Exercise 1: Writing Classes & Initializing Objects

Lab Exercise:

1. Create a class named `Car` with the following properties:
   - `Make` (string)
   - `Model` (string)
   - `Year` (int)
2. Create a method `DisplayInfo()` that prints the car's details.
3. Initialize an object of the `Car` class with values for its properties and call the `DisplayInfo()` method.

Solution:

```csharp
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }

    public void DisplayInfo()
    {
        Console.WriteLine($"Make: {Make}, Model: {Model}, Year: {Year}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Car car = new Car
        {
```

```
        Make = "Toyota",
        Model = "Corolla",
        Year = 2020
    };
    car.DisplayInfo();
  }
}
```

Exercise 2: Access Specifiers

Lab Exercise:

1. Create a class named `BankAccount` with private fields `accountNumber` (int) and `balance` (decimal).

2. Create public methods `Deposit()` and `Withdraw()` to modify the balance.

3. Add a public method `GetBalance()` to return the balance.

Solution:

```csharp
public class BankAccount
{
  private int accountNumber;
  private decimal balance;

  public BankAccount(int accountNumber, decimal initialBalance)
  {
    this.accountNumber = accountNumber;
    this.balance = initialBalance;
  }

  public void Deposit(decimal amount)
```

```
    {
        balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= balance)
        {
            balance -= amount;
        }
        else
        {
            Console.WriteLine("Insufficient funds.");
        }
    }

    public decimal GetBalance()
    {
        return balance;
    }
}

class Program
{
    static void Main(string[] args)
    {
        BankAccount account = new BankAccount(123456, 1000);
        account.Deposit(500);
        account.Withdraw(200);
        Console.WriteLine($"Current Balance: {account.GetBalance()}");
    }
```

}
```

### Exercise 3: Writing Methods in Classes

**Lab Exercise:**

1. Create a class named `Calculator` with methods for `Add`, `Subtract`, `Multiply`, and `Divide`.
2. Each method should take two parameters and return the result.

**Solution:**

```csharp
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public int Subtract(int a, int b)
    {
        return a - b;
    }

    public int Multiply(int a, int b)
    {
        return a * b;
    }

    public double Divide(int a, int b)
    {
        if (b == 0)
```
```

```
        {
            throw new DivideByZeroException("Cannot divide by zero");
        }
        return (double)a / b;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Calculator calculator = new Calculator();
        Console.WriteLine($"Add: {calculator.Add(10, 5)}");
        Console.WriteLine($"Subtract: {calculator.Subtract(10, 5)}");
        Console.WriteLine($"Multiply: {calculator.Multiply(10, 5)}");
        Console.WriteLine($"Divide: {calculator.Divide(10, 5)}");
    }
}
```

Exercise 4: Working with Properties in Class

Lab Exercise:

1. Create a class named `Person` with properties `FirstName`, `LastName`, and `Age`.
2. Ensure that `Age` is always positive.
3. Add a method `GetFullName()` that returns the full name of the person.

Solution:

```csharp
public class Person
{
```

```
    public string FirstName { get; set; }
    public string LastName { get; set; }
    private int age;

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

    public string GetFullName()
    {
        return $"{FirstName} {LastName}";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person person = new Person
```

```
        {
            FirstName = "John",
            LastName = "Doe",
            Age = 25
        };
        Console.WriteLine($"Full Name: {person.GetFullName()}, Age: {person.Age}");
    }
}
```

Exercise 5: Constructors and Destructors

Lab Exercise:

1. Create a class named `Book` with properties `Title`, `Author`, and `Price`.
2. Implement a constructor to initialize these properties.
3. Implement a destructor that displays a message when the object is destroyed.

Solution:

```csharp
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public double Price { get; set; }

    public Book(string title, string author, double price)
    {
        Title = title;
        Author = author;
        Price = price;
    }
```

```
    ~Book()
    {
        Console.WriteLine("Book object is being destroyed.");
    }

    public void DisplayDetails()
    {
        Console.WriteLine($"Title: {Title}, Author: {Author}, Price: {Price}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Book book = new Book("The Catcher in the Rye", "J.D. Salinger", 9.99);
        book.DisplayDetails();
    }
}
```

Exercise 6: Parameterized Constructors

Lab Exercise:

1. Create a class named `Rectangle` with properties `Length` and `Breadth`.
2. Implement a parameterized constructor that initializes the `Length` and `Breadth`.
3. Add a method `CalculateArea()` to return the area of the rectangle.

Solution:

```csharp
```

```
public class Rectangle
{
    public double Length { get; set; }
    public double Breadth { get; set; }

    public Rectangle(double length, double breadth)
    {
        Length = length;
        Breadth = breadth;
    }

    public double CalculateArea()
    {
        return Length * Breadth;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Rectangle rectangle = new Rectangle(5.0, 3.0);
        Console.WriteLine($"Area of Rectangle: {rectangle.CalculateArea()}");
    }
}
```

Exercise 7: Copy Constructors

Lab Exercise:

1. Create a class named `Point` with properties `X` and `Y`.

2. Implement a copy constructor that creates a new object as a copy of an existing object.

3. Demonstrate copying a `Point` object.

Solution:

```csharp
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public Point(Point point)
    {
        X = point.X;
        Y = point.Y;
    }

    public void Display()
    {
        Console.WriteLine($"Point(X: {X}, Y: {Y})");
    }
}

class Program
{
    static void Main(string[] args)
```

```
    {
        Point p1 = new Point(10, 20);
        Point p2 = new Point(p1); // Copy constructor
        p1.Display();
        p2.Display();
    }
}
```

Exercise 8: Mutable & Immutable Types

Lab Exercise:

1. Create a class `Employee` with properties `Name` (string, immutable) and `Salary` (decimal, mutable).

2. Demonstrate changing the salary while keeping the name immutable.

Solution:

```csharp
public class Employee
{
    public string Name { get; }
    public decimal Salary { get; set; }

    public Employee(string name, decimal salary)
    {
        Name = name;
        Salary = salary;
    }

    public void DisplayInfo()
    {
```

```
        Console.WriteLine($"Employee Name: {Name}, Salary: {Salary}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee("Alice", 50000);
        employee.DisplayInfo();

        // Change salary
        employee.Salary = 55000;
        employee.DisplayInfo();

        // Trying to change the name will cause a compile-time error
        // employee.Name = "Bob"; // Not allowed
    }
}
```

Exercise 9: Singleton Pattern

Lab Exercise:

1. Implement a singleton class `Logger` that allows only one instance to be created.
2. Add a method `LogMessage()` that prints a message to the console.

Solution:

```csharp
public class Logger
{
```

```
    private static Logger instance = null;
    private static readonly object padlock = new object();

    private Logger() { }

    public static Logger Instance
    {
        get
        {
            lock (padlock)
            {
                if (instance == null)
                {


 instance = new Logger();
                }
                return instance;
            }
        }
    }

    public void LogMessage(string message)
    {
        Console.WriteLine($"Log: {message}");
    }
}

class Program
{
    static void Main(string[] args)
```

```
    {
        Logger logger1 = Logger.Instance;
        Logger logger2 = Logger.Instance;

        logger1.LogMessage("Singleton pattern example.");
        Console.WriteLine($"Are both instances equal? {logger1 == logger2}");
    }
}
```

Exercise 10: Working with Properties & Encapsulation

Lab Exercise:

1. Create a class `Student` with properties `Name`, `RollNumber`, and `Grade`.
2. Ensure that `Grade` can only be set if it is between `A` and `F`.
3. Add a method `DisplayStudentDetails()` to show the student's information.

Solution:

```csharp
public class Student
{
    public string Name { get; set; }
    public int RollNumber { get; set; }
    private char grade;

    public char Grade
    {
        get { return grade; }
        set
        {
            if (value >= 'A' && value <= 'F')
```

```
            {
                grade = value;
            }
            else
            {
                throw new ArgumentException("Grade must be between A and F");
            }
        }
    }

    public void DisplayStudentDetails()
    {
        Console.WriteLine($"Name: {Name}, RollNumber: {RollNumber}, Grade: {Grade}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Student student = new Student
        {
            Name = "John Smith",
            RollNumber = 101,
            Grade = 'B'
        };
        student.DisplayStudentDetails();
    }
}
```