Assignment 1: Library Management System

Topics Covered:

- Classes and objects
- Encapsulation
- C# syntax essentials

Objective: Create a simple library management system that demonstrates proper encapsulation principles by creating Book and Library classes with appropriate access modifiers.

Duration: 45-60 minutes

Requirements:

1. Create a `Book` class with private fields for:
    - ISBN (string)
    - Title (string)
    - Author (string)
    - Available (boolean)
2. Implement public properties with getters and setters for each field
3. Create a `Library` class that:
    - Maintains a collection of Book objects
    - Has methods to add a book, remove a book, and check if a book is available
    - Implements a method to display all books in the library

Sample Input:

```
// Create books
Book book1 = new Book();
book1.ISBN = "978-0-123456-78-9";
book1.Title = "C# Programming";
book1.Author = "John Smith";
book1.Available = true;

Book book2 = new Book();
book2.ISBN = "978-0-234567-89-0";
book2.Title = "Object-Oriented Design";
book2.Author = "Jane Doe";
book2.Available = false;

// Create library and add books
Library myLibrary = new Library();
myLibrary.AddBook(book1);
myLibrary.AddBook(book2);

// Display all books
myLibrary.DisplayAllBooks();

// Check availability
```

```
Console.WriteLine($"Is 'C# Programming' available? {myLibrary.IsBookAvailable("C# Programming")}");
```

Sample Output:

```
Library Inventory:
ISBN: 978-0-123456-78-9, Title: C# Programming, Author: John Smith, Available: True
ISBN: 978-0-234567-89-0, Title: Object-Oriented Design, Author: Jane Doe, Available: False

Is 'C# Programming' available? True
```

Assignment 2: Shape Hierarchy

Topics Covered:

- Inheritance
- Polymorphism
- Object-oriented programming principles

Objective: Implement a shape class hierarchy that demonstrates inheritance and polymorphism through method overriding.

Duration: 60-75 minutes

Requirements:

1. Create an abstract `Shape` class with:
    - Abstract method `CalculateArea()` that returns a double
    - Abstract method `CalculatePerimeter()` that returns a double
    - Virtual method `DisplayInfo()` that outputs basic information about the shape
2. Create at least three derived classes (e.g., `Circle`, `Rectangle`, `Triangle`) that:
    - Inherit from the `Shape` class
    - Implement the abstract methods appropriately
    - Override the `DisplayInfo()` method to include shape-specific information
3. Create a `ShapeManager` class that can:
    - Store a collection of different shapes
    - Calculate the total area of all shapes
    - Display information about all shapes

Sample Input:

```
// Create shapes
Circle circle = new Circle(5);
Rectangle rectangle = new Rectangle(4, 6);
Triangle triangle = new Triangle(3, 4, 5);
```

```
// Create ShapeManager and add shapes
ShapeManager manager = new ShapeManager();
manager.AddShape(circle);
manager.AddShape(rectangle);
manager.AddShape(triangle);

// Display information for all shapes
manager.DisplayAllShapes();

// Calculate and display total area
Console.WriteLine($"Total area of all shapes: {manager.CalculateTotalArea()}");
```

Sample Output:

```
Shape Information:
Circle - Radius: 5
Area: 78.54, Perimeter: 31.42

Rectangle - Width: 4, Height: 6
Area: 24, Perimeter: 20

Triangle - Sides: 3, 4, 5
Area: 6, Perimeter: 12

Total area of all shapes: 108.54
```

Assignment 3: Banking System

Topics Covered:

- Classes and objects
- Inheritance
- Polymorphism
- Encapsulation
- C# syntax essentials

Objective: Create a simple banking system that integrates all core OOP principles through account types and transactions.

Duration: 90-120 minutes

Requirements:

1. Create an abstract `Account` class with:
    - Protected fields for account number, owner name, and balance
    - Properties with appropriate access modifiers
    - A constructor that initializes the account
    - Abstract method `CalculateInterest()`
    - Virtual methods for `Deposit()` and `Withdraw()`

2. Create derived classes such as `SavingsAccount` and `CheckingAccount` that:
    - Implement the `CalculateInterest()` method differently
    - May override `Withdraw()` to implement different rules (e.g., withdrawal limits)
3. Create a `Bank` class that:
    - Maintains a collection of accounts
    - Has methods to add accounts, perform transactions, and generate account summaries
4. Implement exception handling for invalid operations (e.g., withdrawing more than the balance)

Sample Input:

```
// Create accounts
SavingsAccount savings = new SavingsAccount("S12345", "Alice Johnson", 1000,
0.05);
CheckingAccount checking = new CheckingAccount("C67890", "Bob Smith", 500, 0.01);

// Create bank and add accounts
Bank bank = new Bank("MyBank");
bank.AddAccount(savings);
bank.AddAccount(checking);

// Perform transactions
bank.DepositToAccount("S12345", 500);
bank.WithdrawFromAccount("C67890", 200);

// Calculate interest
savings.CalculateInterest();
checking.CalculateInterest();

// Display account summaries
bank.DisplayAccountSummaries();

// Try an invalid transaction
try
{
    bank.WithdrawFromAccount("C67890", 1000);
}
catch (Exception ex)
{
    Console.WriteLine($"Transaction failed: {ex.Message}");
}
```

Sample Output:

```
Transaction: Deposited $500 to account S12345
New balance: $1500

Transaction: Withdrew $200 from account C67890
New balance: $300

Interest added: $75 to account S12345
```

```
New balance: $1575

Interest added: $3 to account C67890
New balance: $303

MyBank Account Summaries:
Savings Account - Account #: S12345
Owner: Alice Johnson
Balance: $1575
Interest Rate: 5%

Checking Account - Account #: C67890
Owner: Bob Smith
Balance: $303
Interest Rate: 1%

Transaction failed: Insufficient funds for withdrawal
```