Detailed Explanation of OOP Concepts in C#

Overview of OOP Concepts

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects, which are instances of classes. The four fundamental principles of OOP are Encapsulation, Inheritance, Polymorphism, and Abstraction. Below is a summary table of these concepts, followed by detailed explanations with C# examples.

Concept	**Purpose**	**Keyword**
Encapsulation	Hide internal state using private fields	`private`, methods
Inheritance	Reuse code from base classes	`: base`
Polymorphism	Different behavior for the same method	`virtual`, `override`
Abstraction	Define blueprint methods in base class	`abstract`, `override`

Table 1: Summary of OOP Concepts

1 Encapsulation

Encapsulation is the process of hiding the internal state of an object and only exposing necessary functionality through public methods or properties. This ensures data integrity by restricting direct access to an object's fields, allowing controlled interactions via defined interfaces.

Benefits:

- Protects data from unauthorized access or modification.
- Enhances maintainability by isolating internal implementation details.
- Promotes modularity and reduces complexity.

Example: A `BankAccount` class that encapsulates the balance field and provides methods to interact with it.

```
public class BankAccount
{
    private decimal balance;

    public void Deposit(decimal amount)
    {
        if (amount > 0)
            balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount > 0 && amount <= balance)
            balance -= amount;
    }

    public decimal GetBalance()
    {
        return balance;
    }
```

```
}
```

Usage: Demonstrates how to use the `BankAccount` class to perform deposits and withdrawals while keeping the balance private.

```
BankAccount acc = new BankAccount();
acc.Deposit(1000);
acc.Withdraw(400);
Console.WriteLine(acc.GetBalance());  // Output: 600
```

The `private` keyword ensures the `balance` field cannot be accessed directly from outside the class, preventing invalid modifications. Methods like `Deposit` and `Withdraw` include validation logic to maintain data integrity, showcasing encapsulation's role in safeguarding object state.

2 Inheritance

Inheritance allows a derived class (child) to inherit fields, properties, and methods from a base class (parent). This promotes code reuse and establishes a hierarchical relationship between classes, where the derived class can extend or specialize the base class's functionality.

Benefits:

- Reduces code duplication by reusing base class functionality.
- Facilitates a clear hierarchical structure in code.
- Enables extensibility by allowing derived classes to add new features.

Example: A `Person` base class and a `Student` derived class that inherits from it.

```
public class Person
{
    public string Name;

    public Person(string name)
    {
        Name = name;
    }

    public void Greet()
    {
        Console.WriteLine($"Hi, I am {Name}");
    }
}

public class Student : Person
{
    public Student(string name) : base(name) { }

    public void Study()
    {
        Console.WriteLine($"{Name} is studying.");
    }
}
```

Usage: Shows how the `Student` class inherits the `Greet` method and adds its own `Study` method.

```
Student student = new Student("Kiran");
student.Greet();  // Inherited from Person: Hi, I am Kiran
student.Study();  // Student's own method: Kiran is studying.
```

The `: base(name)` syntax in the `Student` constructor calls the base class's constructor, ensuring proper initialization. Inheritance enables the `Student` class to reuse the `Greet` method without redefining it, demonstrating code reuse and the "is-a" relationship (a `Student` is a `Person`).

3 Polymorphism

Polymorphism allows objects of different classes to be treated as instances of a common base class, with methods behaving differently based on the actual object type. In C#, polymorphism is achieved through method overriding (runtime polymorphism) or method overloading (compile-time polymorphism). This section focuses on runtime polymorphism via overriding.

Benefits:

- Enhances flexibility by allowing interchangeable use of derived classes.
- Simplifies code by enabling generic processing of objects.
- Supports extensibility for adding new derived classes without modifying existing code.

Example: A `Person` base class with a virtual `Greet` method, overridden by `Student` and `Teacher` derived classes.

```
public class Person
{
    public string Name;

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

    public virtual void Greet()
    {
        Console.WriteLine($"Hello, I am {Name}");
    }
}

public class Student : Person
{
    public Student(string name) : base(name) { }

    public override void Greet()
    {
        Console.WriteLine($"Hi, I am student {Name}");
    }
}

public class Teacher : Person
{
    public Teacher(string name) : base(name) { }

    public override void Greet()
    {
        Console.WriteLine($"Good day, I am teacher {Name}");
    }
}
```

Usage: Demonstrates how a list of `Person` objects can call the appropriate `Greet` method based on the actual object type.

```
List<Person> people = new List<Person>
{
    new Student("Kiran"),
    new Teacher("Asha"),
    new Person("Guest")
};

foreach (var person in people)
{
    person.Greet();  // Calls correct method based on object type
}
// Output:
```

```
// Hi, I am student Kiran
// Good day, I am teacher Asha
// Hello, I am Guest
```

The `virtual` keyword in the base class allows the `Greet` method to be overridden in derived classes using the `override` keyword. At runtime, C# determines the actual type of each object and invokes the corresponding `Greet` method, showcasing runtime polymorphism. This enables flexible and extensible code, as new derived classes can be added without modifying the loop logic.

4 Abstraction

Abstraction involves defining a blueprint for classes by specifying what methods must be implemented, without providing their implementation. In C#, abstraction is achieved using abstract classes or interfaces. Abstract classes can include both abstract (unimplemented) and concrete (implemented) members, forcing derived classes to implement the abstract members.

Benefits:

- Enforces a contract for derived classes to follow.
- Hides implementation details, exposing only essential functionality.
- Promotes consistency across related classes.

Example: An abstract `Animal` class with an abstract `MakeSound` method, implemented by `Dog` and `Cat` derived classes.

```
public abstract class AAP
{
    public abstract void MakeSound();
}

public class Dog : Animal
{
    public override void Area()
    {
        Console.WriteLine("Dog is Circle");
    }
}

public class Cat : Animal
{
    public override void Area()
    {
        Console.WriteLine("Cat is Square");
    }
}
```

Usage: Shows how a list of `Animal` objects can call the implemented `MakeSound` method.

```
List<Animal> animals = new List<Animal>
{
    new Dog(),
    new Cat()
};

foreach (var animal in animals)
{
    animal.MakeSound();
}
// Output:
// Woof!
// Meow!
```

The `abstract` keyword ensures the `Animal` class cannot be instantiated and that the `MakeSound` method must be implemented by derived classes. Each derived class (`Dog` and `Cat`) provides its own implementation of `MakeSound`, fulfilling the abstract contract. This approach ensures that all animals share a consistent interface while allowing specific behavior, demonstrating abstraction's role in defining essential behavior without implementation details.