**PART 1 – Generally Topic**

**Slide - 3**

Today's lesson progress will be in 4 parts. Basically, we will look at what OOP is and what are its principles with sample code blocks. In the second part, we will talk about Generics, Reflections, Delegate & Events, Collectins, Enum and Extension structures. In part 3, we will try to reinforce what we have learned through a simple console game project. In part 4 we will summarize and finish the course.

Yes, let's start the first part. Our topic is OOP, normally this topic is given as a one semester course in universities, so we will not go too deep.

**PART 2 – Intr. OOP**

**Slide – 4**

Basically, OOP is a concept that aims to remove the chaotic environment of programming. So it aims to reduce the complexity by putting certain structures and it brings a certain standard. It is not actually a new concept. It started in the 1960s, but it is still in use today. It is common in current programming languages such as Java, C#, Python, and C++.

**Slide – 5 Benefit**

So what are the advantages?

Actually, as I said in the previous slide, reducing chaos. How does it do that? By enabling us to make modular design, it makes it easy to maintain and modify those modules later on. In addition, since each module is made independently of each other, changes can be made without affecting other places. I think one of the most important things is that it prevents code duplication and helps speed up development by enabling the creation of reusable functions.

For example, let's say you are making a 2D space shooter game in Unity. You could use OOP to create a class called `Weapon` that defines the common properties and behaviors of all weapons in the game, such as damage, fire rate, ammo, sound effects, etc. Then you could create subclasses of `Weapon` that inherit from it and add specific features for each weapon type, such as `Laser`, `Missile`, `Plasma`, etc. Each subclass would have its own methods for firing, reloading, displaying UI, etc.

This way, we don't repeat the basic features for each weapon. We eliminate code overload and save the time it would take to develop this kind of code.

You also make your code more modular and maintainable, as you can easily change or add features to one class without affecting the others.

**Slide – 6 -> Image OOP**

In this picture we see 6(six) different terms of OOP.

First, let's shortly talk about classes and objects.

In OOP, classes are like a blueprint that defines variables, properties and methods common to all objects of a certain type. An object is an instance of a class.

**Slide – 7 -> Class and Object Relationship**

Take for example a car class. This class has some properties like model, color and speed and these are variables that can take different values for each object of the class.

The class has a constructor, which is a special method used to create objects of the class. This constructor forces us to enter our model, color and speed variables when we want to create an object of this class.

We also have 3 methods. Start, Accelerate and Stop. The purpose of these methods is to perform the tasks given to them. The structure generated from this class is an object that can use these properties.

As you can see, tesla is a object of Car class and it has some properties. For example model is a Tesla Model S, color is a red and max speed is 200.

So, Car is a class and tesla is a instance of car class.

**Slide – 8 -> Class and Object Relationship**

Let's shortly talk about Inheritance, Polymorphism, Abstraction and Encapsualtion. We will explore each of them in more detail in the next slides.

Encapsulation is the process of binding data members and member functions into a single unit.

Inheritance is a mechanism that allows you to create a new class from an existing class.

Abstraction is the process of hiding implementation details while showing only functionality to the user.

Polymorphism allows you to write code that can be used with multiple types of objects.

For example, let's say you have a class called "Vehicle." You could create subclasses of "Vehicle" called "Car," "Truck," and "Motorcycle." Each subclass would inherit the properties of "Vehicle," but could also have its own unique properties.

**Slide – 9 -> Encapsulation Object-Oriented Programming**

Encapsulation is one of the basic concepts of OOP. In general, it provides limits on how much of the classes, properties, variables and methods can be viewed and modified.

By isolating the public interface from the implementation details, encapsulation helps to keep the code clean and manageable. This allows the developer to interface with objects at a higher degree of abstraction, resulting in easier-to-read, create, and maintain code.

There are generally three access modifiers: public, private and protected.

Public ones can be seen and changed by anyone, so they are the most insecure class type. When writing a program, it is not recommended that methods that will change the internal structure of the program be Public. Public modifiers are used for data that can be added or modified by an external user. Protected modifiers can be viewed or accessed within the same class. They can also be viewed or accessed by superclasses, derived classes and classes in the same package. Private ones can only be viewed or accessed by the class they are in.

**Slide – 10 -> Encapsulation in C# Game Scenario**

Let's look at encapsulation with a small example. Here we see the class of a character in a game and its structure.

There are some public and private structures here. For example, as you can see, the health and name fields are private. This means that these variables cannot be accessed from other classes. So how can we change them or use them if we need the name of the character in other places.

As you can see here there are 2 name fields but 1 of them is variable and 1 of them is property. Here we cannot access variable directly from outside so we use property to access it. You may say why do we need such a thing. But when you look at the set side of the name property, we see that it is also private. And inside we see that there is a method called CheckName.

Now think about it like this here, when creating a character in the game, we already have to give a name in our constructor, that is, when creating a character, we need to create new Character("bobmarley"). We also write our own controls in the checkName field. It must be at least 6 characters or not contain numbers. And when we look at this class, we see that the name must be given mandatory when the user is created and must meet certain conditions and cannot change the name later. We understand this clearly.

Likewise, when the character is created in health, we have 100 health by default. Then the health can only change with the TakeDamage function, so if our character takes damage, our health decreases. Or we can use the isAlive function to check if our character is dead or not.

This way, we can control how the health and name field is modified and ensure that other objects can only interact with the player object in a safe and controlled way.

This is the encapsulation part in general.

**Slide – 11 -> Inheritance in Object-Oriented Programming**

Inheritance is a mechanism that allows us to derive one class from another by establishing a parent-child connection and utilizing shared methods and attributes on these classes. It is one of the foundational ideas of oop. It enables the creation of new classes on top of existing ones.

Inheritance is significant because it allows code to be reused, which reduces duplication and improves efficiency. It also enables developers to design class hierarchies that simulate real-world object connections, making code easier to understand and maintain.

In Java, inheritance is accomplished by using the "extends" keyword, whereas in C# and Python, it is accomplished by using the ":" (Colon) symbol. The superclass is extended by the subclass, which inherits all of its properties and methods. To offer unique behavior, the subclass can override the inherited methods or introduce new ones.

There are 5 types of Inheritance.

- **Single Inheritance:** The subclass carries all the properties of a single superclass.

- **Multiple Inheritance:** A subclass inherits all the properties of more than one superclass.

- **Multilevel Inheritance:** After a subclass of a class is created, a subclass of this subclass is also created.

- **Hierarchical Inheritance:** It is when a superclass acts as a base class for more than one subclass.

- **Hybrid Inheritance:** Inheritance type that contains 2 or more of the other Inheritance types.

**Slide – 12 -> Example of Inheritance in C#**

The character class in this example is the same class as in the previous Encapsulation section. But here there are 2 sub classes created based on this class.

As you can see, by using the ":"(cover) sign we are saying that these classes inherit the Character class. So actually Player and NonPlayer classes contain the properties and methods of the base class Character. But they can also contain extra properties of their own.

As we can see in the code blocks here, the Player class has the Level and PlayerType property, while the NonPlayer class has the NPCType property.

Also, as I said, player and nonplayer have all the properties and methods that we see in the character class.

If you have no questions, I will skip to the next part, Abstraction.

**Slide – 13 -> Abstraction in Object-Oriented Programming**

Abstraction is one of the basic concepts of OOP. If it is wanted to create a superclass that has the common properties and functions of its subclasses but does not yet have an object, an abstract superclass is created.

The methods of the abstract class can be defined as templates to be overwritten by its subclasses or can be created as abstract methods.

This is an important point, a class with an abstract method automatically becomes abstract itself and no objects are created from abstract classes.

It allows developers to work with objects at a higher level of abstraction, which makes code easier to read, write, and maintain.

Abstraction in OOP can be implemented through \*\*interfaces\*\* and \*\*abstract classes\*\*. They are used to create a base implementation or contract for the actual implementation classes.

**Slide – 14 -> Example of Abstraction in C#**

In a game scenario, we can use abstraction to simplify the interface between the player and the game world. For example, we can define an ICharacter interface that exposes only the properties and methods that the player needs to have, and then apply this interface to different types of game objects, such as Player and Non-Player.

As you can see here, there is 1 interface, 1 abstract class and 2 driven (child) classes.

Abstraction can be applied both with interfaces and abstract classes and methods.

Since interfaces and abstract classes are abstract, you cannot create objects from these structures.

As you can see here, I used 2 structures at the same time. Interface shows the methods that our base class should have. So when we implement ICharacter class in Character class, we have to add the methods in ICharacter and fill its body.

Our base class, Character, allows other classes that implement it to use the TakeDamage and Heal methods directly from the base class. This prevents characters of different types from typing the same methods over and over again. If we want the body of the base methods of some characters of different types to be different, we can add the virtual prefix to that method in the base class and then override it in the other subclass.

If the method will not have a body in the base class but needs to be used in subclasses, we use abstract methods. As you can see, here Character has a method called Voice but it doesn't have a body in the base class. So we fill its body from the Player and NonPlayer classes.

Later, when we create an object of the Player class to the Charater variable anywhere, we can use both the basic methods (Take Damage and Heal) and the unique Voice Method.

**Slide – 15 -> Polymorphism in Object-Oriented Programming**

The subclasses derived from the superclasses do not always have to show the policies or behaviors of the superclasses. Based on this, I can say that subclasses behaving differently from the superclasses are called polymorphisms.

Polymorphism is important because it allows us to perform a single action in different ways depending on the type of object that we are working with. Polymorphism enables us to write generic and reusable code that can handle different types of objects without knowing their specific details at compile time. Polymorphism also supports the principle of inheritance, which allows us to create subclasses that inherit the common behavior and state from their parent classes, but also provide their own specific implementation or extension of some methods.

Polymorphism is achieved by using inheritance and interfaces, which allow subclasses and implementing classes to provide their own versions of methods defined in the parent class or interface.

**Slide – 16 -> Polymorphism in C# Game Scenario**

Polymorphism is the ability of an object to take many forms. In C# game development, we can use polymorphism to create different types of game objects that share a common interface but have different behaviors. For example, we can define a base Character class with a CheckName method and then create different Character subclasses, such as Player and NonPlayer, which override the CheckName method that checks for different character names for each character type.

Here again we see our Character base class and the Player and NonPlayer classes based on it. Here we have CheckName method with virtual keyword. But this method shows multiple characters according to the type during runtime. If it works as a player, the method specific to the Player class will work, if it works as a nonplayer, the method specific to the NonPlayer class will work. This is called dynamic polymorphism. The name of the operation is method overriding.

Here, as you can see in the Attack class, we have 2 methods with the same name, but one of them takes parameters and the other does not take parameters. As we can see from here, if a method has the possibility of working with more than one different possibility, we can create methods with the same name but with different signatures.

This is called static polymorphism and the process is called method overloading.

**FINISH OOP SLIDES**