ADVANCED TOPICS C#

Introduction Myself

1

Hi there!

I'm Seyyid Yiit. I graduated CE at Erciyes University.

(2015-2020)

2

Part-Time

I worked as a back-end developer (.net) at Inosis Software.

(Sep 2019 - June 2020)

3

Full-Time

I worked as a back-end developer (.net) focused on e-commerce at Mazaka Software.

(Sep 2020 - Oct 2022)

4

Currently

I'm working as a Game DTSE at Huawei. I'm working on C#/Unity development and also developing the HMS Unity plugin and SDK.

Agenda

1 **Part 1**

Object-Oriented Programming(OOP)

2 **Part 2**

- Generics
- Reflection
- Extension
- Enum
- Collections
- Ref, Out and Return Multiparameter

3 **Part 3**

Basic Console Project

4 **Part 4**

Conclusion

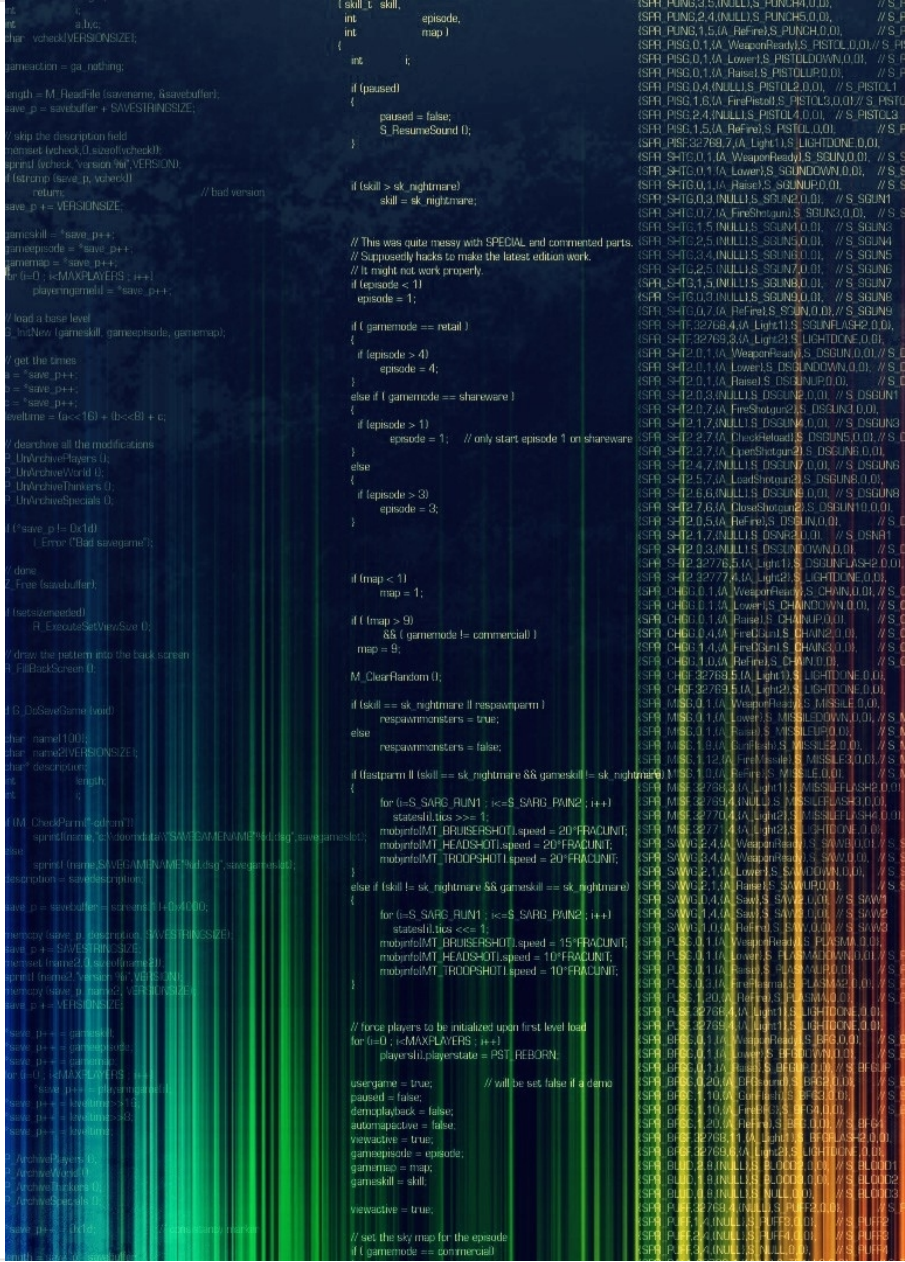
HUAWEI

Introduction to Object-oriented Programming (OOP)

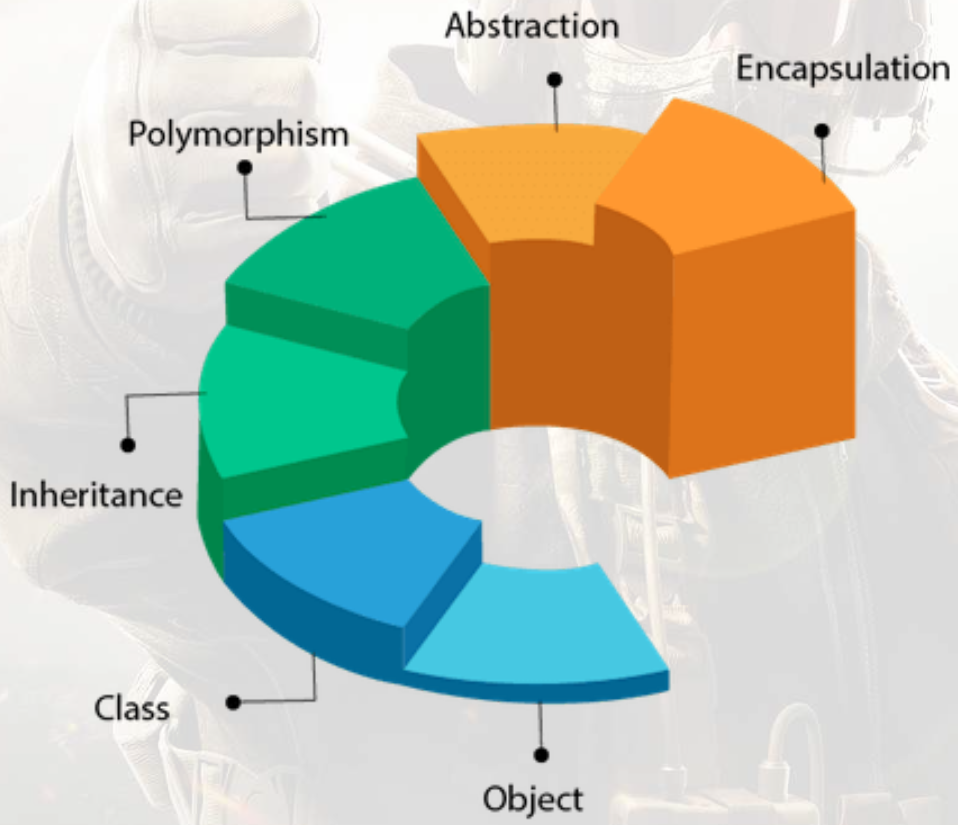
Object-oriented programming is a programming paradigm that organizes data and behavior into reusable and modular units called objects. OOP allows for the creation of complex applications by breaking them down into manageable pieces. It is widely used in modern programming languages like Java, C#, Python, and C++.

The Benefits of Object-Oriented Programming

Object-Oriented Programming (OOP) is a powerful programming paradigm that provides several benefits over traditional procedural programming. Benefits of OOP include encapsulation, inheritance, abstraction, and polymorphism, which can lead to easier-to-understand, maintainable, and scalable code. OOP can also make it easier to reuse code and collaborate with other developers.



OOPs (Object-Oriented Programming System)



Class and Object Relationship

Tesla.cs U

OOP_With_Game > ExternalGameProject_Example > Tesla.cs > ProgramTest

```
//Class Car defines what a car is class
2 references
public class Car {
    // Properties
    4 references
    string model;
    1 reference
    string color;
    3 references
    int speed;

    // Constructor
    1 reference
    public Car(string model, string color, int speed) {
        this.model = model;
        this.color = color;
        this.speed = speed;
    }

    // Methods
    1 reference
    public void start() {
        Console.WriteLine(model + " is starting.");
    }

    1 reference
    public void accelerate() {
        speed += 10;
        Console.WriteLine(model + " is accelerating.");
    }

    1 reference
    public void stop() {
        speed = 0;
        Console.WriteLine(model + " is stopping.");
    }
}

0 references
public class ProgramTest {
    0 references
    public static void Main(string[] args) {
        // Object Tesla is an instance of Car
        Car tesla = new Car("Tesla Model S", "red", 200);
        // Calling methods of Tesla
        tesla.start();
        tesla.accelerate();
        tesla.stop();
    }
}
```

The Principles of Object-Oriented Programming

Encapsulation	The bundling of data and methods within an object
Inheritance	Allows objects to inherit properties and behaviors from parent objects
Abstraction	Hiding implementation details to focus on essential features
Polymorphism	Allows objects to take on different forms and behave differently depending on their context

By following these principles, developers can write code that is more modular, reusable, and maintainable.

Encapsulationin Object-Oriented Programming

- What is Encapsulation?

  Encapsulation is the practice of hiding the internal details of an object from the rest of the program and exposing a public interface for interacting with the object. This allows the internal state of the object to be protected from outside interference and makes it easier to maintain and modify the code in the future.

- Why is encapsulation important?

  Encapsulation helps to keep the code organized and maintainable by separating the public interface from the implementation details. This allows developers to work with objects at a higher level of abstraction, which makes code easier to read, write, and maintain.

- How is encapsulation achieved?

  Encapsulation is achieved through the use of access modifiers such as public, private, and protected. These modifiers control the visibility of the class members and allow developers to control which parts of the code can access the internal state of the object.

edureka!

Encapsulation in C# Game Scenario

Encapsulation is an important concept in game development, where you want to hide the internal details of game objects and only expose a public interface for other objects to interact with. For example, let's say we have a character class that has a some property that we want to hide from other objects.

Code Example

▼ Character.cs

```
public class Character
{
   private string name;
   public string Name
   {
      get { return this.name; }
      private set
      {
         this.name = this.CheckName(value);
      }
   }
   private int health = 100;
   public int Health { get {return health} private set {value = this.health;}}

   public int Attack { get; set;}
   public int Defense { get; set;}

   public Character(string name)
   {
      Name = name;
   }
   public void TakeDamage(int damage)
   {
      Health -= Health ;
      //Health = Health - damage
   }
   public bool IsAlive()
   {
      return health > 0;
   }

}
```

In this example, we've made the health and name field private to hide it from other objects. We've then exposed public methods for taking damage, checking if the player is alive, and public properties for Attack, and Defense. This way, we can control how the health field is modified and ensure that other objects can only interact with the player object in a safe and controlled way.

Inheritance in Object-Oriented Programming

- What is Inheritance?

  > Inheritance is a mechanism in object-oriented programming that allows a new class to be based on an existing one, inheriting its properties and methods. The new class is called the subclass, and the existing class is called the superclass (baseclass). The subclass can then add or modify the inherited properties and methods to suit its specific needs.

- Why is inheritance important?

  > Inheritance is important because it allows code to be reused, reducing duplication and increasing efficiency. It also allows developers to create hierarchies of classes that reflect the relationships between objects in the real world, making the code easier to understand and maintain.

- How is inheritance achieved?

  > Inheritance is achieved through the use of the "extends" keyword in Java or the ":" symbol in C# and Python. The subclass extends the superclass, inheriting all of its properties and methods. The subclass can then override the inherited methods or add new ones to provide specialized behavior.

- How is inheritance achieved?

  > - **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.

Example of Inheritance in C#

One common example of inheritance in C# is the use of a base class to define common functionality and then deriving specialized classes from it. We have a base class `Character` that represents a character in a game with properties such as `Name`, `Health`, `Attack`, and `Defense`. We also have a derived class `Player` that inherits from the `Character` class and adds an additional property `Level and PlayerType`. The `NonPlayer` class has properties such as `NPCType`.

Code Example

▼ Character.cs

```
public class Character
{
  private string name;
  public string Name
  {
    get { return this.name; }
    private set
    {
      this.name  = this.CheckName(value);
    }
  }
  private int health = 100;
  public int Health { get {return health} private set {value = this.health;}}

  public int Attack { get; set;}
  public int Defense { get; set;}

  public Character(string name)
  {
    Name = name;
  }
  public void TakeDamage(int damage)
  {
    health -= damage;
  }
  public bool IsAlive()
  {
    return health > 0;
  }
}
```

▼ Player.cs

```
public class Player: Character
{
  public Player(string name): base(name) {}
  public int Level { get; set; } = 1;
  public PlayerTypeEnums playerType { get; set;
}
```

▼ NonPlayer.cs

```
public class NonPlayer: Character
{
  public NonPlayer(string name): base(name) {}
  public NPCTypeEnums NPCType { get; set; }
}
```

Abstraction in Object-Oriented Programming

- What is Abstraction?

  > Abstraction is the process of hiding complex implementation details and exposing only the essential features of an object.
  >
  > This is an important point, a class with an abstract method automatically becomes abstract itself and no objects are created from abstract classes

- Why is abstraction important?

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

- How is abstraction achieved?

  > Abstraction is achieved through the use of classes and interfaces. A class defines the common properties and methods for a group of objects, while an interface defines a contract for behavior that a class must implement.

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**.

Code Example

▼ ICharacter.cs

```
public interface ICharacter
{
   public void TakeDamage(int damage);
   public void Heal(int health);
}
```

▼ Character.cs

```
public abstract class Character : ICharacter
{
   ....
   public void TakeDamage(int damage)
   {
      Health -= damage;
   }
   public void Heal(int health)
   {
      this.health = Math.Min(this.health + health, 100);
   }
   public abstract void Voice();
   ....
}
```

▼ Player.cs

```
public class Player: Character
{
   ....
   public override void Voice()
   {
      Console.WriteLine("Hello, I'm a player.");
   }
   ....
}
```

▼ NonPlayer.cs

```
public class NonPlayer: Character
{
   ....
   public override void Voice()
   {
      Console.WriteLine("Hello, I'm a NPC.");
   }
  ....
}
```

Polymorphism in Object-Oriented Programming

- What is Polymorphism?

  > Polymorphism is a key concept in object-oriented programming that allows objects of different classes to be treated as if they were of the same class. This allows for more flexibility and modularity in the code, as well as the ability to write more generic and reusable functions.

- Why is polymorphism important?

  > Polymorphism is important because it allows developers to write code that is more flexible and adaptable to different situations. It also promotes code reuse and modularity, which can save time and effort in the long run.

- How is polymorphism achieved?

  > Polymorphism is achieved through the use of inheritance, interfaces, and method overriding. Inheritance allows objects of subclasses to be treated as objects of the superclass. Interfaces provide a way to specify a set of methods that a class must implement, allowing objects of different classes to be treated as if they had the same interface. Method overriding allows a subclass to provide its own implementation of a method defined in the superclass, allowing for more specialized behavior.

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.

Code Example

▼ Character.cs

```
public class Character
{
  ....
//This is called dynamic polymorphism. The name of the operation is method overriding.

  protected virtual void CheckName()
  {
    // Default implementation for checking name a character
  }

  ....
}
```

▼ Player.cs

```
public class Player : Character
{
  ....

  public override void CheckName()
  {
    // Check the player character with a name pattern
  }

  ....
}
```

▼ NonPlayer.cs

```
public class NonPlayer : Character
{
  ....

  public override void CheckName()
  {
    // Check the player character with a name pattern
  }

  ....
}
```

▼ Attack.cs

```
public class Attack
{
  ....
//This is called static polymorphism and the process is called method overloading.

  public int AttackCharacter(int baseDamage)
  {
    // Calculate the actual damage based on the attacker's strength, the  target's defense, and other
factors
    var actualDamage = Math.Max(baseDamage - Target.Defense, 0);
    Target.TakeDamage(actualDamage);
    return actualDamage;
  }

  public int AttackCharacter()
  {
    // Calculate the actual damage based on the attacker's strength, the target's defense, and other
factors
    var actualDamage = Math.Max(Convert.ToInt32((Attacker.Strength - (Target.Defense * 0.75))), 0);
    Target.TakeDamage(actualDamage);
    return actualDamage;
  }
  ....
}
```

Conclusion

In conclusion, object-oriented programming is a programming paradigm that revolves around objects and classes. It offers several benefits and advantages in software development, including code reuse, abstraction, and polymorphism. OOP has practical applications in a variety of fields, including GUI development, game development, and web development. As a computer science student, learning OOP can help you become a more skillful and effective developer.