# 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;  class Animal {  private string name;  private string age;   // Virtual method to be overridden  public virtual void MakeSound()  {  Console.WriteLine("Animal makes Sound");  } }  class Dog : Animal {  public override void MakeSound()  {  Console.WriteLine("Dog Barks");  } }  class Cat : Animal {  public override void MakeSound()  {  Console.WriteLine("Cat meows");  } }  class Bird : Animal {  public override void MakeSound()  {  Console.WriteLine("Bird chirps");  } }  class Program {  public static void Main()  {  Animal animal = new Animal();  animal.MakeSound();   // Polymorphism: Base class reference holding derived class objects  Animal animal1 = new Dog();  animal1.MakeSound();   Animal animal2 = new Cat();  animal2.MakeSound();   Animal animal3 = new Bird();  animal3.MakeSound();   Console.ReadLine();  } } |
| --- |

**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 representing a general employee class Employee {  private string name;  private int id;  private double salary;   public Employee(string name, int id, double salary)  {  this.name = name;  this.id = id;  this.salary = salary;  }   // Virtual method to be overridden by subclasses  public virtual void DisplayDetails()  {  Console.WriteLine($"Employee Name: {name}");  Console.WriteLine($"Employee Id: {id}");  Console.WriteLine($"Employee Salary: {salary}");  } }  // Derived class for Manager class Manager : Employee {  private int teamSize;   public Manager(int teamSize, string name, int id, double salary) : base(name, id, salary)  {  this.teamSize = teamSize;  }   // Overriding DisplayDetails to include TeamSize  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine($"Team Size: {teamSize}");  } }  // Derived class for Developer class Developer : Employee {  private string programmingLanguage;   public Developer(string programmingLanguage, string name, int id, double salary) : base(name, id, salary)  {  this.programmingLanguage = programmingLanguage;  }   // Overriding DisplayDetails to include ProgrammingLanguage  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine($"Programming Language: {programmingLanguage}");  } }  // Derived class for Intern class Intern : Employee {  private string intershipDuration;   public Intern(string intershipDuration, string name, int id, double salary) : base(name, id, salary)  {  this.intershipDuration = intershipDuration;  }   // Overriding DisplayDetails to include InternshipDuration  public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine($"Internship Duration: {intershipDuration}");  } }  class Program {  public static void Main()  {  // Reading inputs for Employee details  Console.WriteLine("Enter Employee Name:");  string name = Console.ReadLine();  Console.WriteLine("Enter Employee ID:");  int id = Convert.ToInt32(Console.ReadLine());  Console.WriteLine("Enter Employee Salary:");  double salary = Convert.ToDouble(Console.ReadLine());  Console.WriteLine("Enter Manager Team Size:");  int teamSize = Convert.ToInt32(Console.ReadLine());  Console.WriteLine("Enter Developer Programming Language:");  string programmingLanguage = Console.ReadLine();  Console.WriteLine("Enter Intern Duration:");  string internshipDuration = Console.ReadLine();   // Creating an Employee object  Employee employee = new Employee(name, id, salary);  employee.DisplayDetails();   // Creating instances of subclasses with correct parameters  Employee employee1 = new Manager(teamSize, name, id, salary);  employee1.DisplayDetails();   Employee employee2 = new Developer(programmingLanguage, name, id, salary);  employee2.DisplayDetails();   Employee employee3 = new Intern(internshipDuration, name, id, salary);  employee3.DisplayDetails();   Console.ReadLine();  } } |
| --- |

**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; class Vehicle {  private int maxSpeed;  private string fuelType;   public Vehicle(int maxSpeed, string fuelType)  {  this.maxSpeed = maxSpeed;  this.fuelType = fuelType;  }   public virtual void DisplayInfo()  {  Console.WriteLine($"Maximum Speed: {maxSpeed}");  Console.WriteLine($"Fuel Type: {fuelType}");  } }  class Car : Vehicle {  private int seatCapacity;  public Car(int maxSpeed, string fuelType, int seatCapacity) : base(maxSpeed, fuelType)  {  this.seatCapacity = seatCapacity;  }   public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine($"Seat Capacity: {seatCapacity}");  } }  class Truck : Vehicle {  private int payLoadCapacity;  public Truck(int maxSpeed, string fuelType, int payLoadCapacity) : base(maxSpeed, fuelType)  {  this.payLoadCapacity = payLoadCapacity;  }   public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine($"Seat Capacity: {payLoadCapacity}");  } }  class MotorCycle : Vehicle {  private bool hasSideCar;  public MotorCycle(int maxSpeed, string fuelType, bool hasSideCar) : base(maxSpeed, fuelType)  {  this.hasSideCar = hasSideCar;  }   public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine($"Seat Capacity: {hasSideCar}");  } }  class Program {  public static void Main()  {   Vehicle vehicle = new Vehicle(50, "Petrol");  vehicle.DisplayInfo();   Vehicle vehicle1 = new Car(50, "Petrol", 5);  vehicle1.DisplayInfo();   Vehicle vehicle2 = new Truck(50, "Petrol", 200);  vehicle2.DisplayInfo();   Vehicle vehicle3 = new MotorCycle(50, "Petrol", true);  vehicle3.DisplayInfo();   Console.ReadLine();  } } |
| --- |

## 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; class Book {  private string title;  private int publicationYear;  public Book(string title, int publicationYear)  {  this.title = title;  this.publicationYear = publicationYear;  }  public virtual void DisplayInfo()  {  Console.WriteLine($"Title:{title} \nPublication Year:{publicationYear}");  } } class Author : Book {  string name, bio;  public Author(string title, int publicationYear, string name, string bio) : base(title, publicationYear)  {  this.name = name;  this.bio = bio;  }  public override void DisplayInfo()  {  base.DisplayInfo();  Console.WriteLine($"Author Name: {name} \nAuthor Bio: {bio}");  } } class HelloWorld {  static void Main()  {  Book mark = new Author("Julius Caesar", 1950, "Mark Twain", "Social writer");  mark.DisplayInfo();  Console.ReadLine();  } } |
| --- |

**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;  class Device {  private int deviceId;  private string status;   public Device(int deviceId, string status)  {  this.deviceId = deviceId;  this.status = status;  }   public int GetDeviceId()  {  return deviceId;  }   public string GetDeviceStatus()  {  return status;  } }  class Thermostat : Device {  private string tempSetting;   public Thermostat(int deviceId, string status, string tempSetting) : base(deviceId, status)  {  this.tempSetting = tempSetting;  }   public void DisplayStatus()  {  Console.WriteLine($"Device Id: {GetDeviceId()}");  Console.WriteLine($"Device Status: {GetDeviceStatus()}");  Console.WriteLine($"Temperature Setting: {tempSetting}");  } }  class Program {  public static void Main()  {  Thermostat thermostat = new Thermostat(1, "On", "23");  thermostat.DisplayStatus();  Console.ReadLine();  } } |
| --- |

## 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; class Order {  private string orderId, orderDate;   public Order(string orderId, string orderDate)  {  this.orderId = orderId;  this.orderDate = orderDate;  }   public virtual void GetOrderStatus()  {  Console.WriteLine($"Order ID: {orderId} \nOrder Date: {orderDate}");  } }  class ShippedOrder : Order {  private string trackingNumber;   public ShippedOrder(string orderId, string orderDate, string trackingNumber)  : base(orderId, orderDate)  {  this.trackingNumber = trackingNumber;  }   public override void GetOrderStatus()  {  base.GetOrderStatus();  Console.WriteLine($"Tracking Number: {trackingNumber}");  } }  class DeliveredOrder : ShippedOrder {  private string deliveryDate;   public DeliveredOrder(string orderId, string orderDate, string trackingNumber, string deliveryDate)  : base(orderId, orderDate, trackingNumber)  {  this.deliveryDate = deliveryDate;  }   public override void GetOrderStatus()  {  base.GetOrderStatus();  Console.WriteLine($"Delivery Date: {deliveryDate}");  } }  class HelloWorld {  static void Main()  {  Order order = new Order("ORD123", "2025-02-08");  order.GetOrderStatus();   Console.WriteLine("\n---");   ShippedOrder shippedOrder = new ShippedOrder("ORD124", "2025-02-07", "TRK987654");  shippedOrder.GetOrderStatus();   Console.WriteLine("\n---");   DeliveredOrder deliveredOrder = new DeliveredOrder("ORD125", "2025-02-06", "TRK654321", "2025-02-08");  deliveredOrder.GetOrderStatus();   Console.ReadLine();  } } |
| --- |

**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; class Course {  private string courseName;  private string duration;   public Course(string courseName, string duration)  {  this.courseName = courseName;  this.duration = duration;  }   public virtual void DisplayDetails()  {  Console.WriteLine($"Course Name: {courseName}");  Console.WriteLine($"Course Duration: {duration}");  } }  class OnlineCourse : Course {  string platform;  bool isRecorded;   public OnlineCourse(string courseName, string duration, string platform, bool isRecorded) : base(courseName, duration)  {  this.platform = platform;  this.isRecorded = isRecorded;  }   public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine($"Platform: {platform}");  Console.WriteLine($"Is Recorded: {isRecorded}");  } }  class PaidOnlineCourse : OnlineCourse {  int fee;  int discount;   public PaidOnlineCourse(string courseName, string duration, string platform, bool isRecorded, int fee, int discount) : base(courseName,duration,platform,isRecorded)  {  this.fee = fee;  this.discount = discount;  }   public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine($"Course Fee: {fee}");  Console.WriteLine($"Discount: {discount}");  } }  class Program {  public static void Main()  {  Course course = new Course("MCA", "2 Years");  course.DisplayDetails();  Console.WriteLine("-------------------------------");  OnlineCourse onlineCourse = new OnlineCourse("BCA", "3 Years", "Online", true);  onlineCourse.DisplayDetails();  Console.WriteLine("-------------------------------");  PaidOnlineCourse paidOnlineCourse = new PaidOnlineCourse("Btech", "4 Yaers", "OnCampus", true, 2300, 1000);  paidOnlineCourse.DisplayDetails();  Console.ReadLine();  } } |
| --- |

## 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;  class BankAccount {  protected string accountNumber;  protected double balance;   public BankAccount(string accountNumber, double balance)  {  this.accountNumber = accountNumber;  this.balance = balance;  }   public virtual void DisplayAccountType()  {  Console.WriteLine("General Bank Account");  } }  class SavingsAccount : BankAccount {  private double interestRate;   public SavingsAccount(string accountNumber, double balance, double interestRate)  : base(accountNumber, balance)  {  this.interestRate = interestRate;  }   public override void DisplayAccountType()  {  Console.WriteLine("This is a Savings Account.");  } }  class CheckingAccount : BankAccount {  private double withdrawalLimit;   public CheckingAccount(string accountNumber, double balance, double withdrawalLimit)  : base(accountNumber, balance)  {  this.withdrawalLimit = withdrawalLimit;  }   public override void DisplayAccountType()  {  Console.WriteLine("This is a Checking Account.");  } }  class FixedDepositAccount : BankAccount {  private int duration;   public FixedDepositAccount(string accountNumber, double balance, int duration)  : base(accountNumber, balance)  {  this.duration = duration;  }   public override void DisplayAccountType()  {  Console.WriteLine("This is a Fixed Deposit Account.");  } }  class HelloWorld {  static void Main()  {  SavingsAccount savings = new SavingsAccount("SAV123", 5000, 5);  CheckingAccount checking = new CheckingAccount("CHK456", 3000, 1000);  FixedDepositAccount fixedDeposit = new FixedDepositAccount("FD789", 10000, 12);   savings.DisplayAccountType();  checking.DisplayAccountType();  fixedDeposit.DisplayAccountType();  Console.ReadLine();  } } |
| --- |

**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; class Person {  string name;  int age;   public Person(string name, int age)  {  this.name = name;  this.age = age;  }   public virtual void DisplayDetails()  {  Console.WriteLine($"Name: {name}");  Console.WriteLine($"Age: {age}");  }   public virtual void DisplayRole()  {  Console.WriteLine("I'm a Person");  } }  class Teacher : Person {  string subject;   public Teacher(string name, int age, string subject) : base(name, age)  {  this.subject = subject;  }   public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine($"Subject: {subject}");  }  public override void DisplayRole()  {  Console.WriteLine("I'm a Person, Whoose Role is Teacher");  } }  class Student : Person {  string grade;   public Student(string name, int age, string grade) : base(name, age)  {  this.grade = grade;  }   public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine($"Your Grade is : {grade}");  }  public override void DisplayRole()  {  Console.WriteLine("I'm a Person, Whoose Role is Student");  } }  class Staff : Person {  string task;   public Staff(string name, int age, string task) : base(name, age)  {  this.task = task;  }   public override void DisplayDetails()  {  base.DisplayDetails();  Console.WriteLine($"Your Task is : {task}");  }  public override void DisplayRole()  {  Console.WriteLine("I'm a Person, Whoose Role is Staff");  } }  class Program {  public static void Main()  {  Person person = new Person("Aarush", 25);  person.DisplayDetails();  person.DisplayRole();  Console.WriteLine("-----------------------");   Student student = new Student("Aarush", 25, "A");  student.DisplayDetails();  student.DisplayRole();  Console.WriteLine("-----------------------");   Teacher teacher = new Teacher("Aarush", 25, "Maths");  teacher.DisplayDetails();  teacher.DisplayRole();  Console.WriteLine("-----------------------");   Staff staff = new Staff("Aarush", 25, "Bring File");  staff.DisplayDetails();  staff.DisplayRole();  Console.WriteLine("-----------------------");   Console.ReadLine();  } } |
| --- |

## 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;  interface Worker {  void PerformDuties(); }  class Person {  string name;  int id;  public Person(string name, int id)  {  this.name = name;  this.id = id;  } }  class Chef : Person, Worker {  public Chef(string name,int id) : base(name,id) { }  public void PerformDuties()  {  Console.WriteLine("I am chef and i cook food");  } }  class Waiter : Person, Worker {  public Waiter(string name, int id) : base(name, id) { }  public void PerformDuties()  {  Console.WriteLine("I am Waiter and i serve food");  } }  class Program {  static void Main()  {  Chef c = new Chef("Rohan", 123);  c.PerformDuties();   Waiter w = new Waiter("Ramu", 111);  w.PerformDuties();  Console.ReadLine();  } } |
| --- |

**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;  // Abstract class Vehicle abstract class Vehicle {  protected int maxSpeed;  protected string model;   public Vehicle(int maxSpeed, string model)  {  this.maxSpeed = maxSpeed;  this.model = model;  }   public void DisplayInfo()  {  Console.WriteLine("Model: " + model + ", Max Speed: " + maxSpeed + " km/h");  } }  // Interface for refuelable vehicles interface IRefuelable {  void Refuel(); }  // Derived class for Electric Vehicles class ElectricVehicle : Vehicle {  private int batteryCapacity; // in kWh   public ElectricVehicle(int maxSpeed, string model, int batteryCapacity)  : base(maxSpeed, model)  {  this.batteryCapacity = batteryCapacity;  }   public void Charge()  {  Console.WriteLine(model + " is charging. Battery capacity: " + batteryCapacity + " kWh");  } }  // Derived class for Petrol Vehicles implementing IRefuelable class PetrolVehicle : Vehicle, IRefuelable {  private int fuelCapacity; // in liters   public PetrolVehicle(int maxSpeed, string model, int fuelCapacity)  : base(maxSpeed, model)  {  this.fuelCapacity = fuelCapacity;  }   public void Refuel()  {  Console.WriteLine(model + " is refueling. Fuel capacity: " + fuelCapacity + " liters");  } }  // Main Program class Program {  static void Main()  {  ElectricVehicle tesla = new ElectricVehicle(200, "Tesla Model S", 100);  PetrolVehicle honda = new PetrolVehicle(180, "Honda Civic", 50);   tesla.DisplayInfo();  tesla.Charge();   Console.WriteLine();   honda.DisplayInfo();  honda.Refuel();   Console.ReadLine();  } } |
| --- |

**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.