# Introduction of Inheritance

### **Assisted Problems**

**1. Animal Hierarchy**

**Description:**  
Create a hierarchy where **Animal** is the superclass, and **Dog**, **Cat**, and **Bird** are subclasses. Each subclass has a unique behavior.

**Tasks:**

1. **Define a superclass Animal:**
   * Add two attributes: Name (string) and Age (integer).
   * Add a method MakeSound() that provides a generic sound message (e.g., "Animal makes a sound").
2. **Define subclasses Dog, Cat, and Bird:**
   * Each subclass should **override** the MakeSound() method to provide its unique behavior (e.g., "Dog barks", "Cat meows", "Bird chirps").
3. **Goal:**
   * Learn basic **inheritance**, **method overriding**, and **polymorphism** by calling MakeSound() on instances of different subclasses.

| using System; // Base class Animal class Animal {  public string Name { get; set; }  public int Age { get; set; }   // Constructor  public Animal(string name, int age)  {  Name = name;  Age = age;  }   // Virtual method  public virtual string MakeSound()  {  return "Animal makes a sound";  } }  // Derived class Dog class Dog : Animal {  public Dog(string name, int age) : base(name, age) { }   // Overriding MakeSound method  public override string MakeSound()  {  return "Dog barks";  } }  // Derived class Cat class Cat : Animal {  public Cat(string name, int age) : base(name, age) { }   // Overriding MakeSound method  public override string MakeSound()  {  return "Cat meows";  } }  // Derived class Bird class Bird : Animal {  public Bird(string name, int age) : base(name, age) { }   // Overriding MakeSound method  public override string MakeSound()  {  return "Bird chirps";  } }  // Main program class Program {  static void Main()  {  // Creating instances of Dog, Cat, and Bird  Animal dog = new Dog("Tom", 3);  Animal cat = new Cat("Sony", 2);  Animal bird = new Bird("Methu", 1);   // Storing instances in an array  Animal[] animals = { dog, cat, bird };    // Iterating through the array and calling MakeSound method  foreach (var animal in animals)  {  Console.WriteLine(animal.Name+" ("+animal.Age+" years old): "+animal.MakeSound());  }  } } |
| --- |

**2. Employee Management System**

**Description:**  
Create an **Employee** hierarchy for different employee types such as **Manager**, **Developer**, and **Intern**.

**Tasks:**

1. **Define a base class Employee:**
   * Add three attributes: Name (string), Id (integer), and Salary (double).
   * Add a method DisplayDetails() to display the details of an employee.
2. **Define subclasses Manager, Developer, and Intern:**
   * **Manager:** Add an additional attribute TeamSize (integer).
   * **Developer:** Add an additional attribute ProgrammingLanguage (string).
   * **Intern:** Add an additional attribute InternshipDuration (string).
3. **Goal:**
   * Practice **inheritance** by creating subclasses with specific attributes and overriding superclass methods (e.g., DisplayDetails()) to display details specific to each type of employee

| using System; // Base class Employee public class Employee {    public string Name { get; set; }  public int Id { get; set; }  public double Salary { get; set; }   // Constructor  public Employee(string name, int id, double salary)  {  this.Name = name;  this.Id = id;  this.Salary = salary;  }   // Method to display Employee details  public virtual void DisplayDetails()  {  Console.WriteLine("Name: "+Name+", Id: "+Id+", Salary: "+Salary);  } }  // Subclass Manager public class Manager : Employee {  public int TeamSize { get; set; }   // Constructor  public Manager(string name, int id, double salary, int teamSize)  : base(name, id, salary)  {  this.TeamSize = teamSize;  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Team Size: "+TeamSize);  } }  // Subclass Developer public class Developer : Employee {  public string ProgrammingLanguage { get; set; }   // Constructor  public Developer(string name, int id, double salary, string programmingLanguage)  : base(name, id, salary)  {  this.ProgrammingLanguage = programmingLanguage;  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Programming Language: "+ProgrammingLanguage);  } }  // Subclass Intern public class Intern : Employee {  public string InternshipDuration { get; set; }   // Constructor  public Intern(string name, int id, double salary, string internshipDuration)  : base(name, id, salary)  {  this.InternshipDuration = internshipDuration;  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Internship Duration: "+InternshipDuration);  } }  // Main program class Program {  static void Main(string[] args)  {  // Creating instances of different employee types  Employee manager = new Manager("Abhishek", 101, 95000, 10);  Employee developer = new Developer("Bobby", 102, 80000, "C#");  Employee intern = new Intern("Tushar", 103, 40000, "6 months");   // Displaying details of each employee  manager.DisplayDetails();  Console.WriteLine();   developer.DisplayDetails();  Console.WriteLine();   intern.DisplayDetails();  } } |
| --- |

**3. Vehicle and Transport System**

**Description:**  
Design a **vehicle hierarchy** where **Vehicle** is the superclass, and **Car**, **Truck**, and **Motorcycle** are subclasses with unique attributes.

**Tasks:**

1. **Define a superclass Vehicle:**
   * Add two attributes: MaxSpeed (integer) and FuelType (string).
   * Add a method DisplayInfo() to display the vehicle's information.
2. **Define subclasses** Car, Truck, and Motorcycle that inherit from Vehicle:
   * **Car:** Add an additional attribute SeatCapacity (integer).
   * **Truck:** Add an additional attribute PayloadCapacity (integer).
   * **Motorcycle:** Add an additional attribute HasSidecar (boolean).
3. **Demonstrate polymorphism:**
   * Store objects of type Car, Truck, and Motorcycle in an array of Vehicle type.
   * Call the DisplayInfo() method on each object in the array and observe dynamic method dispatch.

**Goal**: Understand how inheritance helps in organizing shared and unique features across subclasses and use polymorphism for dynamic method calls.

| using System; // Base class Vehicle public class Vehicle {  public int MaxSpeed { get; set; }  public string FuelType { get; set; }   // Constructor  public Vehicle(int maxSpeed, string fuelType)  {  this.MaxSpeed = maxSpeed;  this.FuelType = fuelType;  }   // Method to display Vehicle information  public virtual void DisplayInfo()  {  Console.WriteLine(""+MaxSpeed+", Fuel Type: "+FuelType);  } }  // Subclass Car public class Car : Vehicle {  public int SeatCapacity { get; set; }   // Constructor  public Car(int maxSpeed, string fuelType, int seatCapacity)  : base(maxSpeed, fuelType)  {  this.SeatCapacity = seatCapacity;  }   // Overriding the DisplayInfo method  public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine("Seat Capacity: "+SeatCapacity);  } }  // Subclass Truck public class Truck : Vehicle {  public int PayloadCapacity { get; set; }   // Constructor  public Truck(int maxSpeed, string fuelType, int payloadCapacity)  : base(maxSpeed, fuelType)  {  this.PayloadCapacity = payloadCapacity;  }   // Overriding the DisplayInfo method  public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine("Payload Capacity: "+PayloadCapacity);  } }  // Subclass Motorcycle public class Motorcycle : Vehicle {  public bool HasSidecar { get; set; }   // Constructor  public Motorcycle(int maxSpeed, string fuelType, bool hasSidecar)  : base(maxSpeed, fuelType)  {  this.HasSidecar = hasSidecar;  }   // Overriding the DisplayInfo method  public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine("Has Sidecar: "+HasSidecar);  } }  // Main program class Program {  static void Main(string[] args)  {  // Creating instances of different vehicle types  Vehicle car = new Car(180, "Petrol", 6);  Vehicle truck = new Truck(120, "Diesel", 2500);  Vehicle motorcycle = new Motorcycle(150, "Petrol", true);   // Storing the objects in an array of Vehicle type  Vehicle[] vehicles = { car, truck, motorcycle };   // Calling the DisplayInfo method on each object in the array  foreach (Vehicle vehicle in vehicles)  {  vehicle.DisplayInfo();  Console.WriteLine();  }  } } |
| --- |

## Single Inheritance

### **Sample Problem 1: Library Management with Books and Authors**

* + **Description**: Model a Book system where Book is the superclass, and Author is a subclass.
  + **Tasks**:
    - Define a superclass Book with attributes like Title and PublicationYear.
    - Define a subclass Author with additional attributes like Name and Bio.
    - Create a method DisplayInfo() to show details of the book and its author.
  + **Goal**: Practice single inheritance by extending the base class and adding more specific details in the subclass.

| using System; // Superclass Book public class Book {  public string Title { get; set; }  public int PublicationYear { get; set; }   // Constructor  public Book(string title, int publicationYear)  {  this.Title = title;  this.PublicationYear = publicationYear;  }   // Method to display book information  public virtual void DisplayInfo()  {  Console.WriteLine("Title: "+Title+", Publication Year: "+PublicationYear);  } }  // Subclass Author public class Author : Book {  public string Name { get; set; }  public string Bio { get; set; }   // Constructor  public Author(string title, int publicationYear, string name, string bio)  : base(title, publicationYear)  {  this.Name = name;  this.Bio = bio;  }   // Overriding the DisplayInfo method  public override void DisplayInfo()  {  base.DisplayInfo();   Console.WriteLine("Author: "+Name+", Bio: "+Bio);  } }  // Main program class Program {  static void Main(string[] args)  {  // Creating an instance of Author  Author author = new Author("The Great Wall", 1950, "Tushar", "Indian novelist and short story writer");   // Displaying book and author information  author.DisplayInfo();  } } |
| --- |

**Sample Problem 2: Smart Home Devices**

* + **Description**: Create a hierarchy for a smart home system where Device is the superclass and Thermostat is a subclass.
  + **Tasks**:
    - Define a superclass Device with attributes like DeviceId and Status.
    - Create a subclass Thermostat with additional attributes like TemperatureSetting.
    - Implement a method DisplayStatus() to show each device's current settings.
  + **Goal**: Understand single inheritance by adding specific attributes to a subclass, keeping the superclass general.

| using System; // Superclass Device public class Device {  public string DeviceId { get; set; }  public string Status { get; set; }   // Constructor  public Device(string deviceId, string status)  {  this.DeviceId = deviceId;  this.Status = status;  }   // Method to display device status  public virtual void DisplayStatus()  {  Console.WriteLine("Device ID: "+DeviceId+", Status: "+Status);  } }  // Subclass Thermostat public class Thermostat : Device {  public double TemperatureSetting { get; set; }   // Constructor  public Thermostat(string deviceId, string status, double temperatureSetting)  : base(deviceId, status)  {  this.TemperatureSetting = temperatureSetting;  }   // Overriding the DisplayStatus method  public override void DisplayStatus()  {  base.DisplayStatus();  Console.WriteLine("Temperature Setting: "+TemperatureSetting);  } }  // Main program class Program {  static void Main(string[] args)  {  // Creating an instance of Thermostat  Thermostat thermostat = new Thermostat("N12345", "Active", 25.5);   // Displaying thermostat status  thermostat.DisplayStatus();  } } |
| --- |

## Multilevel Inheritance

**Sample Problem 1: Online Retail Order Management**

* + **Description**: Create a multilevel hierarchy to manage orders, where Order is the base class, ShippedOrder is a subclass, and DeliveredOrder extends ShippedOrder.
  + **Tasks**:
    - Define a base class Order with common attributes like orderId and OrderDate.
    - Create a subclass ShippedOrder with additional attributes like TrackingNumber.
    - Create another subclass DeliveredOrder extending ShippedOrder, adding a DeliveryDate attribute.
    - Implement a method GetOrderStatus() to return the current order status based on the class level.
  + **Goal**: Explore multilevel inheritance, showing how attributes and methods can be added across a chain of classes.

| using System; public class Order {  public string OrderId { get; set; }  public DateTime OrderDate { get; set; }   // Constructor  public Order(string orderId, DateTime orderDate)  {  this.OrderId = orderId;  this.OrderDate = orderDate;  }   // Virtual method to get order status  public virtual string GetOrderStatus()  {  return "Order Placed";  }   // Method to display order details  public virtual void DisplayDetails()  {  Console.WriteLine("Order ID: "+OrderId+", Order Date: "+OrderDate);  } }  public class ShippedOrder : Order {  public string TrackingNumber { get; set; }   // Constructor  public ShippedOrder(string orderId, DateTime orderDate, string trackingNumber)  : base(orderId, orderDate)  {  this.TrackingNumber = trackingNumber;  }   // Overriding the GetOrderStatus method  public override string GetOrderStatus()  {  return "Order Shipped";  }   // Overriding the DisplayDetails method t  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Tracking Number: "+TrackingNumber);  } }  public class DeliveredOrder : ShippedOrder {  public DateTime DeliveryDate { get; set; }   // Constructor  public DeliveredOrder(string orderId, DateTime orderDate, string trackingNumber, DateTime deliveryDate)  : base(orderId, orderDate, trackingNumber)  {  this.DeliveryDate = deliveryDate;  }   // Overriding the GetOrderStatus method  public override string GetOrderStatus()  {  return "Order Delivered";  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Delivery Date: "+DeliveryDate);  } }  class Program {  static void Main(string[] args)  {  // Creating an instance of DeliveredOrder  DeliveredOrder deliveredOrder = new DeliveredOrder("N12345", DateTime.Now.AddDays(-5), "R987654", DateTime.Now);   // Displaying order details  deliveredOrder.DisplayDetails();   // Getting the order status  Console.WriteLine("Order Status: "+deliveredOrder.GetOrderStatus());  } } |
| --- |

**Sample Problem 2: Educational Course Hierarchy**

* + **Description**: Model a course system where Course is the base class, OnlineCourse is a subclass, and PaidOnlineCourse extends OnlineCourse.
  + **Tasks**:
    - Define a superclass Course with attributes like CourseName and Duration.
    - Define OnlineCourse to add attributes such as Platform and IsRecorded.
    - Define PaidOnlineCourse to add Fee and Discount.
  + **Goal**: Demonstrate how each level of inheritance builds on the previous, adding complexity to the system.

| using System; public class Course {  public string CourseName { get; set; }  public int Duration { get; set; }   // Constructor  public Course(string courseName, int duration)  {  this.CourseName = courseName;  this.Duration = duration;  }   // Method to display course information  public virtual void DisplayInfo()  {  Console.WriteLine("Course Name: "+CourseName+", Duration: "+Duration+" hours");  } }  public class OnlineCourse : Course {  public string Platform { get; set; }  public bool IsRecorded { get; set; }   // Constructor  public OnlineCourse(string courseName, int duration, string platform, bool isRecorded)  : base(courseName, duration)  {  this.Platform = platform;  this.IsRecorded = isRecorded;  }   // Overriding the DisplayInfo method  public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine("Platform: "+Platform+", Is Recorded: "+IsRecorded);  } }  public class PaidOnlineCourse : OnlineCourse {  public double Fee { get; set; }  public double Discount { get; set; }   // Constructor  public PaidOnlineCourse(string courseName, int duration, string platform, bool isRecorded, double fee, double discount)  : base(courseName, duration, platform, isRecorded)  {  this.Fee = fee;  this.Discount = discount;  }   // Overriding the DisplayInfo method  public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine("Fee: "+Fee+", Discount: "+Discount+"%");  } }  class Program {  static void Main(string[] args)  {  // Creating an instance of PaidOnlineCourse  PaidOnlineCourse paidCourse = new PaidOnlineCourse("Basic C#", 50, "BridgeLabz", true, 499.99, 10);   // Displaying course details  paidCourse.DisplayInfo();  } } |
| --- |

## Hierarchical Inheritance

**Sample Problem 1: Bank Account Types**

* + **Description**: Model a banking system with different account types using hierarchical inheritance. BankAccount is the superclass, with SavingsAccount, CheckingAccount, and FixedDepositAccount as subclasses.
  + **Tasks**:
    - Define a base class BankAccount with attributes like AccountNumber and Balance.
    - Define subclasses SavingsAccount, CheckingAccount, and FixedDepositAccount, each with unique attributes like interestRate for SavingsAccount and WithdrawalLimit for CheckingAccount.
    - Implement a method DisplayAccountType() in each subclass to specify the account type.
  + **Goal**: Explore hierarchical inheritance, demonstrating how each subclass can have unique attributes while inheriting from a shared superclass.

| using System; // Base class BankAccount public class BankAccount {  public string AccountNumber { get; set; }  public double Balance { get; set; }   // Constructor  public BankAccount(string accountNumber, double balance)  {  this.AccountNumber = accountNumber;  this.Balance = balance;  }   // Virtual method to display account type  public virtual void DisplayAccountType()  {  Console.WriteLine("Bank Account");  }   // Method to display account details  public virtual void DisplayDetails()  {  Console.WriteLine("Account Number: "+AccountNumber+", Balance: "+Balance);  } }  // Subclass SavingsAccount public class SavingsAccount : BankAccount {  public double InterestRate { get; set; }   // Constructor  public SavingsAccount(string accountNumber, double balance, double interestRate)  : base(accountNumber, balance)  {  this.InterestRate = interestRate;  }   // Overriding the DisplayAccountType method  public override void DisplayAccountType()  {  Console.WriteLine("Savings Account");  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Interest Rate: "+InterestRate+"%");  } }  // Subclass CheckingAccount public class CheckingAccount : BankAccount {  public double WithdrawalLimit { get; set; }   // Constructor  public CheckingAccount(string accountNumber, double balance, double withdrawalLimit)  : base(accountNumber, balance)  {  this.WithdrawalLimit = withdrawalLimit;  }   // Overriding the DisplayAccountType method  public override void DisplayAccountType()  {  Console.WriteLine("Checking Account");  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Withdrawal Limit: "+WithdrawalLimit);  } }  // Subclass FixedDepositAccount public class FixedDepositAccount : BankAccount {  public int TermLength { get; set; }   // Constructor  public FixedDepositAccount(string accountNumber, double balance, int termLength)  : base(accountNumber, balance)  {  this.TermLength = termLength;  }   // Overriding the DisplayAccountType method  public override void DisplayAccountType()  {  Console.WriteLine("Fixed Deposit Account");  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Term Length: "+TermLength+" months");  } }  // Main program class Program {  static void Main(string[] args)  {  // Creating instances of different account types  BankAccount savingsAccount = new SavingsAccount("SA12345", 50000.00, 3.5);  BankAccount checkingAccount = new CheckingAccount("CA12345", 30000.00, 1000.00);  BankAccount fixedDepositAccount = new FixedDepositAccount("FD12345", 100000.00, 10);   // Storing the objects in an array of BankAccount type  BankAccount[] accounts = { savingsAccount, checkingAccount, fixedDepositAccount };   // Displaying account type and details  foreach (BankAccount account in accounts)  {  account.DisplayAccountType();  account.DisplayDetails();  Console.WriteLine();  }  } } |
| --- |

**Sample Problem 2: School System with Different Roles**

* + **Description**: Create a hierarchy for a school system where Person is the superclass, and Teacher, Student, and Staff are subclasses.
  + **Tasks**:
    - Define a superclass Person with common attributes like Name and Age.
    - Define subclasses Teacher, Student, and Staff with specific attributes (e.g., Subject for Teacher and Grade for Student).
    - Each subclass should have a method like DisplayRole() that describes the role.
  + **Goal**: Demonstrate hierarchical inheritance by modeling different roles in a school, each with shared and unique characteristics.

| using System; // Superclass Person public class Person {  public string Name { get; set; }  public int Age { get; set; }   // Constructor  public Person(string name, int age)  {  this.Name = name;  this.Age = age;  }   // Virtual method to display the role  public virtual void DisplayRole()  {  Console.WriteLine("Person");  }   // Method to display person details  public virtual void DisplayDetails()  {  Console.WriteLine("Name: "+Name+", Age: "+Age);  } }  // Subclass Teacher public class Teacher : Person {  public string Subject { get; set; }   // Constructor  public Teacher(string name, int age, string subject)  : base(name, age)  {  this.Subject = subject;  }   // Overriding the DisplayRole method  public override void DisplayRole()  {  Console.WriteLine("Teacher");  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Subject: "+Subject);  } }  // Subclass Student public class Student : Person {  public string Grade { get; set; }   // Constructor  public Student(string name, int age, string grade)  : base(name, age)  {  this.Grade = grade;  }   // Overriding the DisplayRole method  public override void DisplayRole()  {  Console.WriteLine("Student");  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Grade: "+Grade);  } }  // Subclass Staff public class Staff : Person {  public string Position { get; set; }   // Constructor  public Staff(string name, int age, string position)  : base(name, age)  {  this.Position = position;  }   // Overriding the DisplayRole method  public override void DisplayRole()  {  Console.WriteLine("Staff");  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Position: "+Position);  } }  // Main program class Program {  static void Main(string[] args)  {  // Creating instances of different roles  Person teacher = new Teacher("Annu", 35, "Mathematics");  Person student = new Student("Bobby", 15, "10th Grade");  Person staff = new Staff("Tulshi", 40, "Administrator");   // Storing the objects in an array of Person type  Person[] people = { teacher, student, staff };   // Displaying role and details  foreach (Person person in people)  {  person.DisplayRole();  person.DisplayDetails();  Console.WriteLine();  }  } } |
| --- |

## Hybrid Inheritance (Simulating Multiple Inheritance)

Since Java doesn’t support multiple inheritance directly, hybrid inheritance is typically achieved through **interfaces**.

**Sample Problem 1: Restaurant Management System with Hybrid Inheritance**

* + **Description**: Model a restaurant system where Person is the superclass and Chef and Waiter are subclasses. Both Chef and Waiter should implement a Worker interface that requires a PerformDuties() method.
  + **Tasks**:
    - Define a superclass Person with attributes like Name and Id.
    - Create an interface Worker with a method PerformDuties().
    - Define subclasses Chef and Waiter that inherit from Person and implement the Worker interface, each providing a unique implementation of PerformDuties().
  + **Goal**: Practice hybrid inheritance by combining inheritance and interfaces, giving multiple behaviors to the same objects.

| using System; // Superclass Person public class Person {  public string Name { get; set; }  public int Id { get; set; }   // Constructor  public Person(string name, int id)  {  this.Name = name;  this.Id = id;  }   // Method to display person details  public virtual void DisplayDetails()  {  Console.WriteLine("Name: "+Name+", ID: "+Id);  } }  // Interface Worker public interface Worker {  void PerformDuties(); }  // Subclass Chef public class Chef : Person, Worker {  // Constructor  public Chef(string name, int id) : base(name, id) { }   // Implementing PerformDuties method for Chef  public void PerformDuties()  {  Console.WriteLine("Chef is preparing meals.");  }   // Overriding DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Role: Chef");  } }  // Subclass Waiter public class Waiter : Person, Worker {  // Constructor  public Waiter(string name, int id) : base(name, id) { }   // Implementing PerformDuties method  public void PerformDuties()  {  Console.WriteLine("Waiter is serving customers.");  }   // Overriding DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Role: Waiter");  } }  // Main program class Program {  static void Main(string[] args)  {  // Creating instances of Chef and Waiter  Chef chef = new Chef("Mohit", 101);  Waiter waiter = new Waiter("Bobby", 102);   // Displaying details and performing duties  chef.DisplayDetails();  chef.PerformDuties();  Console.WriteLine();   waiter.DisplayDetails();  waiter.PerformDuties();  } } |
| --- |

**Sample Problem 2: Vehicle Management System with Hybrid Inheritance**

* + **Description**: Model a vehicle system where Vehicle is the superclass and ElectricVehicle and PetrolVehicle are subclasses. Additionally, create a Refuelable interface implemented by PetrolVehicle.
  + **Tasks**:
    - Define a superclass Vehicle with attributes like MaxSpeed and Model.
    - Create an interface Refuelable with a method Refuel().
    - Define subclasses ElectricVehicle and PetrolVehicle. PetrolVehicle should implement Refuelable, while ElectricVehicle include a Charge() method.
  + **Goal**: Use hybrid inheritance by having PetrolVehicle implement both Vehicle and Refuelable, demonstrating how Java interfaces allow adding multiple behaviors.

| using System; // Superclass Vehicle public class Vehicle {  public int MaxSpeed { get; set; }  public string Model { get; set; }   // Constructor  public Vehicle(int maxSpeed, string model)  {  this.MaxSpeed = maxSpeed;  this.Model = model;  }   // Method to display vehicle details  public virtual void DisplayDetails()  {  Console.WriteLine("Model: "+Model+", Max Speed: "+MaxSpeed+" km/h");  } }  // Interface Refuelable public interface Refuelable {  void Refuel();  // Subclass ElectricVehicle public class ElectricVehicle : Vehicle {  public ElectricVehicle(int maxSpeed, string model)  : base(maxSpeed, model) { }   // Method to charge the electric vehicle  public void Charge()  {  Console.WriteLine("Charging electric vehicle...");  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Vehicle Type: Electric");  } }  // Subclass PetrolVehicle public class PetrolVehicle : Vehicle, Refuelable {  // Constructor  public PetrolVehicle(int maxSpeed, string model)  : base(maxSpeed, model) { }   // Implementing the Refuel method  public void Refuel()  {  Console.WriteLine("Refueling petrol vehicle...");  }   // Overriding the DisplayDetails method  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine("Vehicle Type: Petrol");  } }  // Main program class Program {  static void Main(string[] args)  {  // Creating instances of ElectricVehicle and PetrolVehicle  ElectricVehicle electricVehicle = new ElectricVehicle(140, "Tesla Model S");  PetrolVehicle petrolVehicle = new PetrolVehicle(160, "Toyota Corolla");   // Displaying details and performing specific actions  electricVehicle.DisplayDetails();  electricVehicle.Charge();  Console.WriteLine();   petrolVehicle.DisplayDetails();  petrolVehicle.Refuel();  } } |
| --- |

**Best Practices for Inheritance in C#**

1. **Favor Composition Over Inheritance:**
   * Prefer **composition** over inheritance when a class can be described as "has-a" instead of "is-a."
   * Composition allows you to create more flexible, loosely coupled designs, providing better maintainability and easier future changes.
2. **Ensure Proper Use of is-a Relationship:**
   * Use inheritance only when the subclass **truly** extends the behavior of the superclass, maintaining the "is-a" relationship.
   * Avoid misusing inheritance just for code reuse; inheritance should represent a logical relationship, not a mere organizational convenience.
3. **Follow the Liskov Substitution Principle (LSP):**
   * Ensure that subclasses can be substituted for their base class without altering the correct behavior of the program.
   * Overridden methods should preserve the behavior of the superclass or provide behavior that is a logical extension.
4. **Avoid Deep Inheritance Hierarchies:**
   * Keep the inheritance hierarchy **shallow** to reduce complexity and improve maintainability.
   * Deep inheritance hierarchies can become difficult to manage, making code harder to debug and understand.
5. **Mark Superclass Methods as final If Needed:**
   * If you want to prevent subclasses from overriding critical methods, mark them as sealed in C#.
   * This ensures that essential functionality remains unchanged and provides better control over inheritance.
6. **Use override Keyword:**
   * Always use the override keyword to indicate that a method is being overridden in a subclass.
   * This makes the code more readable and helps the compiler catch errors when the method signature is incorrect.
7. **Minimize Public Fields in Superclasses:**
   * Avoid public fields; use **private** or **protected** fields with proper **getters** and **setters** to ensure encapsulation.
   * This prevents subclasses from directly modifying the base class fields, which can lead to unexpected behavior.
8. **Avoid Overloading Alongside Overriding:**
   * Overloading methods with similar names and parameters in subclasses can cause confusion. Clearly separate overloaded methods from overridden methods.
   * Ensure that method names and parameters are distinct to avoid ambiguity.
9. **Prefer Abstract Classes for Partial Implementation:**
   * Use **abstract classes** to provide a common blueprint with partial implementation for related subclasses.
   * Abstract classes can define abstract methods that subclasses must implement, while still providing shared behavior.
10. **Use Interfaces for Multiple Inheritance:**
    * C# does not support multiple inheritance through classes, but **interfaces** can be used to achieve multiple inheritance-like behavior.
    * This helps avoid the "diamond problem" while still allowing classes to implement multiple behaviors.
11. **Document Inheritance Behavior:**
    * Always clearly document the expected behavior of both the superclass and the subclass.
    * Describe how methods are intended to be overridden or extended by subclasses and specify any restrictions on their use.
12. **Avoid Overriding Methods Unnecessarily:**
    * Override methods only when it is necessary to change or extend the behavior of the superclass.
    * Overriding for the sake of overriding can lead to unnecessary complexity and reduce code clarity.
13. **Be Cautious with Constructors:**
    * In C#, use the base() keyword to call the superclass constructor explicitly if needed.
    * Avoid calling non-final methods in constructors, as subclasses may not yet be fully initialized, leading to unpredictable behavior.
14. **Use Polymorphism Effectively:**
    * Leverage polymorphism where possible, allowing objects of different subclasses to be treated as instances of the superclass.
    * This provides flexibility and extensibility in your application, enabling cleaner and more maintainable code.
15. **Beware of Fragile Base Class Problem:**
    * Changing the base class can unintentionally affect all subclasses, leading to bugs.
    * Minimize dependencies on base class behavior and be cautious when making changes to it, especially in a large codebase.
16. **Test Subclass and Superclass Interactions:**
    * Thoroughly test interactions between subclasses and their base classes to ensure that subclass modifications do not break expected behavior.
    * Use unit tests and integration tests to check that methods behave as expected across inheritance hierarchies.
17. **Avoid Inheriting from Concrete Classes:**
    * Prefer inheriting from **abstract classes** or **interfaces** rather than concrete classes to reduce tight coupling with specific implementations.
    * This allows for greater flexibility and easier modification of the system.
18. **Consider Using Delegation for Special Cases:**
    * For specific behavior needed in a few instances, consider **delegation** instead of inheritance.
    * Delegation can be a better choice when you need to maintain flexibility, avoid unnecessary complexity, and improve separation of concerns.