Contents

Exercise 1: Basic Class and Object Creation 3

Problem Statement 3

Why We Need Classes and Objects 3

Step-by-Step Setup 3

Solution 3

Exercise 2: Encapsulation with Private Fields 5

Problem Statement 5

Why We Need Encapsulation 5

Step-by-Step Setup 5

Solution 5

`public class BankAccount` 5

Exercise 3: Constructor Overloading 7

Problem Statement 7

Why We Need Constructor Overloading 7

Step-by-Step Setup 7

Solution 7

`public class Rectangle` 7

Exercise 4: Inheritance - Basic 9

Problem Statement 9

Why We Need Inheritance 9

Step-by-Step Setup 9

Solution 9

Exercise 5: Method Overriding and Polymorphism 12

Problem Statement 12

Why We Need Polymorphism 12

Step-by-Step Setup 12

Solution 12

Exercise 6: Interface Implementation 15

Problem Statement 15

Why We Need Interfaces 15

Step-by-Step Setup 15

Solution 15

Exercise 7: Abstract Classes 18

Problem Statement 18

Why We Need Abstract Classes 18

Step-by-Step Setup 18

Solution 18

Exercise 8: Static Members 21

Problem Statement 21

Why We Need Static Members 21
Step-by-Step Setup 21
Solution 21

C# Object-Oriented Programming Exercises with Solutions

Exercise 1: Basic Class and Object Creation

Problem Statement

You work for a HR department and need to manage employee information. Currently, you're storing employee data in separate variables, making it difficult to organize and manage. Create a system to store and display person information efficiently.

Why We Need Classes and Objects

- **Real-world Modeling**: Classes help us model real-world entities (like people) in code
- **Data Organization**: Instead of scattered variables, we group related data together
- **Code Reusability**: Once we define a class, we can create multiple objects from it
- **Encapsulation**: We can control how data is accessed and modified

Step-by-Step Setup

1. Create a `Person` class to represent individual people
2. Add properties for storing name and age
3. Create a constructor to initialize the object
4. Add a method to display person information
5. Create multiple person objects and test the functionality

Solution

```
public class

Person

{
    // Properties to store person data
public string Name { get; set; }
public int Age { get; set; }

    // Constructor to initialize the object
public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Method to display person information
public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}

// Usage example
class Program
{     static void
Main()
    {
        // Creating objects from the Person class
Person person1 = new Person("Alice", 25);
        Person person2 = new Person("Bob", 30);

        // Using the objects
person1.DisplayInfo();
person2.DisplayInfo();
    }
}
```

Exercise 2: Encapsulation with Private Fields

Problem Statement

You're building a banking system where account balances must be protected from unauthorized access. Direct access to balance could lead to security issues - someone could accidentally or maliciously modify the balance without proper validation.

Why We Need Encapsulation

- **Data Protection**: Private fields prevent unauthorized direct access
- **Validation**: We can add validation logic in methods
- **Security**: Banking operations require controlled access to sensitive data
- **Maintainability**: Changes to internal implementation don't affect external code

Step-by-Step Setup

1. Create a `BankAccount` class with a private balance field
2. Add public methods for deposit, withdrawal, and balance inquiry
3. Implement validation logic in each method
4. Ensure balance cannot be directly accessed from outside the class
5. Test with various scenarios including invalid operations

Solution

```
public class BankAccount
  {
      // Private field - cannot be accessed directly from
  outside     private decimal balance;
          public BankAccount(decimal
  initialBalance = 0)
      {
          // Validation in constructor         balance =
  initialBalance >= 0 ? initialBalance : 0;
      }         public void
  Deposit(decimal amount)
      {         if
  (amount > 0)
          {
              balance += amount;
              Console.WriteLine($"Deposited: ${amount}. New balance: ${balance}");
          }
  else
          {
              Console.WriteLine("Deposit amount must be positive.");
          }
      }         public bool
  Withdraw(decimal amount)
      {         if (amount > 0 && amount <=
  balance)
          {
              balance -= amount;
              Console.WriteLine($"Withdrawn: ${amount}. New balance: ${balance}");
  return true;
          }
  else
          {
              Console.WriteLine("Invalid withdrawal amount or insufficient funds.");
  return false;
          }
      }

      // Controlled access to balance through method
  public decimal GetBalance()
      {         return
  balance;
      }
  }
  // Usage example
  class Program
  {
      static void Main()
      {
          BankAccount account = new BankAccount(1000);
                  account.Deposit(500);    // Valid operation
  account.Withdraw(200);   // Valid operation
  account.Withdraw(2000);  // Invalid - insufficient funds

          Console.WriteLine($"Current balance: ${account.GetBalance()}");
          // account.balance = 5000; // This would cause compilation error - good!
      }
  }
```

Exercise 3: Constructor Overloading

Problem Statement

You're developing a graphics application that needs to create rectangles in different ways: sometimes you know both width and height, sometimes you want a square (same width and height), and sometimes you want a default unit rectangle.

Why We Need Constructor Overloading

- **Flexibility**: Different ways to create objects based on available information
- **User Convenience**: Users can choose the most appropriate constructor
- **Default Values**: Provide sensible defaults when specific values aren't provided
- **Code Clarity**: Each constructor serves a specific purpose

Step-by-Step Setup

1. Create a `Rectangle` class with width and height properties
2. Add a default constructor (creates unit rectangle)
3. Add a constructor for squares (one parameter)
4. Add a constructor for general rectangles (two parameters)
5. Implement methods to calculate area and perimeter
6. Test all constructor variations

Solution

```
public class Rectangle
  {     public double Width { get; set;
  }     public double Height { get;
  set; }

      // Default constructor - creates a unit
  rectangle     public Rectangle()     {
          Width = 1;
          Height = 1;
          Console.WriteLine("Created unit rectangle (1x1)");
      }

      // Constructor for squares - one parameter
  public Rectangle(double side)
      {
          Width = side;
          Height = side;
          Console.WriteLine($"Created square ({side}x{side})");
      }
```

```
    // Constructor for general rectangles - two parameters
public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
        Console.WriteLine($"Created rectangle ({width}x{height})");
    }
    public double CalculateArea()
    {          return Width *
Height;
    }          public double
CalculatePerimeter()
    {          return 2 * (Width +
Height);
    }          public void
DisplayInfo()
    {
        Console.WriteLine($"Rectangle: {Width} x {Height}");
        Console.WriteLine($"Area: {CalculateArea()}");
        Console.WriteLine($"Perimeter: {CalculatePerimeter()}");
    }
}
// Usage example
class Program
{     static void
Main()
    {
        Rectangle defaultRect = new Rectangle();      // Unit rectangle
        Rectangle square = new Rectangle(5);          // 5x5 square
        Rectangle rect = new Rectangle(4, 6);         // 4x6 rectangle

defaultRect.DisplayInfo();
square.DisplayInfo();
rect.DisplayInfo();
    }
}
```

Exercise 4: Inheritance - Basic

Problem Statement

You're developing a pet management system for a veterinary clinic. Different animals have common characteristics (name, age) but also specific behaviors. Instead of creating separate, unrelated classes for each animal, you need a way to share common features while allowing specialization.

Why We Need Inheritance

- **Code Reuse**: Common properties and methods are defined once in the base class
- **Hierarchical Organization**: Models real-world "is-a" relationships
- **Polymorphism Foundation**: Enables treating different objects uniformly
- **Maintainability**: Changes to common behavior affect all derived classes

Step-by-Step Setup

1. Create a base `Animal` class with common properties and methods
2. Create derived classes `Dog` and `Cat` that inherit from Animal
3. Override methods in derived classes for specific behaviors
4. Add unique methods to each derived class
5. Test inheritance and method overriding

Solution

```
// Base class containing common animal characteristics public class Animal
{     public string Name { get;
set; }     public int Age { get;
set; }
        public Animal(string name,
int age)
    {
        Name = name;
        Age = age;
    }

    // Virtual method - can be overridden in derived classes
public virtual void MakeSound()
    {
        Console.WriteLine($"{Name} makes a sound.");
    }

    // Common method inherited by all animals
public void Sleep()
    {
        Console.WriteLine($"{Name} is sleeping.");
    }
}

// Derived class - inherits from Animal
public class Dog : Animal
{     public string Breed { get;
set; }

    // Constructor calls base class constructor     public
Dog(string name, int age, string breed) : base(name, age)
    {
        Breed = breed;
    }

    // Override the base class method
public override void MakeSound()
    {
        Console.WriteLine($"{Name} barks: Woof! Woof!");
    }

    // Dog-specific method
public void Fetch()
    {
        Console.WriteLine($"{Name} is fetching the ball.");
    }
}

// Another derived class
public class Cat : Animal
{     public bool IsIndoor { get;
set; }
        public Cat(string name, int age, bool isIndoor) :
base(name, age)
    {
        IsIndoor = isIndoor;
    }         public override void
MakeSound()
    {
        Console.WriteLine($"{Name} meows: Meow! Meow!");
    }

    // Cat-specific method
public void Purr()
    {
        Console.WriteLine($"{Name} is purring contentedly.");
    }
}

// Usage example
class Program {
static void Main()
    {
        Dog dog = new Dog("Buddy", 3, "Golden Retriever");
        Cat cat = new Cat("Whiskers", 2, true);

        // Using inherited
methods         dog.Sleep();
cat.Sleep();

        // Using overridden methods
dog.MakeSound();
cat.MakeSound();

        // Using specific
methods         dog.Fetch();
cat.Purr();
    }
}
```

Exercise 5: Method Overriding and Polymorphism

Problem Statement

You're creating a drawing application that needs to calculate areas for different shapes. Each shape calculates area differently, but you want to treat all shapes uniformly. You need a way to call the same method name on different objects and get shape-specific behavior.

Why We Need Polymorphism

- **Uniform Interface**: Same method name works on different object types
- **Extensibility**: Easy to add new shapes without changing existing code
- **Runtime Behavior**: The correct method is chosen at runtime
- **Code Flexibility**: Write code that works with future, unknown derived classes

Step-by-Step Setup

1. Create an abstract base class `Shape` with abstract methods
2. Create concrete derived classes (`Circle`, `Square`) that implement the abstract methods
3. Use polymorphism to treat different shapes uniformly
4. Demonstrate how the correct method is called for each shape type
5. Show how easy it is to extend with new shapes

Solution

```
// Abstract base class - cannot be instantiated directly public abstract class Shape
{
    // Abstract methods - must be implemented by derived
classes     public abstract double CalculateArea();     public
abstract double CalculatePerimeter();

    // Virtual method - can be overridden but has default implementation
public virtual void DisplayInfo()
    {
        Console.WriteLine($"Area: {CalculateArea():F2}");
        Console.WriteLine($"Perimeter: {CalculatePerimeter():F2}");
    }
}

public class Circle : Shape
{     public double Radius { get;
set; }
        public Circle(double
radius)
    {
        Radius = radius;
    }

    // Must implement abstract method
public override double CalculateArea()
    {          return Math.PI * Radius *
Radius;
    }          public override double
CalculatePerimeter()
    {          return 2 * Math.PI *
Radius;
    }          public override void
DisplayInfo()
    {
        Console.WriteLine($"Circle with radius: {Radius}");
base.DisplayInfo();
    }
}

public class Square : Shape
{
    public double Side { get; set; }

    public Square(double side)
    {
        Side = side;
    }          public override double
CalculateArea()
    {          return Side
* Side;
    }          public override double
CalculatePerimeter()
    {          return 4
* Side;
    }          public override void
DisplayInfo()
    {
        Console.WriteLine($"Square with side: {Side}");
base.DisplayInfo();
    }
}

// Usage example demonstrating polymorphism
class Program
{     static void
Main()
    {
        // Array of Shape references - polymorphism in
action          Shape[] shapes = {             new
Circle(5),             new Square(4),             new
Circle(3)
        };

        Console.WriteLine("Processing shapes polymorphically:");
foreach (Shape shape in shapes)
        {
            // Same method call, different behavior for each
shape               shape.DisplayInfo();
Console.WriteLine();
        }

        // Demonstrate that the correct method is called
        Shape myShape = new Circle(7);  // Shape reference, Circle object
        Console.WriteLine($"Area of shape: {myShape.CalculateArea():F2}"); // Calls Circle's
me      }
}
```

Exercise 6: Interface Implementation

Problem Statement

You're building a vehicle rental system that needs to handle different types of vehicles (cars, motorcycles, trucks). Each vehicle type has different starting procedures and information, but the rental system needs to treat them uniformly for basic operations like start, stop, and display information.

Why We Need Interfaces

- **Contract Definition**: Interfaces define what methods a class must implement
- **Multiple Inheritance**: A class can implement multiple interfaces
- **Loose Coupling**: Code depends on interfaces, not concrete implementations
- **Standardization**: Ensures consistent method signatures across different classes

Step-by-Step Setup

1. Define an `IVehicle` interface with common vehicle operations
2. Create different vehicle classes that implement the interface
3. Each class provides its own implementation of interface methods
4. Demonstrate how interface references can hold different object types
5. Show how new vehicle types can be easily added

Solution

```
// Interface defines the contract for all vehicles public interface IVehicle
{     string Brand { get;
set; }     string Model {
get; set; }     void Start();
void Stop();     void
DisplayInfo();
}

public class Car : IVehicle
{     public string Brand { get; set; }
public string Model { get; set; }
public int NumberOfDoors { get; set; }
        public Car(string brand, string model, int
doors)
    {
        Brand = brand;
        Model = model;
        NumberOfDoors = doors;
    }
    // Implementing interface methods with car-specific
behavior     public void Start()
    {
        Console.WriteLine($"{Brand} {Model} car engine started with ignition key.");
    }
    public void Stop()
    {
        Console.WriteLine($"{Brand} {Model} car engine stopped and doors locked.");
    }          public void
DisplayInfo()
    {
        Console.WriteLine($"Car: {Brand} {Model}, Doors: {NumberOfDoors}");
    }
}

public class Motorcycle : IVehicle
{     public string Brand { get;
set; }     public string Model {
get; set; }     public int EngineCC
{ get; set; }     public
Motorcycle(string brand, string
model, int engineCC)
    {
        Brand = brand;
        Model = model;
        EngineCC = engineCC;
    }

    // Same interface methods, different implementation
public void Start()
    {
        Console.WriteLine($"{Brand} {Model} motorcycle engine roared to life with electric
star
    }          public
void Stop()
    {
        Console.WriteLine($"{Brand} {Model} motorcycle engine stopped and kickstand
deployed.")
    }          public void
DisplayInfo()
    {
        Console.WriteLine($"Motorcycle: {Brand} {Model}, Engine: {EngineCC}CC");
    }
}

// Usage example
class Program
```

```
{     static void
Main()
    {
        // Interface references can hold different implementing
objects          IVehicle[] vehicles = {             new
Car("Toyota", "Camry", 4),             new Motorcycle("Honda",
"CBR600", 600),             new Car("Ford", "Mustang", 2)
        };

        Console.WriteLine("Vehicle Rental System - Starting all vehicles:");
foreach (IVehicle vehicle in vehicles)
        {
vehicle.DisplayInfo();
vehicle.Start();
vehicle.Stop();
Console.WriteLine();
        }
    }
}
```

Exercise 7: Abstract Classes

Problem Statement

You're developing a payroll system for a company that has different types of employees: full-time and part-time. Both employee types share common information (name, ID) and some common behaviors, but salary calculation differs significantly between them. You need a way to enforce that all employee types implement salary calculation while sharing common functionality.

Why We Need Abstract Classes

- **Partial Implementation**: Can provide some implemented methods and some abstract ones
- **Shared Code**: Common functionality is implemented once in the abstract class
- **Enforced Implementation**: Abstract methods must be implemented by derived classes
- **Template Pattern**: Defines the structure that derived classes must follow

Step-by-Step Setup

1. Create an abstract `Employee` class with common properties and methods
2. Define abstract methods that derived classes must implement
3. Create concrete derived classes for different employee types
4. Implement the abstract methods with type-specific logic
5. Demonstrate how the abstract class provides both structure and implementation

Solution

```
// Abstract class - cannot be instantiated but provides structure public abstract class
Employee
{     public string Name { get; set;
}     public int EmployeeId { get;
set; }
        public Employee(string name, int
employeeId)
    {
        Name = name;
        EmployeeId = employeeId;
    }

    // Concrete method - shared by all employees
public void DisplayBasicInfo()
    {
        Console.WriteLine($"Employee: {Name}, ID: {EmployeeId}");
    }
    // Abstract methods - must be implemented by derived
classes     public abstract double CalculateSalary();
public abstract string GetJobTitle();

    // Template method using abstract methods
public void DisplayFullInfo()
    {
        DisplayBasicInfo();
        Console.WriteLine($"Job Title: {GetJobTitle()}");
        Console.WriteLine($"Annual Salary: ${CalculateSalary():F2}");
    }
}

public class FullTimeEmployee : Employee
{     public double MonthlySalary { get;
set; }
        public   FullTimeEmployee(string   name,   int   employeeId,   double
monthlySalary)         : base(name, employeeId)
    {
        MonthlySalary = monthlySalary;
    }

    // Must implement abstract method
public override double CalculateSalary()
    {
        return MonthlySalary * 12; // Annual salary
    }
        public override string
GetJobTitle()
    {         return "Full-Time
Employee";
    }
}

public class PartTimeEmployee : Employee
{     public double HourlyRate { get;
set; }     public int HoursPerWeek {
get; set; }
        public  PartTimeEmployee(string  name,  int  employeeId,  double  hourlyRate,  int
hoursPerWeek)          : base(name, employeeId)     {
        HourlyRate = hourlyRate;
        HoursPerWeek = hoursPerWeek;
    }         public override double
CalculateSalary()
    {         return HourlyRate * HoursPerWeek * 52; //
Annual salary
```

```
    }
    public override string GetJobTitle()
    {         return "Part-Time
Employee";
    }
}

// Usage example
class Program
{     static void
Main()     {
        // Cannot create abstract class instance: Employee emp = new Employee(); // Error!

        Employee[] employees = {             new
FullTimeEmployee("Alice Johnson", 101, 5000),
new PartTimeEmployee("Bob Smith", 102, 25, 20),
new FullTimeEmployee("Carol Williams", 103, 6000)
        };

        Console.WriteLine("Payroll Report:");
        Console.WriteLine("================");
                foreach (Employee emp in
employees)         {
            emp.DisplayFullInfo();
            Console.WriteLine();
        }

        // Calculate total payroll
        double totalPayroll = employees.Sum(emp => emp.CalculateSalary());
        Console.WriteLine($"Total Annual Payroll: ${totalPayroll:F2}");
    }
}
```

Exercise 8: Static Members

Problem Statement

You're building a student management system for a university. You need to track the total number of students enrolled and store information that's common to all students (like the school name). This information shouldn't be tied to any specific student instance but should be accessible to all.

Why We Need Static Members

- **Shared Data**: Information that belongs to the class, not individual instances
- **Global Access**: Can be accessed without creating an object
- **Memory Efficiency**: Only one copy exists regardless of how many objects are created
- **Utility Functions**: Methods that don't need instance data

Step-by-Step Setup

1. Create a `Student` class with both instance and static members
2. Use a static field to count the number of students created
3. Add static properties for school-wide information
4. Implement static methods for class-level operations
5. Demonstrate the difference between static and instance members

Solution

```
public class Student
{
    // Static members - shared by all instances
private static int studentCount = 0;      public
static string SchoolName = "ABC University";
public static string CurrentSemester = "Fall 2024";

    // Instance members - unique to each
student      public string Name { get; set; }
public int StudentId { get; private set; }
public string Major { get; set; }
        public Student(string name, string major =
"Undeclared")
    {
        Name = name;
Major = major;
        studentCount++;  // Increment static counter
StudentId = studentCount;  // Assign unique ID

        Console.WriteLine($"New student enrolled: {Name} (ID: {StudentId})");
    }
```

```csharp
    // Static method - can be called without creating an instance
public static int GetStudentCount()
    {         return
studentCount;
    }         public static void
DisplaySchoolInfo()
    {
        Console.WriteLine($"School: {SchoolName}");
        Console.WriteLine($"Current Semester: {CurrentSemester}");
        Console.WriteLine($"Total Students Enrolled: {studentCount}");
    }         public static void
SetCurrentSemester(string semester)
    {
        CurrentSemester = semester;
        Console.WriteLine($"Semester updated to: {semester}");
    }

    // Instance method     public
void DisplayStudentInfo()
    {
        Console.WriteLine($"Student: {Name} (ID: {StudentId})");
        Console.WriteLine($"Major: {Major}");
        Console.WriteLine($"School: {SchoolName}"); // Can access static members
    }         public void
ChangeMajor(string newMajor)
    {
        string oldMajor = Major;
        Major = newMajor;
        Console.WriteLine($"{Name} changed major from {oldMajor} to {newMajor}");
    }
}

// Usage example
class Program
{     static void
Main()
    {
        // Display school info before creating any students
Console.WriteLine("=== Initial School Status ===");
        Student.DisplaySchoolInfo();
        Console.WriteLine();

        // Create students
        Console.WriteLine("=== Enrolling Students ===");
        Student alice = new Student("Alice Johnson", "Computer Science");
        Student bob = new Student("Bob Smith", "Mathematics");
        Student carol = new Student("Carol Williams");

        Console.WriteLine();

        // Display updated school info
        Console.WriteLine("=== Updated School Status ===");
        Student.DisplaySchoolInfo();
        Console.WriteLine();

        // Display individual student info
        Console.WriteLine("=== Student Details ===");
alice.DisplayStudentInfo();
bob.DisplayStudentInfo();
carol.DisplayStudentInfo();

        Console.WriteLine();

        // Change semester (affects all students)
        Student.SetCurrentSemester("Spring 2025");

        // Access static members directly through class name
        Console.WriteLine($"Current  enrollment:  {Student.GetStudentCount()}  students");
//    Instance    operations                                   carol.ChangeMajor("Biology");
carol.DisplayStudentInfo();
    }
}
```