Lecture 04 - Practice Tasks

Constructors, Inheritance, Interfaces & Properties

Created by: ITI

Tasks Overview

#	Task Name	Topic
1	Date Class with Constructors	Constructors & Initializer Lists
2	Counter with Static Constructor	Static Constructor
3	Employee Inheritance System	Inheritance & base keyword
4	Shape with Method Overriding	virtual & override
5	Animal Abstract Class	Abstract Classes
6	IMovable Interface	Interfaces
7	Student with Properties	Properties (get/set)
8	Bank System (Full OOP)	All Concepts Combined

Task 1: Date Class with Constructors

Description

Create a Date class with multiple constructors using constructor overloading and initializer lists.

The initializer list (`: this()`) allows one constructor to call another constructor in the same class.

Example

```
Date d1 = new Date();              // Default: 01/01/1990
Date d2 = new Date(2024);          // 01/01/2024
Date d3 = new Date(2024, 6);       // 01/06/2024
Date d4 = new Date(2024, 6, 15);   // 15/06/2024
```

Illustration

```
CONSTRUCTOR OVERLOADING:
Multiple constructors with different parameters.


INITIALIZER LIST:
One constructor calling another using ': this()'


  ┌────────────────────────────────────────────────────────┐
  │  public Date() : this(1990, 1, 1) { }                  │
  │                  ↑                                     │
  │      Calls the 3-parameter constructor                 │
  └────────────────────────────────────────────────────────┘


CONSTRUCTOR CHAIN:

  new Date()
      │
      │  : this(1990, 1, 1)
      ↓
  Date(int year, int month, int day)  ← Main constructor
      │
      │  Sets: year = 1990
      │        month = 1
      │        day = 1
      ↓
  Object Created!


CONSTRUCTOR RULES:
  1. Same name as the class
  2. No return type (not even void)
  3. Must be public (usually)
  4. Can be overloaded (multiple versions)
```

Task 2: Counter with Static Constructor

Description

Create a Counter class with a static constructor that initializes class-level data, and instance constructors for object-level data.

Demonstrate the difference between static and instance members.

Example

```
// Before any objects are created:
// Static constructor runs automatically!

Counter c1 = new Counter();  // Instance constructor runs
Counter c2 = new Counter();  // Instance constructor runs
Counter c3 = new Counter();  // Instance constructor runs

// Static member (shared by ALL objects):
Counter.totalObjectsCreated  →  3

// Instance members (unique to EACH object):
c1.instanceId  →  1
c2.instanceId  →  2
c3.instanceId  →  3
```

Illustration

```
EXECUTION ORDER:

  ┌──────────────────────────────────────────┐
  │       STATIC CONSTRUCTOR                 │
  │  • Runs ONCE automatically               │
  │  • Before first object is created        │
  │  • No access modifier allowed            │
  │  • Cannot be called manually            │
  └─────────────────────────────────────────┘
                    ↓ (runs first)
  ┌─────────────────────────────────────────┐
  │      INSTANCE CONSTRUCTOR               │
  │  • Runs EACH time 'new' is called       │
  │  • Can have access modifiers            │
  │  • Can be overloaded                    │
  └─────────────────────────────────────────┘


STATIC vs INSTANCE:

  ┌──────────────────────┬──────────────────────┐
  │   STATIC MEMBERS     │   INSTANCE MEMBERS   │
  ├──────────────────────┼──────────────────────┤
  │ Belong to CLASS      │ Belong to OBJECT     │
  │ ONE copy in memory   │ MANY copies          │
  │ Shared by all        │ Unique to each       │
  │ Class.Member         │ object.Member        │
  └──────────────────────┴──────────────────────┘


MEMORY VIEW:

  ┌───────────────────────────────┐
  │     Counter CLASS             │
  │  static totalObjectsCreated=3 │  ← ONE copy
  └───────────────────────────────┘
           │
     ┌─────┼─────┐
     ↓     ↓     ↓
   ┌───┐ ┌───┐ ┌───┐
   │c1 │ │c2 │ │c3 │  ← Each has own instanceId
   │id1│ │id2│ │id3│
   └───┘ └───┘ └───┘
```

Task 3: Employee Inheritance System

Description

Create an Employee base class and derived classes (Manager, Developer, Intern).

Demonstrate inheritance, the `base` keyword, and constructor chaining between parent and child.

Example

```
class Employee { ... }                  // Base class
class Manager : Employee { ... }        // Child class

Manager mgr = new Manager(101, "Ahmed", 8000, 2000);

// Constructor order:
// 1. Employee constructor runs FIRST
// 2. Manager constructor runs SECOND

mgr.DisplayInfo();       // Inherited from Employee
mgr.DisplayManagerInfo(); // Manager's own method
```

Illustration

```
INHERITANCE HIERARCHY:

                 ┌────────────────────┐
                 │      Employee      │  ← Base class (Parent)
                 │                    │
                 │  - id              │
                 │  - name            │
                 │  - baseSalary      │
                 │                    │
                 │  + DisplayInfo()   │
                 │  + CalculateSalary()│
                 └─────────┬──────────┘
                           │ inherits
          ┌────────────────┼────────────────┐
          ↓                ↓                ↓
   ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
   │   Manager    │ │  Developer   │ │    Intern    │
   │              │ │              │ │              │
   │  - bonus     │ │  - language  │ │  - university│
   │  - teamSize  │ │  - projects  │ │  - duration  │
   └──────────────┘ └──────────────┘ └──────────────┘


CONSTRUCTOR CHAIN:

  new Manager(id, name, salary, bonus, teamSize)
      │
      │  : base(id, name, salary)    ← Calls parent!
      ↓
  Employee(id, name, salary)         ← Parent runs FIRST
      │
      ↓
  Manager constructor body           ← Child runs SECOND


ACCESS MODIFIERS:

  public:     Accessible from anywhere
  protected:  Accessible in class AND derived classes
  private:    Accessible ONLY in same class


  In Employee:
  ┌────────────────────────────────────────────┐
  │ protected int id;      → Manager CAN access│
  │ protected string name;→ Developer CAN      │
  │ private string ssn;    → ONLY Employee     │
  └────────────────────────────────────────────┘
```

Task 4: Shape with Method Overriding

Description

Create a Shape base class with virtual methods (CalculateArea, CalculatePerimeter).

Create derived classes (Circle, Rectangle, Triangle) that override these methods.

Example

```
class Shape {
    public virtual double CalculateArea() {
        return 0;  // Default implementation
    }
}

class Circle : Shape {
    public override double CalculateArea() {
        return Math.PI * radius * radius;
    }
}

Shape s = new Circle(5);
s.CalculateArea();  // Returns 78.54 (Circle's method!)
```

Illustration

```
VIRTUAL AND OVERRIDE:

  BASE CLASS (Shape):
  ┌─────────────────────────────────────────────────┐
  │ public virtual double CalculateArea()           │
  │ {                                               │
  │     return 0;   // Default                      │
  │ }                                               │
  │         ↑                                       │
  │    'virtual' = CAN be overridden                │
  └─────────────────────────────────────────────────┘

  DERIVED CLASS (Circle):
  ┌─────────────────────────────────────────────────┐
  │ public override double CalculateArea()          │
  │ {                                               │
  │     return Math.PI * radius * radius;           │
  │ }                                               │
  │         ↑                                       │
  │    'override' = REPLACES base method            │
  └─────────────────────────────────────────────────┘


POLYMORPHISM:
  Shape[] shapes = new Shape[3];
  shapes[0] = new Circle(5);
  shapes[1] = new Rectangle(4, 6);
  shapes[2] = new Triangle(3, 4, 5);

  foreach (Shape s in shapes) {
      s.CalculateArea();  // Correct override called!
  }

  ┌─────────┐     ┌─────────────┐     ┌──────────┐
  │ Circle  │     │  Rectangle  │     │ Triangle │
  │ π × r²  │     │  w × h      │     │ Heron's  │
  └─────────┘     └─────────────┘     └──────────┘


OVERRIDE RULES:
  1. Base method must be 'virtual'
  2. Derived method must be 'override'
  3. Same signature (name, parameters, return type)
  4. Cannot be private or static
```

Task 5: Animal Abstract Class

Description

Create an abstract Animal class with abstract methods (MakeSound, Move).

Create concrete classes (Dog, Cat, Bird) that implement the abstract methods.

Example

```
abstract class Animal {
    public abstract void MakeSound();  // No body!
    public abstract void Move();
}

class Dog : Animal {
    public override void MakeSound() {
        Console.WriteLine("Woof! Woof!");
    }
    public override void Move() {
        Console.WriteLine("Running on four legs!");
    }
}

// Animal a = new Animal();  ← ERROR! Cannot create!
Dog d = new Dog();              // OK!
d.MakeSound();                  // "Woof! Woof!"
```

Illustration

```
ABSTRACT CLASS:
Cannot create objects - meant to be inherited!


  ┌─────────────────────────────────────────────────┐
  │          abstract class Animal                  │
  │                                                 │
  │  public abstract void MakeSound();  ← NO BODY! │
  │  public abstract void Move();       ← NO BODY! │
  │                                                 │
  │  public void Sleep() { ... }  ← Regular method │
  └─────────────────────────────────────────────────┘
                        │
          ┌─────────────┼─────────────┐
          ↓             ↓             ↓
   ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
   │     Dog      │ │     Cat      │ │     Bird     │
   ├──────────────┤ ├──────────────┤ ├──────────────┤
   │ MakeSound()  │ │ MakeSound()  │ │ MakeSound()  │
   │  → "Woof!"   │ │  → "Meow!"   │ │  → "Tweet!"  │
   │              │ │              │ │              │
   │ Move()       │ │ Move()       │ │ Move()       │
   │  → "Running" │ │  → "Walking" │ │  → "Flying"  │
   └──────────────┘ └──────────────┘ └──────────────┘


ABSTRACT vs VIRTUAL:

  ABSTRACT METHOD:
```

```
  • NO implementation in base class
  • MUST be overridden in derived class
  • Can ONLY exist in abstract class

  VIRTUAL METHOD:
  • HAS implementation in base class
  • CAN be overridden (optional)
  • Can exist in any class


ABSTRACT CLASS RULES:
  1. Use 'abstract' keyword before class
  2. Cannot create objects with 'new'
  3. Can have abstract methods (no body)
  4. Can have virtual methods (with body)
  5. Can have regular methods
  6. Derived class MUST implement all abstract methods
```

Task 6: IMovable Interface

Description

Create an IMovable interface with methods (Move, Stop, GetSpeed).

Create classes (Car, Robot) that implement the interface. Robot should implement multiple interfaces.

Example

```
interface IMovable {
    void Move();
    void Stop();
    int GetSpeed();
}

class Car : IMovable {
    public void Move() { speed = 60; }
    public void Stop() { speed = 0; }
    public int GetSpeed() { return speed; }
}
```

```
class Robot : IMovable, IChargeable {
    // Must implement ALL methods from BOTH interfaces!
}
```

Illustration

```
INTERFACE = CONTRACT
Defines WHAT to do, not HOW to do it.

  ┌─────────────────────────────────────────────────┐
  │              interface IMovable                 │
  │                                                 │
  │   void Move();      ← No implementation         │
  │   void Stop();      ← No access modifier        │
  │   int GetSpeed();   ← Just the signature        │
  │                                                 │
  └─────────────────────────────────────────────────┘
                        │
              ┌─────────┴─────────┐
              ↓                   ↓
       ┌─────────────┐     ┌─────────────┐
       │     Car     │     │    Robot    │
       ├─────────────┤     ├─────────────┤
       │ Move()      │     │ Move()      │
       │  → drives   │     │  → walks    │
       │ Stop()      │     │ Stop()      │
       │  → brakes   │     │  → halts    │
       │ GetSpeed()  │     │ GetSpeed()  │
       └─────────────┘     └─────────────┘
          MUST implement ALL interface methods!


MULTIPLE INTERFACES:

  ┌───────────────┐     ┌───────────────┐
  │   IMovable    │     │  IChargeable  │
  └───────┬───────┘     └───────┬───────┘
          │                     │
          └─────────┬───────────┘
                    ↓
```

```
            ┌───────────────┐
            │     Robot     │
            │               │
            │ Implements    │
            │ BOTH!         │
            └───────────────┘


INTERFACE vs ABSTRACT CLASS:

  ┌─────────────────────────┬─────────────────────────┐
  │       INTERFACE         │     ABSTRACT CLASS      │
  ├─────────────────────────┼─────────────────────────┤
  │ No implementation       │ Can have implementation │
  │ No access modifiers     │ Has access modifiers    │
  │ Class can implement MANY│ Class inherits only ONE │
  │ No fields               │ Can have fields         │
  │ Defines behavior        │ Defines identity        │
  └─────────────────────────┴─────────────────────────┘
```

Task 7: Student with Properties

Description

Create a Student class using properties (get/set) instead of traditional getter and setter methods.

Include validation in the set accessor.

Example

```
class Student {
    private int age;

    public int Age {
        get { return age; }
        set {
            if (value >= 16 && value <= 100)
                age = value;
        }
    }
}

Student s = new Student();
s.Age = 20;         // Calls SET
int a = s.Age;      // Calls GET
s.Age = 10;         // Validation fails, age unchanged!
```

Illustration

```
PROPERTY = GET + SET

  ┌─────────────────────────────────────────────────┐
  │  private int age;   ← Private field             │
  │                                                 │
  │  public int Age     ← Property (like a field)   │
  │  {                                              │
  │      get { return age; }      ← Called on READ  │
  │      set { age = value; }     ← Called on WRITE │
  │  }                                              │
  └─────────────────────────────────────────────────┘


USING A PROPERTY:

  student.Age = 20;     →  SET is called
                            │
                            ↓
                        age = value; (value = 20)


  int x = student.Age;  →  GET is called
                            │
                            ↓
                        return age; (returns 20)


VALIDATION IN SET:

  public int Age {
      get { return age; }
      set {
          if (value >= 16 && value <= 100)  ← Check!
              age = value;
          else
              Console.WriteLine("Invalid age!");
      }
  }


PROPERTY TYPES:

  READ-WRITE:
  public int Age { get; set; }

  READ-ONLY:
  public int Id { get; }  ← No set!

  AUTO-PROPERTY (shorthand):
  public string Name { get; set; }
  // C# creates private field automatically!


PROPERTY vs GETTER/SETTER:

  TRADITIONAL:                PROPERTY:
  ┌──────────────────────┐    ┌──────────────────────┐
  │ public int GetAge()  │    │ public int Age       │
  │ { return age; }      │ →  │ {                    │
  │                      │    │   get { return age;}│
  │ public void SetAge   │    │   set { age=value;} │
  │ (int v){ age = v; }  │    │ }                    │
  └──────────────────────┘    └──────────────────────┘

  student.GetAge();           student.Age
  student.SetAge(20);         student.Age = 20;
```

Task 8: Bank System (Full OOP)

Description

Create a mini bank system combining ALL OOP concepts:

- Abstract class (Account)
- Interfaces (IPrintable, ITransactable)
- Inheritance (SavingsAccount, CheckingAccount)
- Properties (Balance, AccountNumber)
- Method overriding (CalculateInterest)

Example

```
abstract class Account : IPrintable, ITransactable { }

class SavingsAccount : Account {
    public override double CalculateInterest() {
        return balance * interestRate;
    }
}

SavingsAccount savings = new SavingsAccount("Ahmed", 1000, 5);
savings.Deposit(500);
savings.ApplyInterest();
savings.PrintDetails();
```

Illustration

```
FULL OOP ARCHITECTURE:

  ┌───────────────┐     ┌────────────────┐
  │  IPrintable   │     │ ITransactable  │
  │               │     │                │
  │ PrintDetails()│     │ Deposit()      │
  │               │     │ Withdraw()     │
  └───────┬───────┘     └───────┬────────┘
          │                     │
          └─────────┬───────────┘
                    ↓
  ┌─────────────────────────────────────────────┐
  │           abstract class Account            │
  │                                             │
  │  Properties:                                │
  │    - AccountNumber { get; }                 │
  │    - Balance { get; }                       │
  │    - OwnerName { get; set; }                │
  │                                             │
  │  Abstract:                                  │
  │    - abstract CalculateInterest()           │
  │                                             │
  │  Virtual:                                   │
  │    - virtual Deposit()                      │
  │    - virtual Withdraw()                     │
  └───────────────────┬─────────────────────────┘
                      │
          ┌───────────┴───────────┐
          ↓                       ↓
  ┌───────────────────┐   ┌───────────────────┐
  │  SavingsAccount   │   │ CheckingAccount   │
  │                   │   │                   │
  │ - interestRate    │   │ - overdraftLimit  │
  │ - minimumBalance  │   │ - freeTransactions│
  │                   │   │                   │
  │ CalculateInterest │   │ CalculateInterest │
  │  → balance × rate │   │  → 0              │
  │                   │   │                   │
  │ Withdraw()        │   │ Withdraw()        │
  │  → check minimum  │   │  → allow overdraft│
  └───────────────────┘   └───────────────────┘


CONCEPTS USED:

  ┌────────────────────────────────────────────────┐
  │  1. ABSTRACT CLASS                             │
  │     Cannot create Account objects directly     │
  │                                                │
  │  2. INTERFACES                                 │
  │     IPrintable, ITransactable contracts        │
  │                                                │
  │  3. INHERITANCE                                │
  │     SavingsAccount : Account                   │
  │                                                │
  │  4. PROPERTIES                                 │
  │     Balance { get; } - read only               │
  │                                                │
  │  5. METHOD OVERRIDING                          │
  │     virtual → override                         │
  │                                                │
  │  6. POLYMORPHISM                               │
  │     Account[] can hold any account type        │
  └────────────────────────────────────────────────┘
```

Good Luck!

Notes:

- Test your code with different inputs
- Make sure your code compiles without errors
- Add comments to explain your logic
- Understand when to use abstract class vs interface

Created by: ITI