Object-Oriented Programming (OOP) Principles

Object-Oriented Programming (OOP) is a paradigm that organizes code into objects, which are instances of classes. C# is a popular language for OOP, offering robust support for its core principles: **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**. This material focuses on the requested topics with C# examples.

1. Object-Oriented Programming Principles

What is OOP?

OOP models real-world entities as objects that contain data (fields/properties) and behavior (methods). It emphasizes modularity, reusability, and maintainability.

Core Principles of OOP

1. **Encapsulation**: Bundling data and methods within a class, restricting direct access to some components.
2. **Inheritance**: Allowing a class to inherit properties and methods from another class.
3. **Polymorphism**: Enabling objects of different types to be treated as instances of a common type.
4. **Abstraction**: Hiding implementation details and exposing only essential features (often tied to abstract classes/interfaces in C#).

2. Classes and Objects

Classes

- A **class** defines the structure and behavior of objects. It includes fields, properties, and methods.
- In C#, classes are declared with the `class` keyword.

Objects

- An **object** is an instance of a class, created using the `new` keyword.

Example (in C#)

```
// Defining a class
public class Dog
{
    // Fields (private by convention)
    private string name;
    private int age;

    // Constructor
    public Dog(string name, int age)
    {
        this.name = name;  // "this" disambiguates instance variables
        this.age = age;
    }

    // Method
    public string Bark()
    {
        return $"{name} says Woof!";
    }

    // Properties (optional, for controlled access)
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}

// Using the class
class Program
{
    static void Main()
    {
        // Creating objects
        Dog dog1 = new Dog("Buddy", 3);
        Dog dog2 = new Dog("Max", 5);

        // Accessing properties and methods
        Console.WriteLine(dog1.Name);     // Output: Buddy
        Console.WriteLine(dog2.Bark());   // Output: Max says Woof!
    }
}
```

Key Points

- Constructors in C# have the same name as the class and no return type.
- Properties (`Name`) provide a C#-specific way to encapsulate fields with `get` and `set` accessors.
- Objects are instantiated with `new`.

---

3. Inheritance

What is Inheritance?

Inheritance allows a **derived class** to inherit members (fields, properties, methods) from a **base class**. In C#, a class can inherit from only one base class (single inheritance).

Example (in C#)

```
// Base class
public class Animal
```

```csharp
{
    public string Species { get; set; }

    public Animal(string species)
    {
        Species = species;
    }

    public virtual string MakeSound()  // "virtual" allows overriding
    {
        return "Some generic sound";
    }
}

// Derived class
public class Cat : Animal  // Inherit using ":"
{
    public string Name { get; set; }

    public Cat(string name) : base("Feline")  // Call base constructor
    {
        Name = name;
    }

    public override string MakeSound()  // Override base method
    {
        return "Meow";
    }
}

// Usage
class Program
{
    static void Main()
    {
        Cat cat = new Cat("Whiskers");
        Console.WriteLine(cat.Species);      // Output: Feline
        Console.WriteLine(cat.MakeSound());  // Output: Meow
    }
}
```

Key Points

- Use `:` to indicate inheritance.
- The `base` keyword calls the base class constructor or methods.
- `virtual` methods in the base class can be overridden with `override` in the derived class.

---

4. Polymorphism

What is Polymorphism?

Polymorphism allows methods to have different implementations depending on the object type, achieved in C# through **method overriding** (runtime polymorphism). C# does not support method overloading as compile-time polymorphism in the same way as C++, but it supports parameter-based overloading.

Example (in C#)

```
// Base class
public class Animal
{
    public virtual string Speak()
    {
        return "Generic animal sound";
    }
}

// Derived classes
public class Dog : Animal
{
    public override string Speak()
    {
        return "Woof";
    }
}

public class Cat : Animal
{
    public override string Speak()
    {
        return "Meow";
    }
}

// Polymorphism in action
class Program
{
    static void Main()
    {
        Animal[] animals = new Animal[] { new Dog(), new Cat() };
        foreach (Animal animal in animals)
        {
            Console.WriteLine(animal.Speak());  // Output: Woof, then Meow
        }
    }
}
```

Key Points

- `virtual` and `override` enable runtime polymorphism.
- An array of the base type (`Animal[]`) can hold derived class objects (`Dog`, `Cat`).
- The actual method called depends on the object’s runtime type, not the reference type.

5. Encapsulation

What is Encapsulation?

Encapsulation restricts direct access to an object’s data, exposing it only through controlled methods or properties. In C#, this is achieved with access modifiers (`private`, `protected`, `public`) and properties.

Example (in C#)

```
public class BankAccount
{
    private decimal balance;  // Private field
    public string Owner { get; set; }  // Public property

    public BankAccount(string owner, decimal initialBalance)
    {
        Owner = owner;
        balance = initialBalance;
    }

    // Public method to access private field
    public decimal GetBalance()
    {
        return balance;
    }

    // Public method to modify private field
    public bool Deposit(decimal amount)
    {
        if (amount > 0)
        {
            balance += amount;
            return true;
        }
        return false;
    }
}

// Usage
class Program
{
    static void Main()
    {
        BankAccount account = new BankAccount("Alice", 1000m);  // "m" for decimal
        Console.WriteLine(account.GetBalance());  // Output: 1000
        account.Deposit(500m);
        Console.WriteLine(account.GetBalance());  // Output: 1500
        // Console.WriteLine(account.balance);    // Error: balance is private
    }
}
```

Key Points

- `private` fields can only be accessed within the class.
- Properties or methods (`GetBalance`, `Deposit`) provide controlled access.
- C# properties (`Owner`) are a concise alternative to traditional getters/setters.

Practical Application Example

Here’s a combined example in C#:

```
// Base class with encapsulation
public class Vehicle
{
    private string brand;

    public Vehicle(string brand)
    {
        this.brand = brand;
    }

    public string GetBrand()
    {
        return brand;
    }

    public virtual string Move()
    {
        return "Moving...";
    }
}

// Derived class
public class Car : Vehicle
{
    public string Model { get; set; }

    public Car(string brand, string model) : base(brand)
    {
        Model = model;
    }

    public override string Move()
    {
        return $"{GetBrand()} {Model} is driving!";
    }
}

// Derived class
public class Bike : Vehicle
{
    public Bike(string brand) : base(brand) { }
    public override string Move()
    {
        return $"{GetBrand()} bike is pedaling!";
    }
}

// Usage
class Program
{
    static void Main()
    {
        Vehicle[] vehicles = new Vehicle[] { new Car("Toyota", "Camry"), new Bike("Trek") };
        foreach (Vehicle vehicle in vehicles)
        {
            Console.WriteLine(vehicle.Move());
        }
        // Output:
        // Toyota Camry is driving!
        // Trek bike is pedaling!
    }
}
```

Summary Table

Concept	Definition	Key Feature
Classes & Objects	Blueprint (class) and instances (objects)	Encapsulates data and behavior
Inheritance	Reusing code from a base class	Promotes code reuse
Polymorphism	Same interface, different implementations	Flexibility via overriding
Encapsulation	Data hiding and bundling	Controlled access via properties

Exercises

1. Create a `Person` class with `Name` and `Age` properties, and a `Introduce()` method. Then create a `Student` class that inherits from `Person` and adds a `Grade` property.
2. Implement polymorphism with a `Shape` class and an `Area()` method, overridden in `Circle` and `Rectangle` classes.
3. Use encapsulation to create a `Library` class with a private `List<string>` for books, and public methods to add or remove books.