C# Object Oriented Programming

Notes
rajeeshvengalapurath@gmail.com

What is OOP?

Object-oriented programming is about creating objects that contain both data and methods.

What are Classes and Objects?

A class is a template for objects, and an object is an instance of a class. A Class is like an object constructor, or a "blueprint" for creating objects.

Class Members

Fields and methods inside classes are often referred to as Class Members

Field or attribute

When a variable is declared directly in a class, it is often referred to as a field (or attribute).

Access Modifiers

Access Modifiers are used to control the visibility of class members and to achieve "Encapsulation" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as private.

public	The code is accessible for all classes
private	The code is only accessible within the same class
protected	The code is accessible within the same class, or in a class that is inherited from that class
internal	The code is only accessible within its own assembly, but not from another assembly

Encapsulation

Encapsulation is an OOPs concept to create and define the restrictions and permissions of an object and its member variable and methods. It is very simple to explain the concept is to

make the member variables of a class private and providing public getter and setter methods.

- declare fields/variables as private
- provide public get and set methods, through properties, to access and update the value of a private field

Why Encapsulation?

- Better control of class members (reduce the possibility of yourself (or others) to mess up the code)
- Fields can be made read-only (if you only use the get method), or write-only (if you only use the set method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

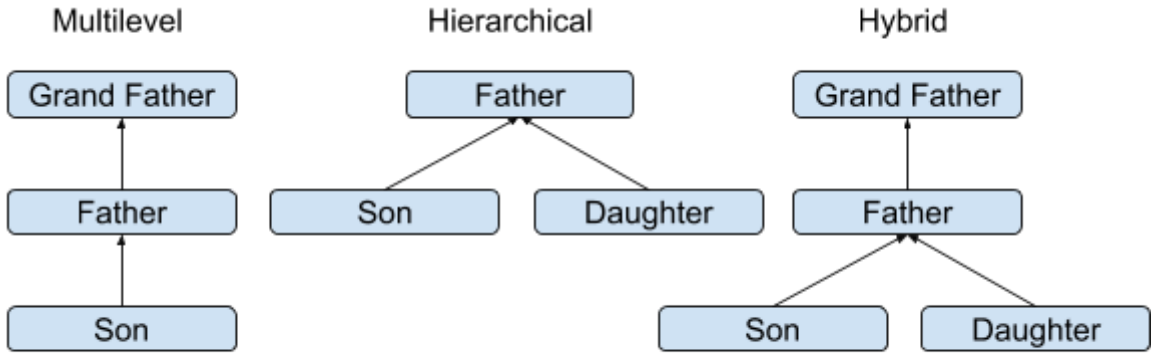
Automatic Properties (Shorthand)

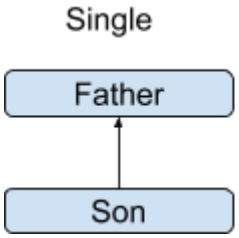
```
class Person
{
	public string Name { get; set; }
}
```

Inheritance

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

Inheritance Types

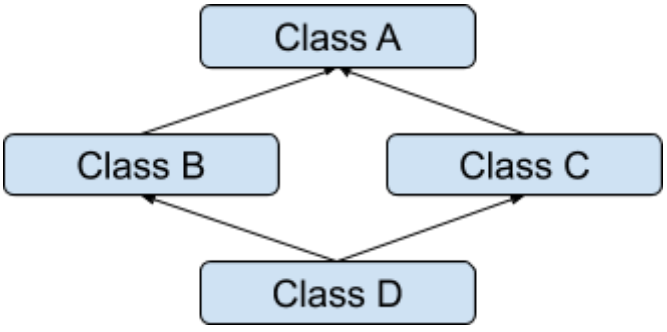




Problems with Multiple Class Inheritance (Diamond Problem)

- Class B and class C inherit from class A
- Class D inherits from both B and C
- If a method in D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B or C?
- This ambiguity is called as Diamond Problem

Diagram



Sealed class

If you don't want other classes to inherit from a class, use the sealed keyword.

Polymorphism

Polymorphism means "many forms". Inheritance lets us inherit fields and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

- Function overloading
- Operator overloading

Dynamic Polymorphism

Dynamic polymorphism is implemented by

- Abstract classes
- Virtual functions.

The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Method overriding

C# provides an option to override the base class method, by adding the virtual keyword to the method inside the base class, and by using the override keyword for each derived class methods.

```
class Animal
{
  public virtual void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Dog : Animal
{
  public override void animalSound()
  {
    Console.WriteLine("The dog says: wee wee");
  }
}
```

Abstraction

Abstraction is an OOPs approach to construct the structure of the real-world objects. During the construction, only the general states and behaviours are taken, and more specific states and actions are left aside for the implementers.

Abstract Classes

- Cannot be instantiated
- Cannot be sealed
- May contain abstract members, but not mandatory
- A non-abstract class derived from an abstract class must provide implementation for all inherited abstract members
- Can have non-abstract methods
- An abstract class derived from an abstract class can have abstract methods. And it cannot be instantiated.

Abstract method

Abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the derived class

Why and When to Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Abstract Classes vs Interfaces

Abstract classes	Interfaces
Can have implementations for some of its members	Can't have implementation for any of its members
Can have fields	Cannot have fields
Can inherit from another abstract class or another interface	Can inherit from another interface only
A class cannot inherit from multiple classes	A class can inherit from multiple interfaces
Can have access modifiers	Cannot have access modifiers

Abstract Classes to Abstract Classes Inheritance

```
public abstract class Human
{
	public abstract void DoSomething();
}
public abstract class Employee : Human
{
	public abstract void DoSomethingElse();
}
```

Method Overriding

```
public abstract class Human
{
	public abstract void DoSomething();
}
public abstract class Rajeesh
{
	public override void DoSomething()
	{
		...
	}
}
```

Virtual vs Abstract Methods

- **Virtual methods** have an implementation and provide the derived classes with the option of overriding it. It can be of a regular class
- **Abstract methods** do not provide an implementation and force the derived classes to override the method.

Code

```
public abstract class E
{
    public abstract void AbstractMethod(int i);
    public virtual void VirtualMethod(int i)
    {
        // Default implementation which can be overridden by subclasses.
    }
}

public class D : E
{
    public override void AbstractMethod(int i)
    {
        // You HAVE to override this method
    }
    public override void VirtualMethod(int i)
    {
        // You are allowed to override this method.
    }
}
```

Interfaces

- An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies).
- By default, members of an interface are abstract and public. Interfaces can contain properties and methods, but not fields.
- To access the interface methods, the interface must be "implemented" by another class.

- You do not have to use the override keyword when implementing an interface
- Interface members are by default abstract and public

Why And When To Use Interfaces?

1. To achieve security - hide certain details and only show the important details of an object (interface).
2. C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can implement multiple interfaces.

Early and Late Binding (Static and Dynamic Binding)

Early Binding or Static Binding

It recognizes and checks the methods, or properties during compile time. In this binding, the compiler already knows about what kind of object it is and what are the methods or properties it holds, here the objects are static objects. The performance of early binding is fast and it is easy to code. It decreases the number of run-time errors.

Examples

```
ComboBox cboItems;
ListBox lstItems;
```

If we type the cboItem and place a dot followed by, it will automatically populate all the methods, events and properties of a combo box, because the compiler already knows it's a combobox.

Late Binding or Dynamic Binding

In late binding, the compiler does not know about what kind of object it is and what are the methods or properties it holds, here the objects are dynamic objects. The type of the object is decided on the bases of the data it holds on the right-hand side during run-time. Basically, late binding is achieved by using virtual methods. The performance of late binding is slower than early binding because it requires lookups at run-time.

Examples

```
dynamic obj = 4;
dynamic obj1 = 5.678;

Console.WriteLine(obj.GetType());
Console.WriteLine(obj1.GetType());
```

Constructors types

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Static Constructor
- Private Constructor

Default Constructor

A constructor without any parameters is called a default constructor

Parameterized Constructor

A constructor with at least one parameter is called a parameterized constructor

Copy Constructor

The constructor which creates an object by copying variables from another object is called a copy constructor

```
public employee(employee emp)
{
    name=emp.name;
    age=emp.age;
}
```

Static Constructor

A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Private Constructor

- It is the implementation of a singleton class pattern.
- Use private constructor when class have only static members.
- Using private constructor, prevents the creation of the instances of that class.
- If a class contains only private constructor without parameter, then it prevents the automatic generation of default constructor.
- If a class contains only private constructors and does not contain public constructor, then other classes are not allowed to create instances of that class except nested class.

Overloading and Overriding

Overloading

Overloading is static binding, whereas overriding is productive binding. Overloading is the same method with different arguments, and it may or may not return the equal value in the same class itself.

Overriding

Overriding is the same method names with the same arguments and return types identified with the class and its child class.
EoF