Chapter 3
Object Oriented Programming Using C# .NET

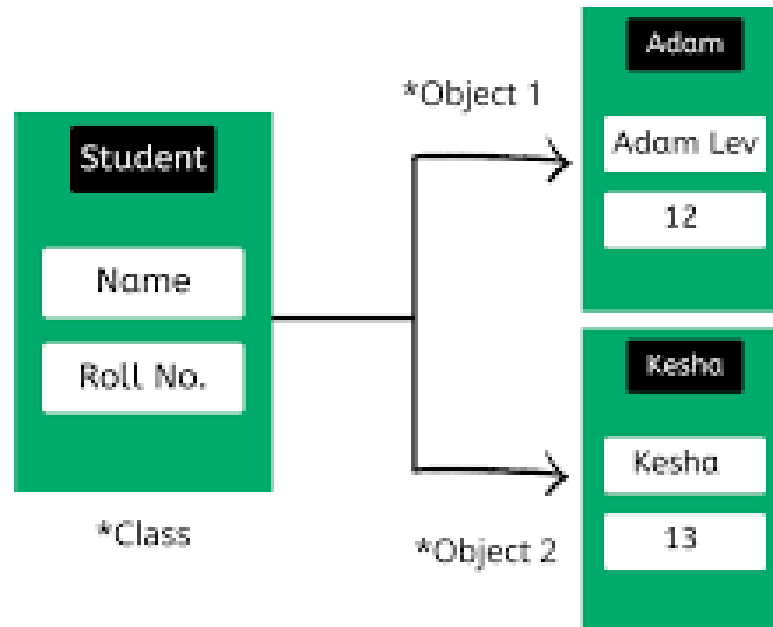# Object-Oriented Programming(OOP)

◆ **Programming languages are based on two fundamental concepts: data and ways to manipulate data. This approach had several drawbacks such as lack of re-use and lack of maintainability**

◆ **To overcome these difficulties, OOP was introduced, which focused on data rather than the ways to manipulate data**

◆ **The object-oriented approach defines objects as entities having a defined set of values and a defined set of operations that can be performed on these values**

◆ **Abstraction, encapsulation, polymorphism, and inheritance are the core principles of object-oriented programming**

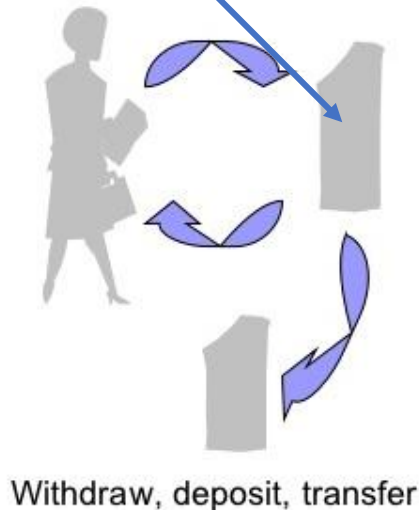# What is Object-Oriented Programming?

- A programming paradigm that focuses on objects
- What is an object?
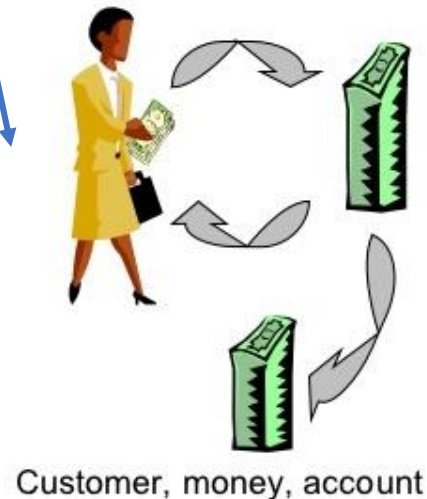 - Entity with unique identity that encapsulate state

Object-Oriented vs Procedural

- Procedural programming starts with features (functions) and how these features can be linked together

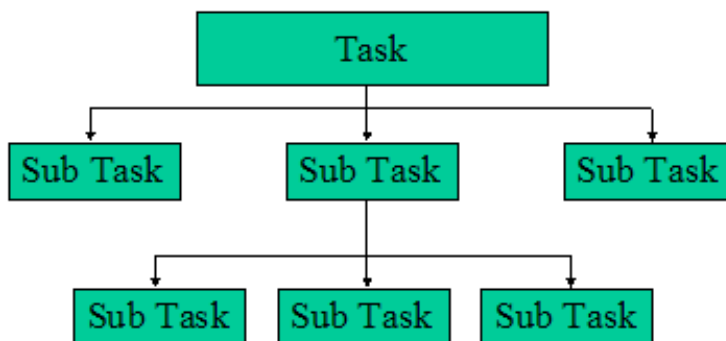- OOP starts with objects and how they communicate

■ Procedural

■ Object Oriented

Withdraw, deposit, transfer

Customer, money, account

Object-Oriented vs Procedural

- Procedural programming is top-down

❑ **OOP is bottom-up**
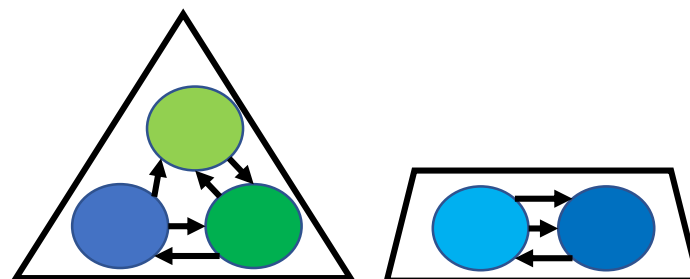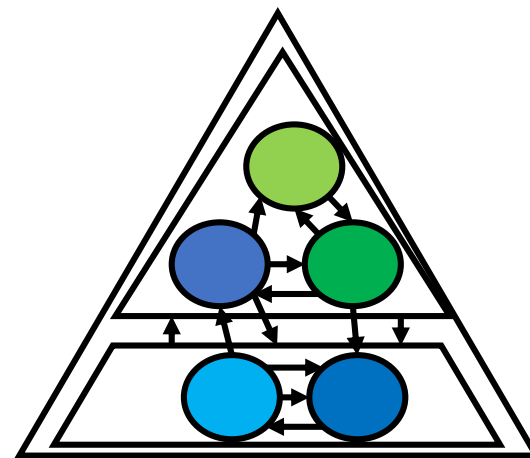
Object-Oriented vs Procedural

OOP Paradigm

Procedure-Oriented Program

| data1 |
| data2 |
| Function1 (data1) |
| Function2 (data1) |
| Function3 (data2) |
| Function4 (data2) |

Object = Data + Methods

Basic Concepts
- **Abstraction**
- Encapsulation
- Inheritance
- Polymorphism

Particular methods: Constructors

Class A
{
| data1 |
| Function1 () |
| Function2 () |
}

Modifiers

Class B
{
| data2 |
| Function3 () |
| Function4() |
}

Modifiers
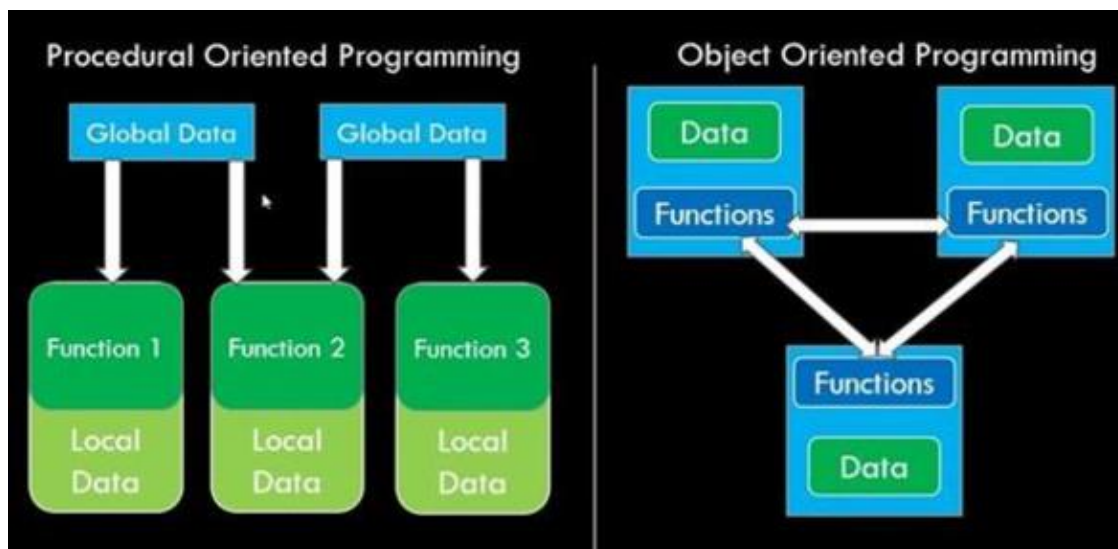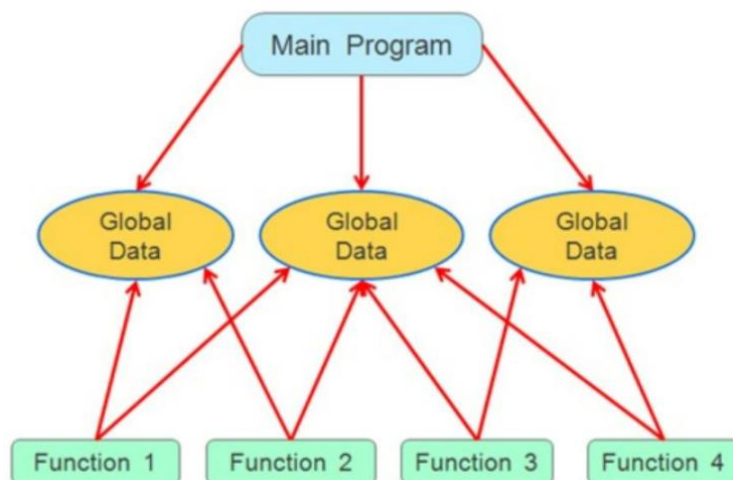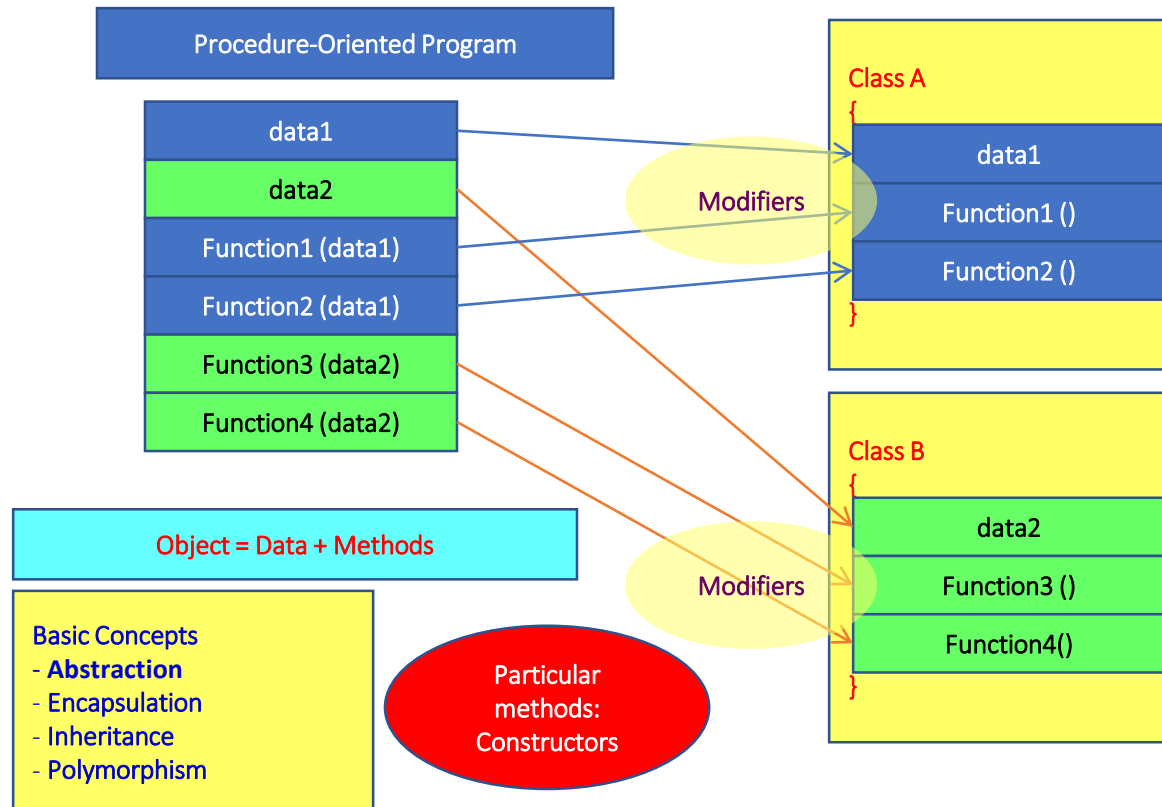
OOP Paradigm

Procedural oriented programming	Object Oriented Programming
It is known as POP	It is known as OOP
It deals with algorithms	It deals with data
Programs are divided into functions	Programs are divided into objects
Most of the functions share global data	Data Structure characterizes objects
Data move from function to function	Functions that operate on data are bind to form classes
Functions are responsible for transforming from one form to another	Data is hidden cannot be accessed by outside functions
It is top down approach	It is Bottom Up approach
It needs very less memory	It needs more memory that POP
Example:- C, Fortran	Example:- C++, JAVA,.NET
It do not have any access specifiers	It has access specifiers like private, public and protected.
It is less secure	It is more secure
It follows no overloading	It follows operator overloading and function overloading

Object-Oriented Programming

- **OOP is a powerful concept that solves many problems found in software development. OOP is not the holy grail of programming but it can help in writing code that is easy to read, easy to maintain, easy to update, and easy to expand.**

- **An object can inherit the properties of another object using the concept of inheritance. Hence, we can say that object-oriented programming is organized around data and the operations that are permitted on the data**

- **When we do object-oriented programming, we start with identifying the entities we need to operate on, how they relate to each other, and how they interact. This is a process called data modeling and the result of this is a set of classes that generalize the identified entities**

Classes and Objects

◆ **A class is a user-defined blueprint or prototype from which objects are created. Basically, a class combines the fields and methods(member function which defines actions) into a single unit**

Classes and Objects

◆ **An object is an instance of the class and represents a real-life entity. To initialize an object in C#, we use a new keyword followed by the name of the class that the object will be based on**

```csharp
public class Car{
    public string Make;
    public string Model;
    public void Driving(){
        Console.WriteLine("Driving.");
    }
    public void Accelerating(){
        Console.WriteLine("Accelerating.");
    }
    public void Braking(){
        Console.WriteLine("Braking.");
    }
}
```

```csharp
static void Main(string[] args)
{
    Car wwPolo = new Car();
    wwPolo.Make = "2050";
    wwPolo.Model = "Volkswagen Polo";
    wwPolo.Accelerating();
    wwPolo.Braking();
    wwPolo.Starting();
    Console.ReadLine();
}
```

Member Visibility

◆ **There are five access specifiers: private, public, protected, internal, and protected internal. By default, the members are private to the class**

- public: The type or member can be accessed by any other code in the same assembly or another assembly that references it

- private: The type or member can be accessed only by code in the same class or struct

- protected: The type or member can be accessed only by code in the same class, or in a class that is derived from that class

Member Visibility

- internal: The type or member can be accessed by any code in the same assembly, but not from another assembly

- protected internal: The type or member can be accessed by any code in the assembly in which it's declared, or from within a derived class in another assembly

- private protected: The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class

OOP-Encapsulation

◆ **Encapsulation is defined as binding data and code that manipulates it together in a single unit**

◆ **Data is privately bound within a class without direct access from the outside of the class**

◆ **All objects that need to read or modify the data of an object should do it through the public methods that a class provides**

◆ **This characteristic is called data hiding and makes code less error-prone by defining a limited number of entry points to an object's data**

OOP-Encapsulation

```csharp
class Customer{
    private int Id;
    //Full properties
    public int CustomerID {
        get {
            return Id;
        }
        set {
            Id = value;
        }
    }
    //Automatic properties
    public string CustomerName { get; set; } = "New customer";
    public void Print(){
        Console.WriteLine($"ID:{CustomerID}, Name:{CustomerName}");
    }
}
```

```csharp
class Program {
    static void Main(string[] args) {
        Customer obj = new Customer();
        obj.CustomerID = 1000;
        Console.WriteLine($"ID:{obj.CustomerID}, Name:{obj.CustomerName}");
        obj.CustomerID = 2000;
        obj.CustomerName = "Jack";
        obj.Print();
        Console.ReadLine();
    }
}
```

D:\Demo\FU\Basic.NET\Demo_Slot_04_05\DemoOO|

```
ID:1000, Name:New customer
ID:2000, Name:Jack
```

Read-only auto properties & Init-Only properties

◆ **When we write a property only with "get", it automatically becomes a Read Only property or we can use Init-Only properties**

```csharp
public class MyClass{
    //Init-Only Properties
    public int x { get; init; }
    //Read-Only Auto Properties
    public int y { get; }
    public MyClass(){
        x = 10;
        y = 20;
    }

    public MyClass(int a, int b){
        x = a;
        y = b;
    }

}
```

```csharp
class Program {
    static void Main(string[] args) {
        MyClass obj1 = new MyClass { x = 1 };
        Console.WriteLine($"x:{obj1.x},y:{obj1.y}");
        //obj.x = 10;  //error
        //obj.y = 20;  //error
        MyClass obj2 = new MyClass();
        Console.WriteLine($"x:{obj2.x},y:{obj2.y}");
        MyClass obj3 = new MyClass(30,50);
        Console.WriteLine($"x:{obj3.x},y:{obj3.y}");
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\
x:1,y:20
x:10,y:20
x:30,y:50
```

OOP-Inheritance

◆ **Inheritance is a mechanism through which a class can inherit the properties and functionalities of another class**

◆ **Other classes can inherit these functionalities and data of the parent class as well as extending or modifying them and adding additional functionalities and properties.**

◆ **There are three types of inheritance supported in C#:**

Single inheritance:

Multilevel inheritance

Hierarchical inheritance
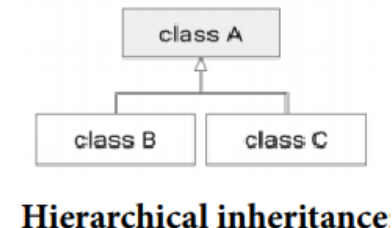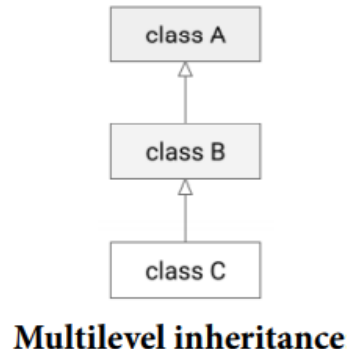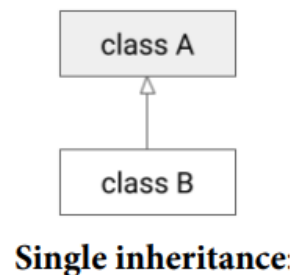
OOP-Inheritance

```csharp
class Employee{
    private int id;
    private string name;
    //Constructor
    public Employee(int id, string name){
        this.Id = id;
        this.Name = name;
    }
    //Properties
    public string Name {
        get { return name; }
        set { name = value; }
    }
    public int Id{
        get { return id; }
        set { id = value; }
    }

}
```

```csharp
class Manager:Employee
{
    private string email;
    public Manager(int id, string name, string email):base(id,name){
        this.Email = email;
    }
    public string Email {
        get { return email; }
        set { email = value; }
    }
    public override string ToString(){
        return $"Id:{Id},Name:{Name},Email:{Email}";
    }
}

class Program {
    static void Main(string[] args) {
        Manager jack = new Manager(1000,"Jack","Jack@gmail.com");
        Console.WriteLine(jack);
        Console.ReadLine();
    }
}
```

C:\ Select D:\Demo\FU\Basic.NET\Demo_Slot_04_05\DemoInheritance\bin

Id:1000,Name:Jack,Email:Jack@gmail.com

OOP-Polymorphism

◆ Ability allows many versions of a method based on overloading and overriding methods techniques

Classical polymorphism
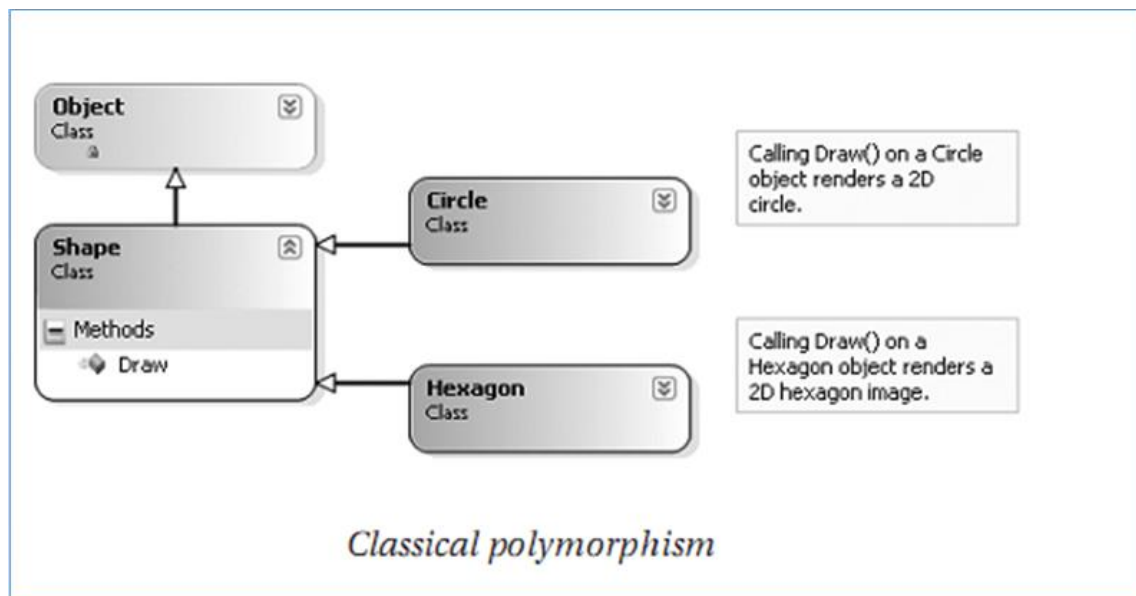
```csharp
public abstract class Shape{
    public int Height { get; set; }
    public int Width { get; set; }
    // Virtual method
    public virtual void Draw() {
        Console.WriteLine("Performing base class Drawing tasks");
    }
    public void Print() {
        Console.WriteLine("Performing base class Printing tasks");
    }
    // Abstract method
    public abstract void Display();
}
```

virtual : provide a default implementation. Can be overridden if necessary

abstract: sub-classes MUST override

```csharp
public class Rectangle : Shape {
    public override void Draw() {
        Console.WriteLine("Drawing a rectangle");
    }
    public override void Display() {
        Console.WriteLine("Display Rectangle");
    }
    public new void Print(){
        Console.WriteLine("Print Rectangle");
    }
}
```

```csharp
public class Circle : Shape {
    public override void Draw() {
        Console.WriteLine("Drawing a circle");
    }
    public override void Display(){
        Console.WriteLine("Display Circle");
    }
}
```

```csharp
class Program {
    static void Main(string[] args)
    {
        Shape p = new Circle();
        p.Display();
        p.Draw();
        p = new Rectangle();
        p.Display();
        p.Draw();
        p.Print();
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Demo_Slot_04_05\DemoPolymorphism\bir
Display Circle
Drawing a circle
Display Rectangle
Drawing a rectangle
Performing base class Printing tasks
```

Besides virtual and abstract, C# provide new keyword that applies to *methods*. What does that mean?

OOP-Interface

- **An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement**

- **An interface cannot be instantiated but can only be inherited by classes or other interfaces**

- **An interface may not declare instance data such as fields, auto-implemented properties, or property-like events. Interface names begin**

```
public interface ISomeInterface
{
    string SomeProperty { get; set; }
    string SomeMethod();
    void SomethingElse();
}
```

```
public class SomeClass : ISomeInterface{
    public string SomeProperty { get; set; }
    public string SomeMethod(){
        return "Hello World.";
    }
    public void SomethingElse(){
        Console.WriteLine(DateTime.Now);
    }
}
```
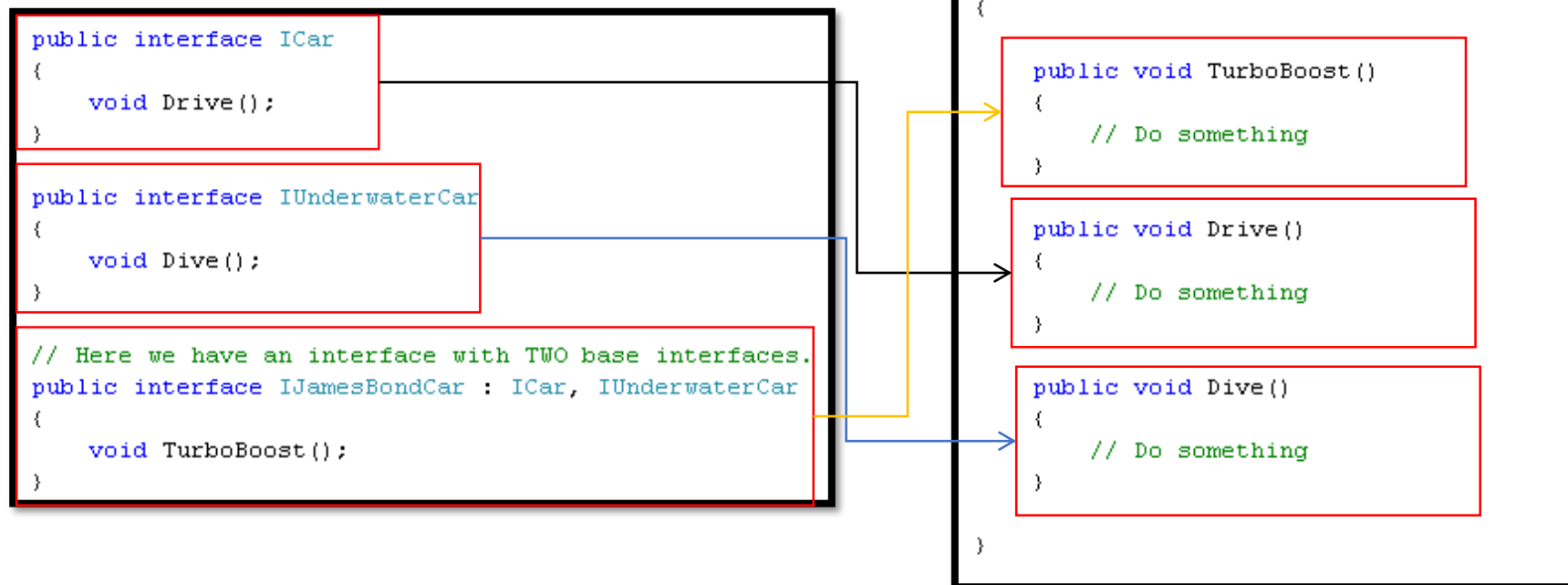
OOP-Interface

```csharp
public interface IFirst{
    void Print();
    void Display();
}
public interface ISecond {
    void Print();
}
public class MyClass : IFirst, ISecond{
    public void Display(){
        Console.WriteLine("Display method");
    }
    //Explicitly Implementing Interfaces
    void IFirst.Print(){
        Console.WriteLine("IFirst's Print method.");
    }
    //Explicitly Implementing Interfaces
    void ISecond.Print(){
        Console.WriteLine("ISecond's Print method.");
    }
}
```

```csharp
class Program{
    static void Main(string[] args){
        MyClass obj = new MyClass();
        obj.Display();
        IFirst first = obj;
        first.Print();
        ISecond second = obj;
        second.Print();
        Console.ReadLine();
    }
}
```

D:\Demo\FU\Basic.NET\Demo_Slot_04_05\

```
Display method
IFirst's Print method.
ISecond's Print method.
```

Interface Inheritance

```
public interface ICar
{
    void Drive();
}

public interface IUnderwaterCar
{
    void Dive();
}

// Here we have an interface with TWO base interfaces.
public interface IJamesBondCar : ICar, IUnderwaterCar
{
    void TurboBoost();
}
```

```
public class MyClass : IJamesBondCar
{
    public void TurboBoost()
    {
        // Do something
    }

    public void Drive()
    {
        // Do something
    }

    public void Dive()
    {
        // Do something
    }
}
```

Default Interface Methods

◆ **C# allows to add a method with their implementation to the interface without breaking the existing implementation of the interface, such type of methods is known as default interface methods(virtual extension methods)**

```csharp
public interface ISample{
    static void Print(){
        Console.WriteLine("Static method");
    }
    string GetString(string s)
    {
        return "Hello "+s;
    }
}
```

```csharp
interface ISample01{
    void MyMethod(){
        Console.WriteLine("ISample01.MyMethod");
    }
}
interface ISample02 : ISample01{
    //override the ISample01's MyMethod()
    void ISample01.MyMethod(){
        Console.WriteLine("ISample02.MyMethod");
    }
}
```

The is and as Operators

- The **is operator is used to check if the run-time type of an object** is compatible with the given type or not whereas as operator is used to perform conversion between compatible reference types or Nullable types

- The is operator **returns true** if the given object is of the same type whereas as operator returns the object when they are compatible with the given type

- The is operator returns false if the given object is not of the same type whereas as operator return null if the conversion is not possible

- The is operator is used for only reference, boxing, and unboxing conversions whereas as operator is used only for nullable, reference and boxing conversions

The is and as Operators

```csharp
interface ICalculate {
    double Area();
}
class Rectangle : ICalculate
{
    float length;
    float width;
    public Rectangle(float x, float y) {
        length = x;
        width = y;
    }
    public double Area() {
        return length * width;
    }
}
```

```csharp
class Program{
    static void Main(string[] args){
        Rectangle objRectangle = new Rectangle(10.2F, 20.3F);
        ICalculate calculate;
        if (objRectangle is ICalculate) {
            calculate = objRectangle as ICalculate;
            Console.WriteLine("Area : {0:F2}", calculate.Area());
        }
        else {
            Console.WriteLine("Interface method not implemented");
        }
    }
}
```

Microsoft Visual Studio Debug Console

Area : 207.06

Static Constructor

- **Prevent static field to be reset**

- **A given class (or structure) may define only a single static constructor**

- **A static constructor executes exactly one time, regardless of how many objects of the type are created**

- **A static constructor does not take an access modifier and cannot take any parameters**

- **The static constructor executes before any instance-level constructors**

```csharp
class MyClass{
    public static int x = 1;
    static MyClass(){
        x = 2;
        Console.WriteLine("Static constructor : x={0}",x);
    }
    public MyClass(){
        x++;
        Console.WriteLine("Object constructor : x={0}", x);
    }
}//end class
class Program {
    static void Main(string[] args)
    {
        MyClass m1 = new MyClass();
        MyClass.x = 4;
        MyClass m2 = new MyClass();
    }
}
```

Static Class

◆ **Classes that cannot be instantiated or inherited are known as classes and the static keyword is used before the class name that consists of static data members and static methods**

◆ **It is not possible to create an instance of a static class using the new keyword. The main features of static classes are as follows:**

▪ They can only contain static members

▪ They cannot be instantiated or inherited and cannot contain instance constructors. However, the developer can create static constructors to initialize the static members

Static Class

```
static class UtilityClass
{
    public static void PrintTime()
    { Console.WriteLine(DateTime.Now.ToShortTimeString()); }

    public static void PrintDate()
    { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}

class Program
{
    static void Main(string[] args)
    {
        UtilityClass.PrintDate();
    }
}
```

No instance created

Extension Method

◆ **Extension methods allow you to extend an existing type with new functionality without directly modifying those types**

◆ **Extension methods are static methods that have to be declared in a static class**

◆ **We can declare an extension method by specifying the first parameter with the this keyword**

◆ **The first parameter in this method identifies the type of objects in which the method can be called**

◆ **The object that we use to invoke the method is automatically passed as the first parameter**

Extension Method

```csharp
static class Utils
{
    public static int Add(int a, int b) { return a + b; }

    public static int Sub(this int a, int b) { return a - b; }
}
class Program
{
    static void Main(string[] args)
    {
        int x = 3, y = 2;
        int r = Utils.Add(x, y);
        Console.WriteLine("{0}+{1}={2}",x,y,r);
        r = x.Sub(y);
        Console.WriteLine($"{x}-{y}={r}");
        Console.ReadLine();
    }
}
```

Expression-bodied Members

- **Expression body definitions let you provide a member's implementation in a very concise, readable form**

- **We can use an expression body definition whenever the logic for any supported member, such as a method or property, consists of a single expression**

- **An expression body definition has the following general syntax:**

```
member => expression;
```

- **Members can be: Method, Read-only property, Property, Constructor, Finalizer and Indexer**

Expression-bodied Members

```csharp
public int Add(int x, int y)
{
    return x + y;
}
```

```csharp
public int Add(int x, int y) => x + y;
```

```csharp
public string Name
{
    get
    {
        return FirstName + " " + LastName;
    }
}
```

```csharp
public string Name => FirstName + " " + LastName;
```

```csharp
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Anonymous Type

- **Is basically a class with no name and is not explicitly defined in code**
- **Uses object initializers to initialize properties and fields. Since it has no name, we need to declare an implicitly typed variable to refer to it**
- **Anonymous types are class types that derive directly from object, and that cannot be cast to any type except object**

```csharp
static void Main(string[] args)
{
    var obj1 = new { id = 1000, name = "jack" };
    Console.WriteLine($"id:{obj1.id},name:{obj1.name}");

    dynamic obj2 = new { id = 2000, name = "scott", Email= "scott@gmail.com" };
    Console.WriteLine($"id:{obj2.id},name:{obj2.name},Email:{obj2.Email}");

    Console.ReadLine();
}
```

D:\Demo\FU\Basic.NET\Demo_Slot_04_05\DemoReadOnlyAutoProperties\b
id:1000,name:jack
id:2000,name:scott,Email:scott@gmail.com

Object Initialize

◆ **Object initializers let we assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements**

◆ **The object initializer syntax enables you to specify arguments for a constructor or omit the arguments**

```csharp
class Customer {
    public string Name { get; set; }
    public int Age { get; set; }
}
Customer c = new Customer();
c.Name = "Jack";
c.Age = 20;
var c = new Customer(){ Name = "Jack", Age = 20};
```

Can be combined with any constructor call

Readonly member and const keyword

- **In a field declaration, readonly indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class**

- **A readonly field can't be assigned after the constructor exits**

- **A const field can only be initialized at the declaration of the field(a compile-time constant)**

- **A readonly field can be assigned multiple times in the field declaration and in any constructor(a run-time constants)**

Readonly member and const keyword

```csharp
public class SamplePoint{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;
    // Initialize a const field
    public const int t = 100;
    public SamplePoint(){
        // Initialize a readonly instance field
        z = 24;
    }
    public SamplePoint(int p1, int p2, int p3,int p4=50){
        x = p1;
        y = p2;
        z = p3;
        //t = p4; //error
    }

}
```

```csharp
class Program{
    static void Main(string[] args) {
        SamplePoint p1 = new SamplePoint(11, 21, 32);
        Console.WriteLine($"t ={SamplePoint.t}");
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
        Console.ReadLine();
    }
}
```

D:\Demo\FU\Basic.NET\Demo_Slot_04_05\

```
t =100
p1: x=11, y=21, z=32
p2: x=0, y=25, z=24
```

Record type

◆ **Records type is a new reference type that we can create instead of classes or structs**

◆ **Records are distinct from classes in that record types use value-based equality**

◆ **We define a record by declaring a type with the record keyword**

```
public record Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

```
public record Person(string Name, int Age);
```

Record type

```csharp
public record Customer{
    public string Name { get; init; } = "New customer";
    public int Age { get; init; } = 20;
    public void Print(){
        Console.WriteLine($"Name:{Name}, Age:{Age}");
    }
}
class Program {
    static void Main(string[] args)
    {
        Customer customer01 = new Customer { Name = "Jack", Age = 25 };
        customer01.Print();
        Customer customer02 = customer01 with { Name = "John"};
        customer02.Print();
        Customer customer03 = new();
        customer03.Print();
        Console.ReadLine();
    }
}
```

D:\Demo\FU\Basic.NET\Demo_Slot_04_05\Demo

Name:Jack, Age:25
Name:John, Age:25
Name:New customer, Age:20

Using Declarations

◆ **With the using declaration, the objects are disposed automatically. Its scope is automatically defined from the object's declaration statement to the end of the current code block**

```csharp
static void Main(string[] args) {
    try
    {
        using var reader = new System.IO.StreamReader(@"C:\data.txt");
        var content = reader.ReadToEnd();
        Console.WriteLine($"File length: {content.Length}");
    }
    catch (System.IO.FileNotFoundException fe) {
        Console.WriteLine(fe.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadLine();
}
```