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 {}
        }
}
```