The Learning House

427 S. 4th Street #300

Louisville KY 40202

Classes and Inheritance

.NET Cohort

Coding Bootcamp

SOFTWARE GUILD

Lesson Goals

- Understand the pillars of Object-Oriented Programming (OOP)
- Learn how to inherit classes to extend and reuse functionality
- Introduce the remaining access levels

The Pillars of OOP

- All object-oriented programming languages must contend with three core principles:
  - *Encapsulation* — How does the language hide the internal implementation details and preserve data integrity?
  - *Inheritance* — How does the language promote code reuse?
  - *Polymorphism* — How does the language let you treat related objects in a similar way?

Word Of Warning

Most developers don't understand this stuff. If you get good at it, congrats, you're a senior-level developer.

The Role of Encapsulation

- Objects should hide their implementation details from the caller.
- This allows the developer to keep code simpler to read and understand.
- This also allows the developer to protect the state of an object.

```
var myFile = new FileReader();

myFile.Open(@"c:\data.txt");

// do stuff with the file

myFile.Close();
```

The Role of Inheritance

- Inheritance allows the developer to build class definitions based on existing definitions. This is called *extending a base class*.
- Inheritance establishes an *is-a* relationship between types:
  - A Rectangle is-a Shape that is-an Object
- In .NET, every class starts with System.Object.

Has-a Relationships

This is a form of reuse based on containment and delegation.

Has-a relationships allow one class to define a member variable of another class and expose its functionality indirectly.

Thus, in the example to the right, the car has-a radio.

Has-a relationships are also a key component of encapsulation.

```
class Radio
{
    public void TurnOn()
    {
        Console.WriteLine("Radio is turned on!");
    }
}

class Car
{
    private Radio myRadio = new Radio();

    public void TurnOnRadio()
    {
        myRadio.TurnOn();
    }
}

class Program
{
    static void Main()
    {
        var myCar = new Car();
        myCar.TurnOnRadio();
    }
}
```

The Role of Polymorphism

- Polymorphism allows a base class to define a set of members (*polymorphic interface*) that is available to all descendants.
- *Virtual members* are members that provide a default implementation that may be changed or *overridden* by derived classes.
- *Abstract members* provide no implementation and **must** be overridden by derived classes

Polymorphism

DEMO

Extending Classes via Inheritance

- You can use an existing class, called the *base class,* as the basis for a new class, called the *derived class*.
  - The derived class can access members of its own class and members of the base class.
- Declaring a derived class requires a colon followed by the type of the base class.
- A derived class is said to *extend* the base class because it can add additional functionality.
- A derived class can not delete any of the members it has inherited.
- You can access members of the base class the same way you access other members.

Basic Inheritance

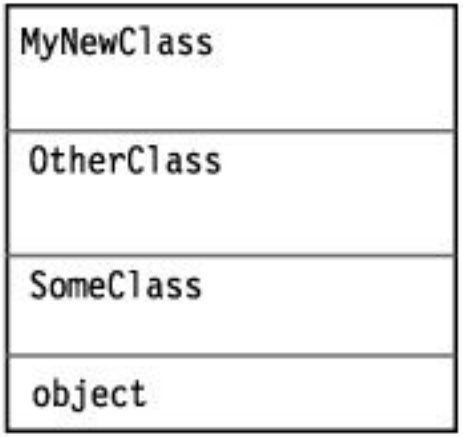
DEMO

All Classes Derive from System.Object

```
class SomeClass
{ ... }

class OtherClass: SomeClass
{ ... }

class MyNewClass: OtherClass
{
      ...
}
```



System.Object Members

Name	Description
Equals(Object)	Determines whether the specified object is equal to the current object.
Equals(Object, Object)	Determines whether the specified object instances are considered equal.
Finalize	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
GetHashCode	Serves as a hash function for a particular type.
GetType	Gets the Type of the current instance.
MemberwiseClone	Creates a shallow copy of the current Object.
ReferenceEquals	Determines whether the specified Object instances are the same instance.
ToString	Returns a string that represents the current object.

System.Object overrides

DEMO

Masking Base Class Members

- Although a derived class cannot delete a member it has inherited, the class can mask it by re-declaring it a member.
  - To mask an inherited data member, declare a new member of the same type and same name.
  - To mask a function member, declare a new function member with the same signature.
  - To let the complier know you're doing it on purpose, use the *new* modifier. It will still work if you don't, but it will throw warnings when you compile.
  - You can also mask static members.
- The *base* keyword lets you access the base class member.

Masking and base

DEMO

Virtual and Override Members

- The default behavior is: if you reference a base class, you only get members of the base class.
- By declaring a method virtual and overriding it, you can call into the derived class from a reference to a base class.
- You can have many overrides in the inheritance chain, and the highest override will win.
- Virtual/Overrides work with methods, properties, events, and indexers.

Masking vs Virtual

In the code sample to the left, we have two print methods: one is virtual, one is not.

When we reference the base class, but assign it a derived class, and call the two print functions, the following output will be displayed:

Hello from the derived class override
Hello from the base class non-virtual

Thus, the virtual method called the derived class method and the masked method used the base class method.

This is a foundational principle with inheritance. You can mix masking and virtual overrides for some interesting results.

```
public class BaseClass
{
    public virtual void Print()
    {
        Console.WriteLine("Hello from the base class virtual");
    }

    public void Print2()
    {
        Console.WriteLine("Hello from the base class non-virtual");
    }
}

public class DerivedClass : BaseClass
{
    public override void Print()
    {
        Console.WriteLine("Hello from the derived class override");
    }

    new public void Print2()
    {
        Console.WriteLine("Hello from the derived class new");
    }
}
```

```
public void BaseVsVirtual()
{
    BaseClass b1 = new DerivedClass();

    b1.Print();
    b1.Print2();
}
```

Constructors and Inheritance

1. Members get initialized
2. Base Class constructor executes
3. Derived class constructor executes

WARNING: Calling virtual methods in constructors of derived classes can cause issues (because the derived class's constructor wouldn't be run yet). As such, this is strongly discouraged.

```
class BaseClass
{
    public BaseClass()          // 2
    {

    }
}

class DerivedClass
{
    int num1 = 5;               // 1
    int num2;

    public DerivedClass()       // 3
    {

    }
}
```

Constructor Initializers

- By default, the parameter-less constructor is called when an object is instantiated. In the case of multiple constructors, you can specify which one to use.
  - Use the *base* keyword to specify which base constructor to use.
  - Use the *this* keyword to specify which other constructors in the current class to use.

Using base() and this()

```
class Player
{
    public string PlayerName { get; set; }
    public int Level { get; set; }

    public Player()
    {
        Level = 1;
    }

    public Player(string name) : this()
    {
        PlayerName = name;
    }
}
```

```
class Vehicle
{
    public readonly int MaxSpeed;

    public Vehicle()
    {
        MaxSpeed = 5;
    }

    public Vehicle(int maxSpeed)
    {
        MaxSpeed = maxSpeed;
    }
}

class GolfCart : Vehicle
{

}

class Car : Vehicle
{
    public Car() : base(85)
    {

    }
}
```

Five Member Access Levels

Modifier	Meaning
private	Accessible only within the current class
internal	Accessible to all classes within the assembly (project)
protected	Accessible to the current class and all derived (child) classes
protected internal	Accessible to the current class, all derived (child) classes, and all classes in the same assembly
public	Accessible to any class in any assembly

Can You See Me?

	Classes in Same Assembly		Classes in Different Assembly	
	Non-derived	Derived	Non-derived	Derived
private				
internal	yes	yes		
protected		yes		yes
protected internal	yes	yes		yes
public	yes	yes	yes	yes

Abstract Members

- *Abstract Members* are function members marked with the *abstract* keyword.
- They must not have an implementation. The signature ends with a semicolon.
- Any class that is derived must implement the abstract member.
- Abstract members can only exist in *abstract classes.*
- Since there is no implementation code, you cannot instantiate an abstract class directly, only the derived class(es).

Abstract Class / Method Example

Our previous shape example is a good usage of abstract.

“Does it make sense to create the base class on its own?”

In this case, we don’t know what kind of shape it is and therefore don’t know how to draw it, so we make the class and the draw method abstract to ensure that no one says:

Shape myShape = new Shape();

We still get the benefit of polymorphism!

```
abstract class Shape
{
    abstract public void Draw();
}

class Triangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("drawing the triangle!");
    }
}


    public void DrawShapes()
    {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Triangle();
        shapes[1] = new Square();
        shapes[2] = new Triangle();

        foreach (var s in shapes)
            s.Draw();
    }
```

Virtual vs Abstract

	Virtual Member	Abstract Member
Keyword Used	virtual	abstract
Implementation Code Block	Has a code block	Does not have a code block, definition ends with semicolon
Overridden in a derived class?	**May** be overridden using the override keyword	**Must** be overridden using the override keyword
Can be used on	Methods Properties Events Indexers	Methods Properties Events Indexers

Sealed Classes

- Whereas an abstract class can not be instantiated and must be inherited from, a *sealed* class cannot be inherited from.
- Most developers don't do this unless they're doing API coding and don't want other people tampering with their classes.

```
public sealed class DoNotInherit
{

}
```

In Conclusion

- Object-oriented programming uses encapsulation, inheritance, and polymorphism.
- We can inherit and extend our classes in various ways, using or overriding members throughout the inheritance chain.
- We can define default implementations or use the abstract keyword to require overrides.
- C# has five different levels of access keywords to control who can use your classes and their members.