C# Intermediate: Classes, Interfaces and OOP

By: Mosh Hamedani

www.programmingwithmosh.com

CLASSES

Introduction

- Classes are building blocks of software applications.
- A class encapsulates data (stored in fields) and behaviour (defined by methods).

```
public class Customer
{
        // Field
        public string Name;

        // Method
        public void Promote()
        {
        }
}
```

- An object is an instance of a class. We can create an object using the new operator.

```
Customer customer = new Customer();

// Or
var customer = new Customer();
```

Constructors

- A constructor is a method that is called when an instance of a class is created.
- We use constructors to put an object in an early state.

- As a best practice, define a constructor only when an object “needs” to be initialised or it won’t be able to do its job.
- Constructors do not have a return type, not even void, and they should have the exact same name as the class.
- A quick way to create a constructor: type **ctor** and press tab. This is a code snippet.

  If you wanna learn more ways to write code fast, check out my course: “Double Your Coding Speed” on Udemy:

  https://www.udemy.com/visual-studio-tips-tricks/

- Constructors can be overloaded. Overloading means creating a method with the same name and different signatures.
- Signature of a method consists of the number, type and order of its parameters.
- We can pass control from one constructor to the other by using the **this** keyword.

```
public class Customer
{
        public int Id;
        public string Name;
        public List<Order> Orders;

        // Default or parameterless constructor
        public Customer()
        {
             // Orders has to be initialized here, otherwise it
             // will be a null reference. As a best practice,
             // anytime your class contains a list, always
             // initialize the list.
             Orders = new List<Order>();
        }

        public Customer(int id)
           : this()  // Calls the default constructor
        {
              this.Id = id;
        }
```

```
}
```

Methods

- Signature of a method consists of the number, type and order of its parameters.
- Overloading a method means having a method with the same name but with different signatures. This makes it easier for the callers of the method to choose the more suitable signature depending on the type of data they have to pass to the method.

```
public class Point
{
	public void Move(int x, int y) {}

	// The Move method overloaded here
	public void Move(Point newLocation) {}
}
```

- We can use the **params** modifier to give a method the ability to receive varying number of parameters.

```
public class Calculator
{
	public int Add(params int[] numbers) {}
}

…

var result = calculator.Add(1, 2, 3, 4);
```

- By default, when we pass a value type (eg int, char, bool) to a method, a copy of that variable is sent to the method. So changes applied to that variable in the method will not be visible upon return from the method. This can be modified using the **ref** modifier. When we use the ref modifier, a reference to the original variable will be sent to the target method.

  The ref modifier, in my opinion, is a smell in the design of the C# language. Please

don’t use it when defining your methods.

```
public void Weirdo(ref int a)
{
      a += 2;
}

…

var a = 1;
Weirdo(ref a);
// Here a will be 3.
```

- The **out** modifier can be used to return multiple values from a method. Any parameter declared with the out modifier is expected to receive a value at the end of the method.

  Again, this is a design smell and I’m totally against that. Don’t use it while declaring your methods.

```
public void Weirdo(out int a)
{
      a = 1;
}

…

int a;
Weirdo(out a);
```

Fields

- A field can be initialized in two ways: In a constructor, or directly upon declaration. The benefit of initialising a field during declaration is that if your class has one or more constructors, you’ll make sure that the field will always be initialised irrespective of which constructor is going to be called.

```
public class Customer
```

```
{
        public List<Order> Orders = new List<Order>();
}
```

- We use the **readonly** modifier to improve the robustness of our code. When a field is declared with readonly, it needs to be initialized either during declaration or in a constructor. The value cannot be changed. This prevents you from accidentally overwriting the value of a field, which can result in an unexpected state. As an example, think of the Orders in the above example. If we accidentally re-initialize this field somewhere else in the class, all the Order objects stored in the list will be lost. So we should declare it as readonly:

```
public class Customer
{
        public readonly List<Order> Orders = new List<Order>();
}
```

Access Modifiers

- In C# we have 5 access modifiers: **public**, **private**, **protected**, **internal** and **protected internal**.
- A class member declared with **public** is accessible everywhere.
- A class member declared with **private** is accessible only from inside the class.
- We'll learn about the other access modifiers when we get to the inheritance.
- We use access modifiers to hide the implementation details of a class. So anything that is about "how" a class does its job should be declared as private. This way, we make sure other parts of the code will not touch the implementation detail of a class. And as a result we improve the robustness of our code. If change the implementation of a class, we only need to make changes inside the class. No other parts of the code will need to be changed.

Properties

- A property is a kind of class member that is used for providing access to fields of a class.

- As a best practice, we must declare fields as private and create public properties to provide access to them.
- A property encapsulates a get and a set method:

```
public class Customer
{
      private string _name;

      public string Name
      {
            get { return _name; }
            set { _name = value; }
      }
}
```

- Inside the get/set methods we can have some logic.
- If you don't need to write any specific logic in the get or set method, it's more efficient to create an auto-implemented property. An auto-implemented property encapsulates a private field behind the scene. So you don't need to manually create one. The compiler creates one for you:

```
public class Customer
{
     public string Name { get; set; }
}
```

Indexers

- Indexer is a special kind of property that allows accessing elements of a list by an index.
- If a class has the semantics of a list, or collection, we can define an indexer property for it. This way it's easier to get or set items in the collection.

```
public class HttpCookie
{
      public string this[string key]
```

```
        {
                get {}
                set {}
        }
}
```

C# Intermediate: Classes, Interfaces and OOP

By: Mosh Hamedani

www.programmingwithmosh.com

ASSOCIATION BETWEEN CLASSES

Class Coupling

- A measure of how interconnected classes and subsystems are.
- The more coupled classes, the harder it is to change them. A change in one class may affect many other classes.
- Loosely coupled software, as opposed to tightly coupled software, is easier to change.
- Two types of relationships between classes: Inheritance and Composition.

Inheritance

- A kind of relationship between two classes that allows one to inherit code from the other.
- Referred to as Is-A relationship: A Car is a Vehicle
- Benefits: code re-use and polymorphic behaviour.

```
public class Car : Vehicle
{
}
```

Composition

- A kind of relationship that allows one class to contain another.
- Referred to as Has-A relationship: A Car has an Engine

- Benefits: Code re-use, flexibility and a means to loose-coupling

```
public class DbMigrator
{
      // We re-use the code in the logger class without
      // the need to repeat that logic here in DbMigrator
      private Logger _logger;
}
```

Favour Composition over Inheritance

- Problems with inheritance:
  - Easily abused by amateur designers / developers
  - Leads to large complex hierarchies
  - Such hierarchies are very fragile and a change may affect many classes
  - Results in tight coupling
- Benefits of composition:
  - Flexible
  - Leads to loose coupling
- Having said all that, it doesn't mean inheritance should be avoided at all times. In fact, it's great to use inheritance when dealing with very stable classes on top of small hierarchies. As the hierarchy grows (or variations of classes increase), the hierarchy, however, becomes fragile. And that's where composition can give you a better design.

C# Intermediate: Classes, Interfaces and OOP

By: Mosh Hamedani

www.programmingwithmosh.com

INTERFACES

Introduction

- An interface is a language construct that is similar to a class (in terms of syntax) but is fundamentally different.
- An interface is simply a declaration of the capabilities (or services) that a class should provide.

```
public interface ITaxCalculator
{
        int Calculate();
}
```

- This interface states a class that wants to play the role of a tax calculator, should provide a method called Calculate() that takes no parameters and returns an int. The implementation of this class might look like this:

```
public class TaxCalculator : ITaxCalculator
{
      public void Calculate() { … }
}
```

- So an interface is purely a declaration. Members of an interface do not have implementation.
- An interface can only declare methods and properties, but not fields (because fields are about implementation detail).
- Members of an interface do not have access modifiers.

- Interfaces help building loosely coupled applications. We reduce the coupling between two classes by putting an interface between them. This way, if one of these classes changes, it will have no impact on the class that is dependent on that (as long as the interface is kept the same).

Interfaces and Testability

- Unit testing is part of the automated practice which helps improve the quality of our code. With automated testing, we write code to test our own code. This helps catching bugs early on as we change the code.
- In order to unit test a class, we need to isolate it. This means: we need to assume that every other class in our application is working properly and see if the class under test is working as expected.
- A class that has tight dependencies to other classes cannot be isolated.
- To solve this problem, we use an interface. Here is an example:

```
public class OrderProcessor
{
	private IShippingCalculator _calculator;

	public Customer(IShippingCalculator calculator)
	{
		_calculator = calculator;
	}

	…
}
```

- So here, OrderProcessor is not dependent on the ShippingCalculator class. It's only dependent on an interface (IShippingCalculator). If we change the code inside the ShippingCalculator (eg add a new method or change the method implementations) it will have no impact on OrderProcessor (as long as the interface is kept the same).

Interfaces and Extensibility

- We can use interfaces to change our application's behaviour by "extending" its code (rather than changing the existing code).
- If a class is dependent on an interface, we can supply a different implementation of that interface at runtime. This way, the behaviour of the application changes without any impact on that class.
- For example, let's assume our **DbMigrator** class is dependent on an **ILogger** interface. At runtime, we can supply a **ConsoleLogger** to log the messages on the console. Later, we may decide to log the messages in a file (or a database). We can simply create a new class that implements the **ILogger** interface and inject it into **DbMigrator**.

Interfaces and Inheritance

- One of the common misconceptions about interfaces is that they are used to implement multiple inheritance in C#. This is fundamentally wrong, yet many books and videos make such a false claim.
- With inheritance, we write code once and re-use it without the need to type all that code again.
- With interfaces, we simply declare the members the implementing class should contain. Then we need to type all that declaration along with the actual implementation in that class. So, code is not inherited, even the declaration of the members!

C# Intermediate: Classes, Interfaces and OOP

By: Mosh Hamedani

www.programmingwithmosh.com

POLYMORPHISM

Method Overriding

- Method overriding means changing the implementation of an inherited method.
- If a declare a method as **virtual** in the base class, we can **override** it in a derived class.

```
public class Shape
{
    public virtual Draw()
    {
        // Default implementation for all derived classes
    }
}

public class Circle : Shape
{
    public override Draw()
    {
        // Changed implementation
    }
}
```

- This technique allows us to achieve polymorphism. Polymorphism is a great object-oriented technique that allows use get rid of long procedural switch statements (or conditionals).

Abstract Classes and Members

- Abstract modifier states that a class or a member misses implementation. We use abstract members when it doesn't make sense to implement them in a base class. For example, the concept of drawing a shape is too abstract. We don't know how to draw a shape. This needs to be implemented in the derived classes.
- When a class member is declared as abstract, that class needs to be declared as abstract as well. That means that class is not complete.
- In derived classes, we need to override the abstract members in the base class.

```
public abstract class Shape
{
    // This method doesn't have a body.
    public abstract Draw();
}

public class Circle : Shape
{
    public override Draw()
    {
        // Changed implementation
    }
}
```

- In a derived class, we need to override all abstract members of the base class, otherwise that derived class is going to be abstract too.
- Abstract classes cannot be instantiated.

Sealed Classes and Members

- If applied to a class, prevents derivation from that class.
- If applied to a method, prevents overriding of that method in a derived class.

- The string class is declared as sealed, and that’s why we cannot inherit from it.

```
public sealed class String
{
}
```

C# Intermediate: Classes, Interfaces and OOP

By: Mosh Hamedani

www.programmingwithmosh.com

INHERITANCE

Access Modifiers

- Your classes should be like a black box. They should have limited visibility from the outside. The implementation, the detail, should be hidden. We use access modifiers (mostly private) to achieve this. This is referred to as Information Hiding (and sometimes Encapsulation) in object-oriented programming.
- Public: A member declared as public is accessible everywhere.
- Private: A member declared as private is accessible only from the class.
- Protected: A member declared as protected is accessibly only from the class and its derived classes. Protected breaks encapsulation (because the implementation details of a class will leak into its derived classes) and is better to be avoided.
- Internal: A member declared as internal is accessibly only from the same assembly.
- Protected Internal: A member declared as protected internal is accessible only from the same assembly or any derived classes. (Sounds weird? Forget it! It's not really used.)

Constructors and Inheritance

- Constructors are not inherited and need to explicitly defined in derived class.
- When creating an object of a type that is part of an inheritance hierarchy, base class constructors are always executed first.
- We can use the base keyword to pass control to a base class constructor.

```
public class Car : Vehicle
{
    public Car(string registration) : base(registration)
    {
    }
}
```

Upcasting and Downcasting

- Upcasting: conversion from a derived class to a base class
- Downcasting: conversion from a base class to a derived class
- All objects can be implicitly converted to a base class reference.

```
// Upcasting
Shape shape = circle;
```

- Downcasting requires a cast.

```
// Downcasting
Circle circle = (Circle)shape;
```

- Casting can throw an exception if the conversion is not successful. We can use the **as** keyword to prevent this. If conversion is not successful, null is returned.

```
Circle circle = shape as Circle;
if (circle != null) …
```

- We can also use the **is** keyword:

```
if (shape is Circle)
{
    var circle = (Circle) shape;
    …
```

```
}
```

Boxing and Unboxing

- C# types are divided into two categories: value types and reference types.
- Value types (eg int, char, bool, all primitive types and struct) are stored in the stack. They have a short life time and as soon as they go out of scope are removed from memory.
- Reference types (eg all classes) are stored in the heap.
- Every object can be implicitly cast to a base class reference.
- The object class is the parent of all classes in .NET Framework.
- So a value type instance (eg int) can be implicitly cast to an object reference.
- Boxing happens when a value type instance is converted to an object reference.
- Unboxing is the opposite: when an object reference is converted to a value type.
- Both boxing and unboxing come with a performance penalty. This is not noticeable when dealing with small number of objects. But if you're dealing with several thousands or tens of thousands of objects, it's better to avoid it.

```
// Boxing
object obj = 1;

// Unboxing
int i = (int)obj;
```