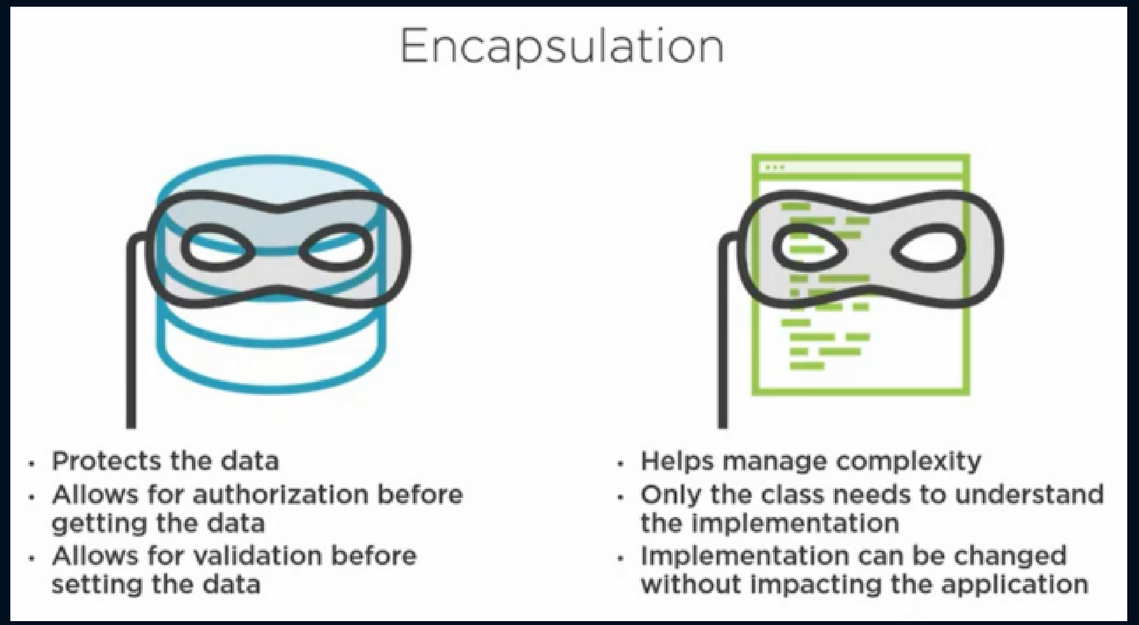
Encapsulation

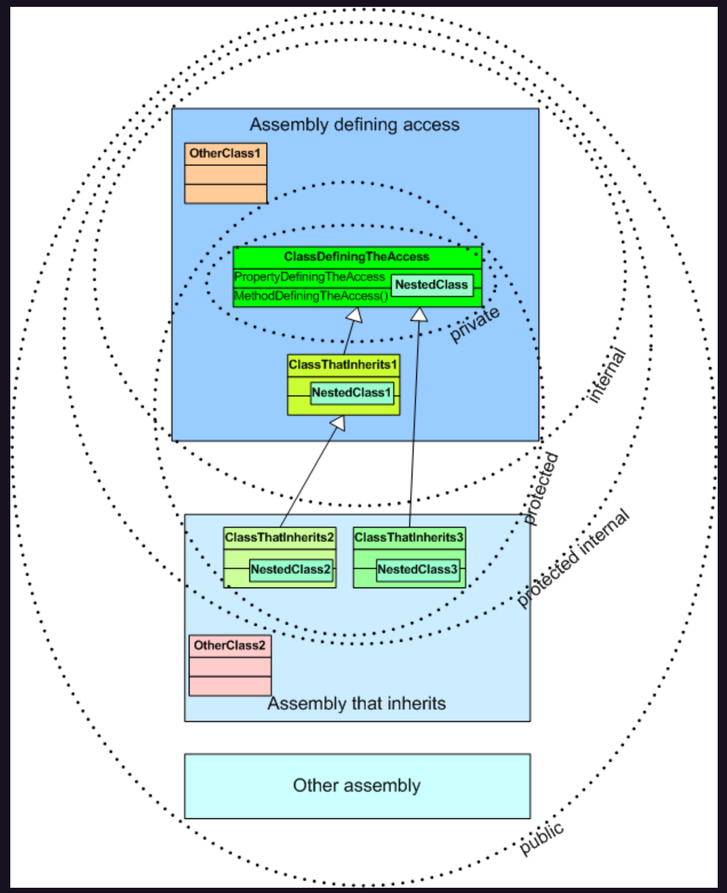
Encapsulation focuses on bundling data and methods together within a class, hiding internal implementation details and exposing a well-defined interface to interact with the object.

Benefits of Encapsulation

- Data Protection: protects the internal state of an object by making data members private. It ensures that data can only be accessed and modified through controlled methods.
- Code Modularity: promotes code modularity by encapsulating related data and behavior into a single unit.
- Security and Integrity: by preventing unauthorized access or modifications.
- Code Flexibility: allows the internal implementation details to be changed without affecting the external code.



Access Modifiers



Access modifiers

Encapsulate collections

Encapsulating collections is an important aspect of encapsulation in C#. It helps to maintain the integrity and control access to the collection's data. Here's an example of encapsulating a collection using a private field and exposing it through a read-only property.

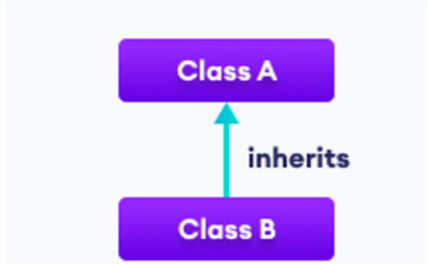
In addition to sealed classes, collection encapsulation, and access modifiers, there are several other techniques to achieve encapsulation in C#:

1. Property Encapsulation
2. Method Encapsulation
3. Interface Encapsulation: Interfaces a

```
public class ShoppingCart
{
    private List<Product> products;

    public ShoppingCart()
    {
        products = new List<Product>();
    }

    public IReadOnlyList<Product> Products
    {
        get { return products.AsReadOnly(); }
    }

    public void AddProduct(Product product)
    {
        products.Add(product);
        Console.WriteLine($"Added product: {product.Name}");
    }

    public void RemoveProduct(Product product)
    {
        products.Remove(product);
        Console.WriteLine($"Removed product: {product.Name}");
    }
}

public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

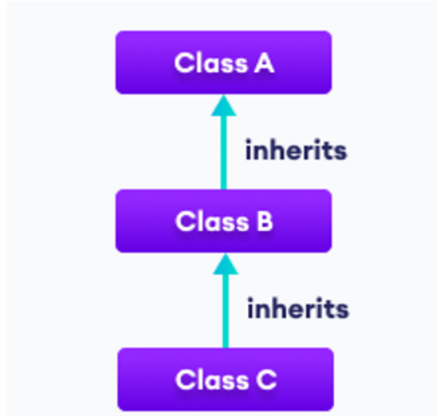
Encapsulate a Collection

Inheritance types



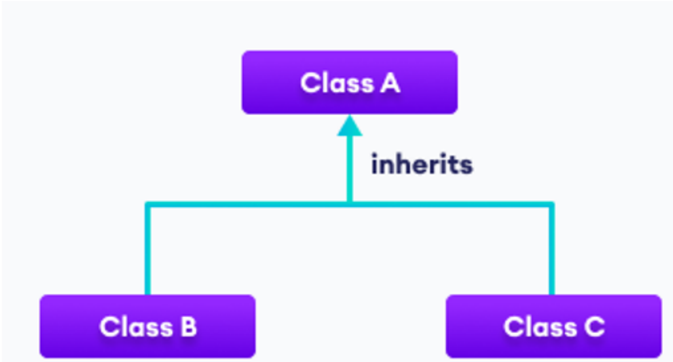
C# Single Inheritance

Derived class inherits from One base class



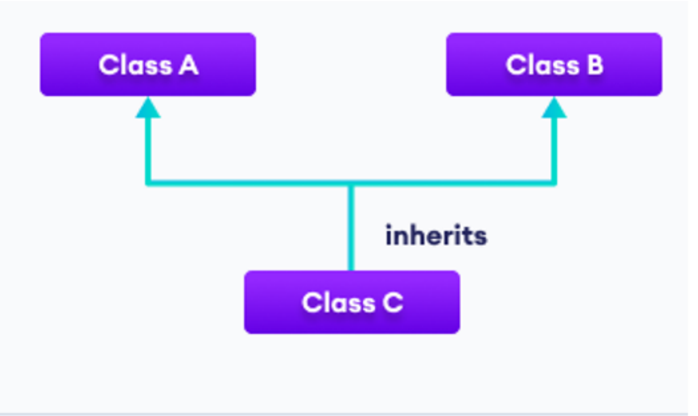
C# Multilevel Inheritance

A derived class inherits from a base class, the derived class can act like a base class for another class



C# Hierarchical Inheritance

Multiple derived class inherit from a base class



Multiple Inheritance

A derived class inherits from multiple base classes, this is supported only using **Interfaces**

Inheritance in C#

Inheritance example

Base class

```
class Shape {
    double a_width;
    double a_length;
    // Default constructor
    public Shape()
    {
        Width = Length = 0.0;
    }
    // Constructor for Shape
    public Shape(double w, double l)
    {
        Width = w;
        Length = l;
    }
    // Construct an object with
    // equal length and width
    public Shape(double y)
    {
        Width = Length = y;
    }
    // Properties for Length and Width
    public double Width
    {
        get { return a_width; }

        set { a_width = value < 0 ? -value : value; }
    }

    public double Length
    {
        get { return a_length; }

        set { a_length = value < 0 ? -value : value; }
    }
    public void DisplayDim()
    {
        Console.WriteLine("Width and Length are " |
                    + Width + " and " + Length);
    }
}
```

Derived class from Shape

```
class Rectangle : Shape {

    string Style;
    // A default constructor.
    // This invokes the default
    // constructor of Shape.
    public Rectangle()
    {
        Style = "null";
    }
    // Constructor
    public Rectangle(string s, double w, double l)
        : base(w, l)
    {
        Style = s;
    }
    // Construct an square.
    public Rectangle(double y)
        : base(y)
    {
        Style = "square";
    }
    // Return area of rectangle.
    public double Area()
    {
        return Width * Length;
    }
    // Display a rectangle's style.
    public void DisplayStyle()
    {
        Console.WriteLine("Rectangle is  " + Style);
    }
}
```

Derived class from Rectangle

```
class ColorRectangle : Rectangle {

    string rcolor;

    // Constructor
    public ColorRectangle(string c, string s,
                          double w, double l)
        : base(s, w, l)
    {
        rcolor = c;
    }

    // Display the color.
    public void DisplayColor()
    {
        Console.WriteLine("Color is " + rcolor);
    }
}
```

Inheritance in C#

Inheritance pitfalls

Inheritance is a powerful mechanism in C# that promotes code reuse, extensibility, and modularity. It allows you to create hierarchies of related classes and build upon existing functionality.

While inheritance is a powerful feature in object-oriented programming, it also comes with certain pitfalls and challenges.

> Inflexible Hierarchy: Inheritance creates a fixed hierarchy of classes, which can limit flexibility and extensibility. Adding new behavior or modifying existing behavior in the inheritance chain may require changing multiple classes
>
> Fragile Base Class Problem: Modifying a base class can inadvertently introduce bugs or unexpected behavior in derived classes.
>
> Lack of Encapsulation: Inheritance can expose implementation details of base classes to derived classes, violating the principle of encapsulation.
>
> Overuse of Inheritance: Inheritance should be used judiciously and when it truly represents an "is-a" relationship. Overuse of inheritance can lead to overly complex class hierarchies, making the code harder to understand and maintain.

Static Polymorphism

It allows objects of different types to be treated as objects of a common base type.
Enables code to be written that can work with objects of multiple related classes through a single interface.

Compile-Time Polymorphism:

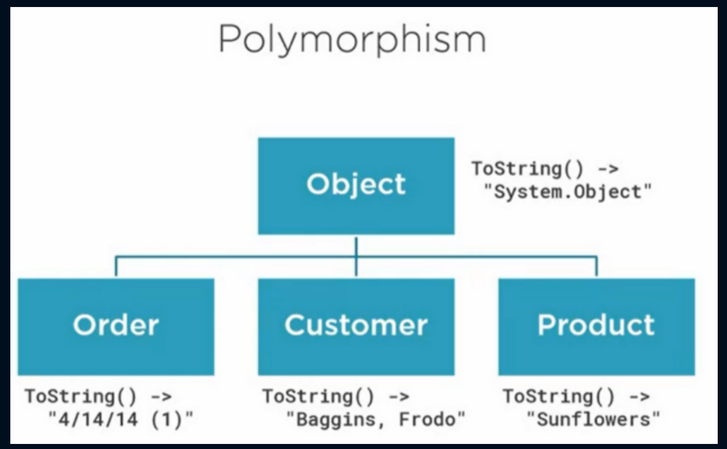
- Also known as method overloading or static polymorphism.
- Multiple methods with the same name but different parameters are defined in the same class.
- The appropriate method is selected based on the arguments' types during compile-time.

```
public class MathUtils
{
    public static int Add(int a, int b)
    {
        return a + b;
    }

    public static double Add(double a, double b)
    {
        return a + b;
    }
}

int result1 = MathUtils.Add(5, 3);
double result2 = MathUtils.Add(2.5, 4.7);
```

Static polymorphism

Dynamic Polymorphism

Also known as method overriding or dynamic polymorphism. Occurs when a derived class overrides a method defined in its base class. The method to be executed is determined at runtime based on the actual object type, not the reference type.



Method Overriding

```
public class Interest
{
    public virtual double TrueBank(double amount, double rate)
    {
        return amount + (amount * rate);
    }
}

public class SimpleInterest: Interest
{
    public override double TrueBank(double amount, double rate)
    {
        return amount + (amount * rate) + 1000;
    }
}

public class FixedInterest: Interest
{
    public override double TrueBank(double amount, double rate)
    {
        return amount + (amount * rate) + 1500;
    }
}


Interest i = new Interest();
double finalamount = i.TrueBank(5000.00,0.1);
Console.WriteLine($"Normal interest for a holder {finalamount}");

i = new SimpleInterest();
finalamount = i.TrueBank(5000.00,0.1);
Console.WriteLine($"Simple interest for a holder {finalamount}");

i = new FixedInterest();
finalamount = i.TrueBank(5000.00,0.1);
Console.WriteLine($"Fixed interest for a holder {finalamount}");
```

Example

Differences and Operator overloading

Method Overloading	Method Overriding
Method overloading means methods with the same name but different signature (number and type of parameters) in the same scope.	Method overriding means methods with the same name and same signature but in a different scope.
It is performed within a class, and inheritance is not involved.	It requires two classes, and inheritance is involved.
The return type may be the same or different.	The return type must be the same.
It is an example of compile-time polymorphism.	It is an example of run-time polymorphism.
Static methods can be overloaded.	Static methods can't be overridden.

Differences of static and dynamic Polymorphism

Method overloading vs Method overriding

Operators	Description
+, -, !, ~, ++, --, true, false	These unary operators take one operand can be overloaded
+, -, *, /, %	These binary operators take two operands and can be overloaded
==, !=, <, >, <=, >=	The comparison operators can be overloaded
&&, ||	The conditional logical operators can't be overloaded directly but these can be evaluated by using the & and | which can be overloaded
+=, -=, *=, /==, %==	The compound assignment operators can't be overloaded
^, =, ? :, ->, is, new, sizeof, typeof, nameof, default	These operators can't be overloaded

Operator overloading support

Operator overloading

Task

Exercise/ Homework

Quiz Game Console App:

- The application will be a console-based quiz game.
- Users will be able to play different quizzes and test their knowledge.

1. Quiz Class:
   - Implement a Quiz class to represent a quiz.
   - The Quiz class should have properties to store the quiz name, description, and a collection of questions.
   - The collection of questions can be of two types: Multiple-choice or Fill-in-the-blank.
2. Question Class:
   - Implement a Question class to represent a question in the quiz.
   - The Question class should have the following properties:
     - Question text,
     - Answer choices (for Multiple-choice questions),
     - Correct answer.
     - Score
3. Gameplay Flow:
   - Users should be able to select a quiz from a list of available quizzes.
   - Once a quiz is selected, the questions should be presented one by one to the user.
   - The user should be able to input their answers or make selections based on the question type.
   - After answering all the questions, the score should be calculated based on the correct answers.
   - The score should be displayed at the end of the quiz.
   - The user should be able to pick another available quiz
   - There should be an option to quit the application

NOTE: Using UML for designing the solution is a plus