Object-oriented programming (OOP)
Lecture 1:Introduction to OOP

Dr. Abdullah Bokir

Syllabus Overview

- Introduction
- Classes in OOP
- Attributes and Methods
- Objects in OOP
- Constructors
- Characteristics of OOP

Syllabus Overview

- Access specifiers
- Exception Handling
- Events and Delegates
- Generics in C#

Introduction to OOP

- First let’s understand the meaning of the term object-oriented programming.

- **object**. According to the dictionary, an object is something that can be seen, felt, or touched; something that has physical existence in the real world

Introduction to OOP

- **Oriented**, which indicates a direction or something to aim for.

- **Programming** is just giving instructions to the computer to do a particular thing.

Introduction to OOP

- The phrase **object-oriented programming. OOP** means that we write our computer programs by keeping objects at the center of our thinking.

- OOP is neither a tool nor a programming language—it is just a concept.

Introduction to OOP

- C#, Java, C++, Python, … are examples of OOP languages.

- Other programming concepts in the programming world, such as **procedural programming** such as C language, **functional programming** such as F# language.

Classes in OOP

- A class is like a template or blueprint that tells us what properties and behaviors an instance of this class will have.

- In most circumstances, a class itself can't actually do anything—it is just used to create objects.

Classes in OOP

- **Example**: Human class. Here, when we say Human, we don't mean any particular person, but we are referring to a human being in general. A human that has two hands, two legs, and a mouth, and which can also walk, talk, eat, and think. These properties and their behaviors are applicable to most human beings.

Classes in OOP

- There are hundreds of properties that you can list for a human, for example:
  - Height
  - Weight
  - Age

Classes in OOP

- Similarly there are hundreds of particular behaviors that a person can perform, for example
  - Walk
  - Talk
  - Eat

The general form of a class in C#

- To create a class in C# language you need to follow the following syntax:

```
class class-name {
    // this is class body
}
```

- The class phrase is a **reserved keyword** in C#, and it is used to tell the compiler that we want to create a class.

The general form of a class in C#

- To create a class, place the **class** keyword and then the name of the class after a space.
- The name of the class can be anything that starts with a character or an underscore. We can also include numbers in the class name, but not the first character of a class name.
- After the chosen name of the class, you have to put an opening curly brace, which denotes the start of the class body.

The general form of a class in C#

- You can add content in the class, such as properties and methods
- then finish the class with a closing curly brace as follows.

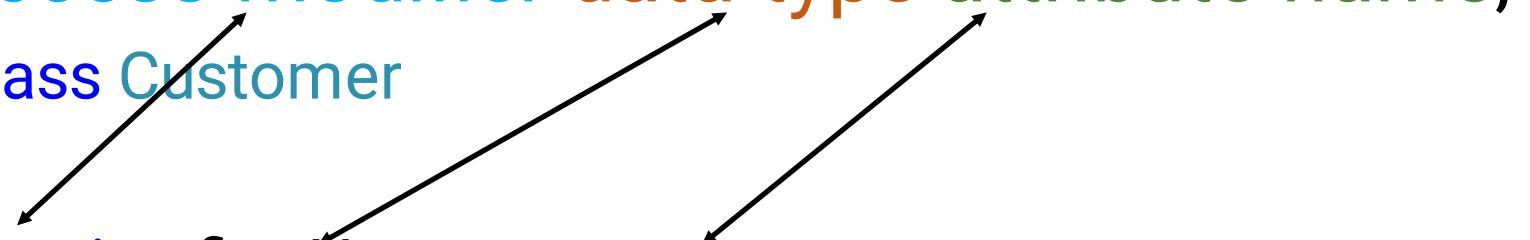
```
class class-name {
  // property 1
  // property 2
  // ...

  // method 1
  // method 2
  // ...
}
```

Attributes in a class

- Attributes are like variables and can be defined using the following syntax

access-modifier data-type attribute-name;



```
public class Customer
{
    public string firstName;
    public string lastName;
    public string phoneNumber;
    public string emailAddress;
}
```

Methods in a class

- A method is a piece of code that is written in the code file and can be reused.

- A method can hold many lines of code, which will be executed when it is called. Let's take a look at the general form of a method:

```
access-modifier return-type method-name(parameter-list) {
    // method body
}
```

Methods in a class

- We can see that the first thing in the method declaration is an **access-modifier**. This will set the access permission of the method.
- Then, we have the **return-type** of the method, which will hold the type that the method will return, such as string, int, double, or another type.
- After that, we have the **method-name** and then brackets, **()**, which indicate that it is a method.

Methods in a class

- In the brackets, we have the **parameter-list**. This can either be empty or can contain one or more parameters.
- Finally, we have curly brackets, **{}**, which hold the **method body**. The code that the method will execute goes inside here.

Creating a method

```
public string GetFullName()
{
    return firstName + " " + lastName;
}
```

Writing a simple class

- Let's imagine that we are developing some software for a bank. Our application should keep track of the bank's customers and their bank accounts, and perform some basic actions on those bank accounts.

- As we are going to design our application using C#, we have to think of our application in an object- oriented way.

Writing a simple class

- Some objects that we will need for this application could be a customer object, a bank account object, and other objects.

- So, to make blueprints of these objects, we have to create a Customer class and a BankAccount class

Writing a simple class

```
class Customer
{
    public string firstName;
    public string lastName;
    public string phoneNumber;
    public string emailAddress;

    public string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}
```

Writing a simple class

- We started with the class keyword and then the name of the class, which is Customer. After that, we added the class body inside curly braces, {}.
- The variables that the class has are firstName, lastName, phoneNumber, and emailAddress.
- The class also has a method called GetFullName(), which uses the firstName and the lastName fields to prepare the full name and return it.

Writing a simple class

```
class BankAccount
{
   public string bankAccountNumber;
   public string bankAccountOwnerName;
   public double amount;
   public DateTime openningDate;

   public string Credit()
   {
      return "amount credited";
   }

   public string Debit()
   {
      return "amount debited";
   }
}
```

Writing a simple class

- Here, the name of the class is BankAccount
- fields of the class are bankAccountNumber, bankAccountOwnerName, amount, and openningDate
- Two methods, Credit and Debit are defined in the class.

Objects in OOP

- An object is an instance of a class. In other words, an object is an implementation of a class.

- For example, in our banking application, we have a Customer class, but that doesn't mean that we actually have a customer in our application

Objects in OOP

- To create a customer, we have to create an object of the Customer class.

- Let's say that we have a customer called Mr. Jack Jones. For this customer, we have to create an object of the Customer class, where the name of the person is Jack Jones.

How to create objects

- In C#, to create an object of a class, you have to use the **new** keyword. As shown in the following syntax

```
class-name object-name  = new class-name();
```

- Example

```
Customer jackJones = new Customer();
```

Assigning values to the variables of the object

```
public class Program
{
    static void Main(string[] args)
    {
        Customer customer1 = new Customer();
        customer1.firstName = "jack";
        customer1.lastName = "Jones";
        customer1.phoneNumber = "1234567890";
        customer1.emailAddress = "jackJones@gmail.com";
        Console.WriteLine("customer's full name is: "+ customer1.GetFullName());
    }
}
```

Constructor of a class

- In every class, there is a special type of method, called a **constructor**.
- You can create a constructor in a class and program it [This is called **Explicit Constructor**]. If you don't create one yourself, the compiler will create a very simple constructor and use that instead [This is called **Implicit Constructor**].

Constructor of a class

- The name of the constructor should be the same as the class name.

- A constructor doesn't have a return type. This is because a constructor can't return anything; it's for initialization, not for any other type of action

Constructor of a class

- A constructor is a method that gets triggered when an object of a class is created.

- A constructor is mainly used to set the prerequisites of the class.

Constructor of a class

- For example, if you are creating an object of the Human class, that human object must have a date of birth. You can set this requirement in the constructor.

- You can also configure the constructor to set the date of birth as today if no date of birth is given. This depends on the needs of your application.

Constructor of a class

- Let's take a look at the general form of a constructor, as shown in the following code:

```
access-modifier class-name(parameter-list) {
    // constructor body
}
```

Example of a Constructor (parameterless Constructor)

```
class BankAccount
{
    public string owner;
    public BankAccount()
    {
        owner = "Some person";
    }
}
```

Example of a Constructor (parameterized Constructor)

- We can also take the name of the owner of the bank

account as a parameter in the constructor and use it to assign the variable, as shown in the following code:

```
class BankAccount
{
    public string owner;
    public BankAccount(string theOwner)
    {
        owner = theOwner;
    }
```

Example of a Constructor (parameterized Constructor)

- If you put parameters in the constructor, then, when initializing the object, the parameters need to be passed, as shown in the following code:

```
BankAccount account = new BankAccount("Some Person");
```

Multiple Constructors in a class

- Another interesting thing is that you can have multiple constructors in a class.
- You might have one constructor that takes one argument and another that doesn't take any arguments.
- Depending on the way in which you are initializing the object, the respective constructor will be called. Let's look at the following example:

Multiple Constructors in a class

```
class BankAccount
{
    public string owner;
    public BankAccount()
    {
        owner = "Some person";
    }
    public BankAccount(string theOwner)
    {
        owner = theOwner;
    }
}
```

Multiple Constructors in a class

- In the preceding example, we can see that we have two constructors for the BankAccount class.
- If you pass a parameter when you create a BankAccount object, it will call the second constructor, which will set the value and create the object.

Multiple Constructors in a class

- If you don't pass a parameter while creating the object, the first constructor will be called. If you don't have the appropriate constructors, then the method of object creation will not work.

Implicit Constructors in a class

- If you don't create a constructor, then the compiler creates an empty constructor for that class, as follows:

```
class BankAccount
{
    public string owner;
    public BankAccount()
    {
    }
}
```

Characteristics of OOP

- OOP is one of the most important programming methodologies nowadays. The whole concept depends on four main ideas, which are known as the **pillars of OOP**. These four pillars are as follows:
  - Inheritance
  - Encapsulation
  - Polymorphism
  - Abstraction

Inheritance

- The word inheritance means receiving or deriving something from something else. In real life, we might talk about a child inheriting a house from his or her parents. In that case, the child has the same power over the house that his parents had

Inheritance

- This concept of inheritance is one of the pillars of OOP.
- In programming, when one class is derived from another class, this is called inheritance. This means that the derived class will have the same properties as the parent class.

Inheritance

- In programming terminology, the class from which another class is derived is called the **parent class**, while the classes that inherit from these are called **child classes**.

Inheritance Example

```
public class Fruit
{
    public string Name { get; set; }
    public string Color { get; set; }
}
public class Apple : Fruit
{
    public int NumberOfSeeds { get; set; }
}
```

Inheritance Example

- In the preceding example, we used inheritance. We have a parent class, called Fruit. This class holds the common properties that every fruit has: a **Name** and a **Color**. We can use this Fruit class for all fruits.

- If we create a new class, called Apple, this class can inherit the Fruit class because we know that an apple is a fruit.

Inheritance Example

- The properties of the Fruit class are also properties of the Apple class.

- If the Apple inherits the Fruit class, we don't need to write the same properties for the Apple class because it inherits these from the Fruit class.

Encapsulation

- Encapsulation means hiding or covering. In C#, encapsulation is achieved by access modifiers. The access modifiers that are available in C# are the following:
  - Public
  - Private
  - Protected
  - Internal
  - Internal protected

Encapsulation

- Encapsulation is when you want to control other classes' access to a certain class. Let's say you have a BankAccount class. For security reasons, it isn't a good idea to make that class accessible to all classes. It's better to make it Private or use another kind of access specifier.

Encapsulation

- You can also limit access to the properties and variables of a class. For example, you might need to keep the BankAccount class public for some reason, but make the AccountBalance property private so that no other class can access this property except the BankAccount class

- Like variables and properties, you can also use access specifiers for methods. You can write private methods that are not needed by other classes, or that you don't want to expose to other classes

Abstraction

- If something is abstract, it means that it doesn't have an instance in reality but does exist as an idea or concept.
- In programming, we use this technique to organize our thoughts. This is one of the pillars of OOP.
- In C#, we have **abstract classes**, which implement the concept of abstraction. Abstract classes are classes that don't have any instances
- classes that implement the abstract class will implement the properties and methods of that abstract class.

Abstraction Example

```
public abstract class Vehicle
{
    public abstract int GetNumberOfTyres();
}
public class Bicycle : Vehicle
{
    public string Company { get; set; }
    public string Model { get; set; }
    public int NumberOfTyres { get; set; }
    public override int GetNumberOfTyres()
    {
        return NumberOfTyres;
    }
}
```

Abstraction Example

- In the preceding example, we have an abstract class called Vehicle.
- It has one abstract method, called GetNumberOfTyres().
- As it is an abstract method, this has to be **overridden** by the classes that implement the abstract class.

Abstraction Example

- Our Bicycle and Car classes implement the Vehicle abstract class, so they also override the abstract method GetNumberOfTyres().

- If you take a look at the implementation of these methods in the two classes, you will see that the implementation is different, which is due to abstraction.

Polymorphism

- The word polymorph means many forms.

- To understand the concept of polymorphism properly, let's work with an example. Let's think about a person, such as Bill Gates. We all know that Bill Gates is a great software developer, businessman, and philanthropist. He is one individual, but he has different roles and performs different tasks. This is polymorphism.

Polymorphism

- When Bill Gates was developing software, he was playing the role of a software developer. He was thinking about the code he was writing.
- Later, when he became the CEO of Microsoft, he started managing people and thinking about growing the business.
- He's the same person, but with different roles and different responsibilities.

Polymorphism

- In C#, there are two kind of polymorphism: **static polymorphism** and **dynamic polymorphism**.

- **Static polymorphism** is a kind of polymorphism where the role of a method is determined at compilation time.
- in **dynamic polymorphism**, the role of a method is determined at runtime.

Polymorphism

- Examples of static polymorphism include method overloading and operator overloading.
- Let's take a look at an example of method overloading:

static polymorphism (method overloading)

```
public class Calculator
{
    public int AddNumbers(int firstNum, int secondNum)
    {
        return firstNum + secondNum;
    }
    public double AddNumbers(double firstNum, double secondNum)
    {
        return firstNum + secondNum;
    }
}
```

static polymorphism (method overloading)

- we can see that we have two methods with the same name, **AddNumbers**.

- Normally, we can't have two methods that have the same name; however, as the parameters of those methods are different, methods are allowed to have the same name by t he compiler.

static polymorphism (method overloading)

- Writing a method with the same name as another method, but with different parameters, is called **method overloading**. This is a kind of **polymorphism**.

Dynamic polymorphism

- Dynamic polymorphism refers to the use of the abstract class.
- When you write an abstract class, no instance can be created from that abstract class.
- When any other class uses or implements that abstract class, the class also has to implement the abstract methods of that abstract class.

Dynamic polymorphism

- As different classes can implement the abstract class and can have different implementations of abstract methods, polymorphic behavior is achieved.

- In this case, we have methods with the same name but different implementations.

static polymorphism (operator overloading)

- Like method overloading, **operator overloading** is also a **static polymorphism**. Let's look at an example of operator overloading to demonstrate this:

static polymorphism (operator overloading)

```
public class MyCalc
{
    public int a;
    public int b;
    public MyCalc(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
    public static MyCalc operator +(MyCalc a, MyCalc b)
    {
        return new MyCalc(a.a * 3, b.b * 3);
    }
}
```

static polymorphism (operator overloading)

- In the preceding example, we can see that the plus sign (+) is overloaded with another kind of calculation. So if you sum up two MyCalc objects, you will get an overloaded result instead of the normal sum, and this overloading happens at compile time, so it is **static polymorphism**.