Complete Guide to Object-Oriented Programming Fundamentals in C#

1. Classes and Objects

Concept Explanation

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects" which contain data (fields/properties) and code (methods). A class is a blueprint or template that defines the structure and behavior of objects, while an object is an instance of a class.

Key Concepts:

- **Class**: A template that defines what an object will look like and how it will behave
- **Object**: An instance of a class - a concrete entity created from the class template
- **Instantiation**: The process of creating an object from a class using the `new` keyword

Benefits:

- Code reusability and modularity
- Better organization of code
- Real-world modeling capability
- Easier maintenance and debugging

Sample Program:

```csharp
```

```
using System;

// Class definition - blueprint for creating objects
public class Student
{
    // Fields (data members)
    public string name;
    public int age;
    public string studentId;
    public double gpa;

    // Method to display student information
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name}");
        Console.WriteLine($"Age: {age}");
        Console.WriteLine($"Student ID: {studentId}");
        Console.WriteLine($"GPA: {gpa:F2}");
        Console.WriteLine("------------------------");
    }

    // Method to calculate if student is eligible for honors
    public bool IsEligibleForHonors()
    {
        return gpa >= 3.5;
    }

    // Method to update GPA
    public void UpdateGPA(double newGPA)
    {
        if (newGPA >= 0.0 && newGPA <= 4.0)
        {
            gpa = newGPA;
            Console.WriteLine($"GPA updated to {gpa:F2}");
        }
        else
        {
            Console.WriteLine("Invalid GPA. Must be between 0.0 and 4.0");
        }
    }
}

// Another class example - Car
```

```
public class Car
{
    public string make;
    public string model;
    public int year;
    public string color;
    public double mileage;
    public bool isRunning;

    public void StartEngine()
    {
        if (!isRunning)
        {
            isRunning = true;
            Console.WriteLine($"The {year} {make} {model} engine is now running.");
        }
        else
        {
            Console.WriteLine("Engine is already running.");
        }
    }

    public void StopEngine()
    {
        if (isRunning)
        {
            isRunning = false;
            Console.WriteLine($"The {year} {make} {model} engine has been turned off.");
        }
        else
        {
            Console.WriteLine("Engine is already off.");
        }
    }

    public void Drive(double miles)
    {
        if (isRunning && miles > 0)
        {
            mileage += miles;
            Console.WriteLine($"Drove {miles} miles. Total mileage: {mileage}");
        }
        else if (!isRunning)
        {
```

```
            Console.WriteLine("Cannot drive. Engine is not running.");
        }
        else
        {
            Console.WriteLine("Invalid distance.");
        }
    }

    public void DisplayCarInfo()
    {
        Console.WriteLine($"Car: {year} {make} {model}");
        Console.WriteLine($"Color: {color}");
        Console.WriteLine($"Mileage: {mileage} miles");
        Console.WriteLine($"Engine Status: {(isRunning ? "Running" : "Off")}");
        Console.WriteLine("------------------------");
    }
}

class ClassesAndObjectsDemo
{
    static void Main()
    {
        Console.WriteLine("=== CLASSES AND OBJECTS DEMO ===\n");

        // 1. Creating objects (instantiation)
        Console.WriteLine("1. Creating Student Objects:");

        // Create first student object
        Student student1 = new Student();
        student1.name = "Alice Johnson";
        student1.age = 20;
        student1.studentId = "STU001";
        student1.gpa = 3.8;

        // Create second student object
        Student student2 = new Student();
        student2.name = "Bob Smith";
        student2.age = 19;
        student2.studentId = "STU002";
        student2.gpa = 3.2;

        // Display student information
        Console.WriteLine("Student 1 Information:");
        student1.DisplayInfo();
```

```
Console.WriteLine("Student 2 Information:");
student2.DisplayInfo();

// 2. Using object methods
Console.WriteLine("2. Using Object Methods:");

Console.WriteLine($"Is {student1.name} eligible for honors? {student1.IsEligibleForHonors()}");
Console.WriteLine($"Is {student2.name} eligible for honors? {student2.IsEligibleForHonors()}");

// Update GPA
Console.WriteLine($"\nUpdating {student2.name}'s GPA:");
student2.UpdateGPA(3.6);
Console.WriteLine($"Now eligible for honors? {student2.IsEligibleForHonors()}");

// 3. Working with Car objects
Console.WriteLine("\n3. Working with Car Objects:");

// Create car objects
Car car1 = new Car();
car1.make = "Toyota";
car1.model = "Camry";
car1.year = 2022;
car1.color = "Blue";
car1.mileage = 15000;
car1.isRunning = false;

Car car2 = new Car();
car2.make = "Honda";
car2.model = "Civic";
car2.year = 2021;
car2.color = "Red";
car2.mileage = 25000;
car2.isRunning = false;

// Display initial car information
Console.WriteLine("Initial Car Information:");
car1.DisplayCarInfo();
car2.DisplayCarInfo();

// Demonstrate car operations
Console.WriteLine("Car Operations:");
car1.StartEngine();
car1.Drive(50);
```

```
car1.DisplayCarInfo();

car2.Drive(30); // This should fail - engine not running
car2.StartEngine();
car2.Drive(30); // This should work
car2.StopEngine();

// 4. Multiple objects of the same class
Console.WriteLine("\n4. Multiple Objects Demonstration:");

Student[] students = new Student[3];

// Create multiple student objects
for (int i = 0; i < students.Length; i++)
{
    students[i] = new Student();
    students[i].name = $"Student {i + 1}";
    students[i].age = 18 + i;
    students[i].studentId = $"STU{(i + 1):D3}";
    students[i].gpa = 2.5 + (i * 0.3);
}

// Display all students
Console.WriteLine("All Students:");
foreach (Student student in students)
{
    student.DisplayInfo();
}

// Count honors students
int honorsCount = 0;
foreach (Student student in students)
{
    if (student.IsEligibleForHonors())
    {
        honorsCount++;
    }
}

Console.WriteLine($"Number of students eligible for honors: {honorsCount}");

// 5. Object independence demonstration
Console.WriteLine("\n5. Object Independence:");
```

```
        Student original = new Student();
        original.name = "Original Student";
        original.gpa = 3.0;

        Student copy = new Student();
        copy.name = "Copy Student";
        copy.gpa = 3.0;

        Console.WriteLine("Before modifying:");
        Console.WriteLine($"Original GPA: {original.gpa}");
        Console.WriteLine($"Copy GPA: {copy.gpa}");

        // Modify one object
        original.UpdateGPA(4.0);

        Console.WriteLine("After modifying original:");
        Console.WriteLine($"Original GPA: {original.gpa}");
        Console.WriteLine($"Copy GPA: {copy.gpa}");
        Console.WriteLine("Objects are independent - changing one doesn't affect the other");
    }
}
```

---

2. Fields, Properties, and Methods

Concept Explanation

Fields, properties, and methods are the core members of a class that define its data and behavior.

Fields:

- Variables that store data for the class
- Can be public, private, protected, or internal
- Directly accessible (if public)
- Should generally be private for encapsulation

Properties:

- Provide controlled access to fields
- Use get and set accessors
- Can include validation logic
- Auto-implemented properties for simple cases

- Can be read-only, write-only, or read-write

Methods:

- Functions that define what objects can do
- Can access and modify fields and properties
- Can be static (belong to class) or instance (belong to object)
- Support overloading

Sample Program:

```csharp
```

```
using System;
using System.Collections.Generic;

public class BankAccount
{
    // 1. FIELDS (data storage)
    private string accountNumber;    // Private field - encapsulated
    private double balance;          // Private field - protected data
    private string accountType;      // Private field
    private DateTime creationDate;  // Private field
    private List<string> transactions; // Private field for transaction history

    // Public field (generally not recommended, but shown for demonstration)
    public string bankName = "ABC Bank";

    // 2. PROPERTIES (controlled access to data)

    // Auto-implemented property (C# creates hidden backing field)
    public string OwnerName { get; set; }

    // Property with backing field and validation
    public string AccountNumber
    {
        get { return accountNumber; }
        set
        {
            if (string.IsNullOrEmpty(value) || value.Length != 10)
            {
                throw new ArgumentException("Account number must be 10 digits");
            }
            accountNumber = value;
        }
    }

    // Read-only property (only getter)
    public double Balance
    {
        get { return balance; }
    }

    // Property with validation in setter
    public string AccountType
    {
```

```
    get { return accountType; }
    set
    {
        if (value == "Savings" || value == "Checking" || value == "Business")
        {
            accountType = value;
        }
        else
        {
            throw new ArgumentException("Invalid account type");
        }
    }
}

// Read-only property
public DateTime CreationDate
{
    get { return creationDate; }
}

// Computed property (calculated from other data)
public string AccountSummary
{
    get
    {
        return $"{OwnerName} - {AccountType} Account ({AccountNumber}) - Balance: ${Balance:F2}";
    }
}

// Property returning collection count
public int TransactionCount
{
    get { return transactions.Count; }
}

// 3. METHODS (behavior and operations)

// Constructor method (special method for object initialization)
public BankAccount(string ownerName, string accountNumber, string accountType)
{
    OwnerName = ownerName;
    AccountNumber = accountNumber; // Uses property setter for validation
    AccountType = accountType;     // Uses property setter for validation
    balance = 0.0;
```

```
    creationDate = DateTime.Now;
    transactions = new List<string>();
    AddTransaction($"Account created for {ownerName}");
}

// Method with return value
public bool Deposit(double amount)
{
    if (amount <= 0)
    {
        Console.WriteLine("Deposit amount must be positive");
        return false;
    }

    balance += amount;
    AddTransaction($"Deposited ${amount:F2}");
    Console.WriteLine($"Deposited ${amount:F2}. New balance: ${balance:F2}");
    return true;
}

// Method with validation and conditional logic
public bool Withdraw(double amount)
{
    if (amount <= 0)
    {
        Console.WriteLine("Withdrawal amount must be positive");
        return false;
    }

    if (amount > balance)
    {
        Console.WriteLine("Insufficient funds");
        return false;
    }

    balance -= amount;
    AddTransaction($"Withdrew ${amount:F2}");
    Console.WriteLine($"Withdrew ${amount:F2}. New balance: ${balance:F2}");
    return true;
}

// Method for transferring between accounts
public bool TransferTo(BankAccount targetAccount, double amount)
{
```

```
    if (amount <= 0)
    {
        Console.WriteLine("Transfer amount must be positive");
        return false;
    }

    if (this.Withdraw(amount))
    {
        targetAccount.Deposit(amount);
        AddTransaction($"Transferred ${amount:F2} to {targetAccount.OwnerName}");
        targetAccount.AddTransaction($"Received ${amount:F2} from {this.OwnerName}");
        return true;
    }

    return false;
}

// Private method (helper method)
private void AddTransaction(string description)
{
    string transaction = $"{DateTime.Now:yyyy-MM-dd HH:mm:ss} - {description}";
    transactions.Add(transaction);
}

// Method to display information
public void DisplayAccountInfo()
{
    Console.WriteLine("=== ACCOUNT INFORMATION ===");
    Console.WriteLine($"Bank: {bankName}");
    Console.WriteLine($"Owner: {OwnerName}");
    Console.WriteLine($"Account Number: {AccountNumber}");
    Console.WriteLine($"Account Type: {AccountType}");
    Console.WriteLine($"Balance: ${Balance:F2}");
    Console.WriteLine($"Created: {CreationDate:yyyy-MM-dd}");
    Console.WriteLine($"Total Transactions: {TransactionCount}");
    Console.WriteLine("============================");
}

// Method to show transaction history
public void ShowTransactionHistory(int lastNTransactions = 5)
{
    Console.WriteLine($"\n=== LAST {Math.Min(lastNTransactions, transactions.Count)} TRANSACTIONS ===");

    int startIndex = Math.Max(0, transactions.Count - lastNTransactions);
```

```
        for (int i = startIndex; i < transactions.Count; i++)
        {
            Console.WriteLine($"{i + 1}. {transactions[i]}");
        }
        Console.WriteLine("================================");
    }

    // Method with multiple parameters and default values
    public void GenerateStatement(bool includeTransactions = true, int transactionLimit = 10)
    {
        Console.WriteLine("\n=== ACCOUNT STATEMENT ===");
        Console.WriteLine(AccountSummary);
        Console.WriteLine($"Statement Date: {DateTime.Now:yyyy-MM-dd HH:mm:ss}");

        if (includeTransactions)
        {
            ShowTransactionHistory(transactionLimit);
        }
        Console.WriteLine("==========================");
    }
}

// Another class to demonstrate different types of members
public class Calculator
{
    // Static field (belongs to the class, not instance)
    public static string CalculatorName = "Advanced Calculator v1.0";
    private static int calculationCount = 0;

    // Instance fields
    private double lastResult;
    private List<string> history;

    // Property for last result
    public double LastResult
    {
        get { return lastResult; }
    }

    // Static property
    public static int TotalCalculations
    {
        get { return calculationCount; }
    }
```

```
// Constructor
public Calculator()
{
    lastResult = 0;
    history = new List<string>();
}

// Instance methods
public double Add(double a, double b)
{
    lastResult = a + b;
    RecordCalculation($"{a} + {b} = {lastResult}");
    return lastResult;
}

public double Multiply(double a, double b)
{
    lastResult = a * b;
    RecordCalculation($"{a} × {b} = {lastResult}");
    return lastResult;
}

// Static method (doesn't need instance)
public static double Power(double baseNumber, double exponent)
{
    calculationCount++;
    return Math.Pow(baseNumber, exponent);
}

private void RecordCalculation(string calculation)
{
    calculationCount++;
    history.Add($"{DateTime.Now:HH:mm:ss} - {calculation}");
}

public void ShowHistory()
{
    Console.WriteLine("Calculation History:");
    foreach (string record in history)
    {
        Console.WriteLine($"  {record}");
    }
}
```

```
}

class FieldsPropertiesMethodsDemo
{
    static void Main()
    {
        Console.WriteLine("=== FIELDS, PROPERTIES, AND METHODS DEMO ===\n");

        // 1. Creating and using objects with properties
        Console.WriteLine("1. Creating Bank Accounts:");

        try
        {
            BankAccount account1 = new BankAccount("John Doe", "1234567890", "Savings");
            BankAccount account2 = new BankAccount("Jane Smith", "0987654321", "Checking");

            // Display initial account information
            account1.DisplayAccountInfo();
            account2.DisplayAccountInfo();

            // 2. Using methods to modify object state
            Console.WriteLine("\n2. Performing Banking Operations:");

            account1.Deposit(1000);
            account1.Deposit(500);
            account1.Withdraw(200);

            account2.Deposit(750);
            account2.Withdraw(100);

            // 3. Demonstrating property access
            Console.WriteLine("\n3. Property Access:");
            Console.WriteLine($"Account 1 Balance: ${account1.Balance:F2}");
            Console.WriteLine($"Account 1 Summary: {account1.AccountSummary}");
            Console.WriteLine($"Account 2 Transaction Count: {account2.TransactionCount}");

            // 4. Transfer between accounts
            Console.WriteLine("\n4. Transfer Operations:");
            account1.TransferTo(account2, 300);

            // Show updated balances
            Console.WriteLine($"Account 1 Balance after transfer: ${account1.Balance:F2}");
            Console.WriteLine($"Account 2 Balance after transfer: ${account2.Balance:F2}");
```

```
    // 5. Show transaction history
    Console.WriteLine("\n5. Transaction History:");
    account1.ShowTransactionHistory(3);
    account2.ShowTransactionHistory(3);

    // 6. Generate statements
    Console.WriteLine("\n6. Account Statements:");
    account1.GenerateStatement(true, 5);

    // 7. Demonstrating Calculator class with static members
    Console.WriteLine("\n7. Calculator Demo (Static vs Instance):");

    Calculator calc1 = new Calculator();
    Calculator calc2 = new Calculator();

    Console.WriteLine($"Calculator Name: {Calculator.CalculatorName}");
    Console.WriteLine($"Initial calculation count: {Calculator.TotalCalculations}");

    // Instance method calls
    calc1.Add(10, 5);
    calc1.Multiply(3, 4);

    calc2.Add(20, 15);

    // Static method call (no instance needed)
    double powerResult = Calculator.Power(2, 8);
    Console.WriteLine($"2^8 = {powerResult}");

    Console.WriteLine($"Total calculations performed: {Calculator.TotalCalculations}");
    Console.WriteLine($"Calc1 last result: {calc1.LastResult}");
    Console.WriteLine($"Calc2 last result: {calc2.LastResult}");

    calc1.ShowHistory();
    calc2.ShowHistory();
}
catch (ArgumentException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}

// 8. Demonstrating property validation
Console.WriteLine("\n8. Property Validation Demo:");
try
{
```

```
            BankAccount invalidAccount = new BankAccount("Test User", "123", "Savings"); // Invalid account number
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Validation Error: {ex.Message}");
        }

        try
        {
            BankAccount account = new BankAccount("Test User", "1111111111", "InvalidType"); // Invalid account type
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Validation Error: {ex.Message}");
        }
    }
}
```

---

3. Constructors and Destructors

Concept Explanation

Constructors and destructors are special methods that manage the lifecycle of objects.

Constructors:

- Special methods called when an object is created
- Used to initialize object state
- Have the same name as the class
- No return type (not even void)
- Can be overloaded
- Default constructor is provided if none is defined

Types of Constructors:

- Default constructor (no parameters)
- Parameterized constructor
- Copy constructor
- Static constructor

Destructors (Finalizers):

- Called when object is being destroyed
- Used for cleanup operations
- Rarely needed in C# due to garbage collection
- Cannot be called directly
- Use `~ClassName()` syntax

Sample Program:

csharp

```
using System;
using System.Collections.Generic;

// Class demonstrating various constructor types
public class Person
{
    // Fields
    private string firstName;
    private string lastName;
    private int age;
    private DateTime birthDate;
    private string email;
    private static int personCount = 0; // Static field to track total persons created

    // Properties
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value?.Trim(); }
    }

    public string LastName
    {
        get { return lastName; }
        set { lastName = value?.Trim(); }
    }

    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0 && value <= 150)
                age = value;
            else
                throw new ArgumentException("Age must be between 0 and 150");
        }
    }

    public DateTime BirthDate
    {
        get { return birthDate; }
        set { birthDate = value; }
```

```
}

public string Email
{
    get { return email; }
    set { email = value; }
}

public string FullName
{
    get { return $"{FirstName} {LastName}"; }
}

public static int PersonCount
{
    get { return personCount; }
}

// 1. Static Constructor - called once when class is first used
static Person()
{
    Console.WriteLine("Static constructor called - Person class initialized");
    personCount = 0;
}

// 2. Default Constructor (no parameters)
public Person()
{
    Console.WriteLine("Default constructor called");
    firstName = "Unknown";
    lastName = "Unknown";
    age = 0;
    birthDate = DateTime.Now;
    email = "";
    personCount++;
    Console.WriteLine($"Person created. Total persons: {personCount}");
}

// 3. Parameterized Constructor (with parameters)
public Person(string firstName, string lastName)
{
    Console.WriteLine("Parameterized constructor (name only) called");
    FirstName = firstName;  // Using property for validation
    LastName = lastName;    // Using property for validation
```

```
    age = 0;
    birthDate = DateTime.Now;
    email = "";
    personCount++;
    Console.WriteLine($"Person '{FullName}' created. Total persons: {personCount}");
}

// 4. Another Parameterized Constructor (constructor overloading)
public Person(string firstName, string lastName, int age)
{
    Console.WriteLine("Parameterized constructor (name and age) called");
    FirstName = firstName;
    LastName = lastName;
    Age = age;  // Using property for validation
    birthDate = DateTime.Now.AddYears(-age); // Approximate birth date
    email = "";
    personCount++;
    Console.WriteLine($"Person '{FullName}' (age {age}) created. Total persons: {personCount}");
}

// 5. Full Parameterized Constructor
public Person(string firstName, string lastName, DateTime birthDate, string email)
{
    Console.WriteLine("Full parameterized constructor called");
    FirstName = firstName;
    LastName = lastName;
    BirthDate = birthDate;
    Email = email;
    // Calculate age from birth date
    age = DateTime.Now.Year - birthDate.Year;
    if (DateTime.Now.DayOfYear < birthDate.DayOfYear)
        age--;

    personCount++;
    Console.WriteLine($"Person '{FullName}' (born {birthDate:yyyy-MM-dd}) created. Total persons: {personCount}");
}

// 6. Copy Constructor (creates a copy of another Person)
public Person(Person other)
{
    Console.WriteLine("Copy constructor called");
    if (other != null)
    {
        FirstName = other.FirstName;
```

```
            LastName = other.LastName;
            Age = other.Age;
            BirthDate = other.BirthDate;
            Email = other.Email;
            personCount++;
            Console.WriteLine($"Copy of '{FullName}' created. Total persons: {personCount}");
        }
        else
        {
            throw new ArgumentNullException("Cannot copy from null Person object");
        }
    }

    // Constructor chaining using 'this' keyword
    public Person(string firstName, string lastName, int age, string email)
        : this(firstName, lastName, DateTime.Now.AddYears(-age), email)
    {
        Console.WriteLine("Constructor chaining completed");
    }

    // Methods
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {FullName}");
        Console.WriteLine($"Age: {Age}");
        Console.WriteLine($"Birth Date: {BirthDate:yyyy-MM-dd}");
        Console.WriteLine($"Email: {Email}");
        Console.WriteLine("-------------------------");
    }

    public void CelebrateBirthday()
    {
        Age++;
        Console.WriteLine($"Happy Birthday {FirstName}! You are now {Age} years old.");
    }

    // 7. Destructor (Finalizer) - rarely used in C#
    ~Person()
    {
        Console.WriteLine($"Destructor called for {FullName}");
        // Note: In real applications, destructors are rarely needed
        // The garbage collector handles memory cleanup automatically
    }
}
```

```
// Class demonstrating resource management with constructor/destructor
public class FileManager
{
    private string fileName;
    private bool isFileOpen;
    private DateTime creationTime;

    // Constructor
    public FileManager(string fileName)
    {
        this.fileName = fileName;
        this.creationTime = DateTime.Now;
        this.isFileOpen = false;

        Console.WriteLine($"FileManager created for '{fileName}' at {creationTime}");

        // Simulate opening a file
        OpenFile();
    }

    private void OpenFile()
    {
        // Simulate file opening
        isFileOpen = true;
        Console.WriteLine($"File '{fileName}' opened successfully");
    }

    public void WriteToFile(string content)
    {
        if (isFileOpen)
        {
            Console.WriteLine($"Writing to '{fileName}': {content}");
        }
        else
        {
            Console.WriteLine("Cannot write - file is not open");
        }
    }

    public void CloseFile()
    {
        if (isFileOpen)
        {
```

```
            isFileOpen = false;
            Console.WriteLine($"File '{fileName}' closed");
        }
    }

    // Destructor for cleanup
    ~FileManager()
    {
        Console.WriteLine($"FileManager destructor called for '{fileName}'");
        if (isFileOpen)
        {
            Console.WriteLine("Cleaning up - closing file in destructor");
            CloseFile();
        }
    }
}

// Class demonstrating initialization patterns
public class BankAccount2
{
    public string AccountNumber { get; private set; }
    public string OwnerName { get; private set; }
    public decimal Balance { get; private set; }
    public DateTime CreationDate { get; private set; }
    public string AccountType { get; private set; }

    private static readonly List<string> ValidAccountTypes = new List<string>
    {
        "Savings", "Checking", "Business", "Investment"
    };

    // Static constructor to initialize static data
    static BankAccount2()
    {
        Console.WriteLine("BankAccount2 static constructor - initializing valid account types");
    }

    // Primary constructor with validation
    public BankAccount2(string accountNumber, string ownerName, string accountType, decimal initialDeposit = 0)
    {
        // Validation
        if (string.IsNullOrWhiteSpace(accountNumber) || accountNumber.Length != 10)
            throw new ArgumentException("Account number must be exactly 10 characters");
```

```
    if (string.IsNullOrWhiteSpace(ownerName))
        throw new ArgumentException("Owner name cannot be empty");

    if (!ValidAccountTypes.Contains(accountType))
        throw new ArgumentException($"Invalid account type. Valid types: {string.Join(", ", ValidAccountTypes)}");

    if (initialDeposit < 0)
        throw new ArgumentException("Initial deposit cannot be negative");

    // Initialize properties
    AccountNumber = accountNumber;
    OwnerName = ownerName.Trim();
    AccountType = accountType;
    Balance = initialDeposit;
    CreationDate = DateTime.Now;

    Console.WriteLine($"BankAccount2 created: {OwnerName} - {AccountType} account with ${Balance:F2}");
}

// Convenience constructor for savings account with minimum deposit
public BankAccount2(string ownerName, string accountNumber)
    : this(accountNumber, ownerName, "Savings", 100.00m)
{
    Console.WriteLine("Convenience constructor used - created Savings account with $100 minimum deposit");
}

public void Deposit(decimal amount)
{
    if (amount > 0)
    {
        Balance += amount;
        Console.WriteLine($"Deposited ${amount:F2}. New balance: ${Balance:F2}");
    }
}

public void DisplayAccount()
{
    Console.WriteLine($"Account: {AccountNumber} ({AccountType})");
    Console.WriteLine($"Owner: {OwnerName}");
    Console.WriteLine($"Balance: ${Balance:F2}");
    Console.WriteLine($"Created: {CreationDate:yyyy-MM-dd HH:mm:ss}");
    Console
```