstract classes introduces tight coupling between classes and can weaken encapsulation by deferring actual implementation to subclasses.
- Sealed classes are used to prevent malicious code from altering the semantics of classes essential to a framework.