Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

INTRODUCTION TO C# PROGRAMMING

UNDERSTANDING DATA TYPES, VARIABLES, LOOPS, AND FUNCTIONS

C# is a versatile, object-oriented programming language that is widely used for developing Windows applications, web applications, and mobile apps. Understanding the foundational concepts such as data types, variables, loops, and functions is essential for writing efficient and readable C# code.

Data Types in C#

Data types define the type of data a variable can store. C# is a strongly-typed language, meaning each variable must have a specific type, and the type must be declared before use. There are several categories of data types in C#, including **value types** and **reference types**.

- **Value Types**: These include simple types like int, char, float, bool, and double. Value types hold data directly, and when assigned to another variable, the data is copied.
  - Example:
  - int age = 25;
  - float temperature = 36.5f;
  - bool isAdult = true;
- **Reference Types**: These include objects like string, arrays, and class types. Reference types store a reference (or pointer) to the actual data, and when assigned to another variable, the reference is copied, not the data itself.
  - Example:
  - string name = "John Doe";

ISDM-NXT

 - string message = name; // message holds a reference to the same object as name

Variables in C#

A variable is a storage location in the memory that holds a value of a specific data type. Declaring a variable in C# involves specifying its data type followed by its name:

int number = 10;

double price = 99.99;

Variables can be initialized at the time of declaration or later on, depending on the logic of the program.

Loops in C#

Loops are essential for executing repetitive tasks. C# provides several loop constructs:

- **For Loop**: The for loop is ideal when the number of iterations is known beforehand. It includes an initialization, a condition, and an increment or decrement.
- for (int i = 0; i < 5; i++) {
-   Console.WriteLine(i); // Outputs 0 to 4
- }
- **While Loop**: The while loop runs as long as the condition evaluates to true. It's useful when the number of iterations is unknown.
- int count = 0;
- while (count < 5) {
-   Console.WriteLine(count);
-   count++;
- }
- **Do-While Loop**: This loop guarantees at least one iteration since the condition is checked after the execution of the loop body.
- int count = 0;

ISDM-NxT

- do {
-     Console.WriteLine(count);
-     count++;
- } while (count < 5);

Functions in C#

Functions, also known as methods, are used to encapsulate logic and allow code to be reused. A function has a return type, a name, parameters (optional), and a body that contains the logic.

```
int AddNumbers(int a, int b) {

   return a + b;

}
```

Functions can also return void if no result is needed:

```
void DisplayMessage() {

   Console.WriteLine("Hello, world!");

}
```

Understanding how to effectively use data types, variables, loops, and functions is critical to writing clean and efficient C# programs.

Object-Oriented Programming (OOP) Concepts in C#

Object-Oriented Programming (OOP) is a fundamental programming paradigm that organizes software design around data, or objects, rather than functions and logic. C# is an object-oriented language, and understanding its OOP principles will help you build scalable, maintainable, and reusable software applications. The four core principles of OOP are **Encapsulation**, **Abstraction**, **Inheritance**, and **Polymorphism**.

Encapsulation in C#

Encapsulation is the concept of bundling the data (variables) and the methods (functions) that operate on the data into a single unit, called a class. It restricts access to some of the object's components, making the object's internal state private and requiring all interaction to be performed through methods. This ensures that an object's state is only modified in well-defined ways.

ISDM-NXT

For example, the following class encapsulates the concept of a Car with properties and methods that define its behavior:

```
public class Car {
  private int speed;  // Private field


  public void SetSpeed(int s) {  // Public method to modify speed
    if (s > 0)
      speed = s;
  }


  public int GetSpeed() {  // Public method to retrieve speed
    return speed;
  }
}
```

Here, the speed field is encapsulated within the Car class, and external code cannot modify it directly. Instead, access to the speed is controlled through the SetSpeed and GetSpeed methods.

Abstraction in C#

Abstraction is the principle of hiding the complex implementation details from the user and only exposing the necessary functionality. This is achieved using abstract classes and interfaces.

An **abstract class** provides a blueprint for derived classes but cannot be instantiated directly. It can have abstract methods, which must be implemented by any derived class.

```
public abstract class Animal {
  public abstract void MakeSound();  // Abstract method
}
```

ISDM-NXT

```
public class Dog : Animal {

  public override void MakeSound() {

    Console.WriteLine("Bark");

  }

}
```

In this example, the Animal class defines an abstract method MakeSound, and the Dog class provides a concrete implementation of this method.

Inheritance in C#

Inheritance allows a class to inherit fields, methods, and properties from another class. This promotes code reuse and establishes a relationship between different classes. C# uses the : base keyword to implement inheritance.

Example:

```
public class Animal {

  public void Eat() {

    Console.WriteLine("Eating...");

  }

}



public class Dog : Animal {

  public void Bark() {

    Console.WriteLine("Barking...");

  }

}
```

In this example, the Dog class inherits from the Animal class and can access its methods, such as Eat.

Polymorphism in C#

Polymorphism allows objects of different types to be treated as objects of a common

ISDM-NxT

base type. This enables one interface to be used for a general class of actions, with specific actions depending on the actual object that is invoked.

Polymorphism in C# can be implemented through **method overriding** (runtime polymorphism) or **method overloading** (compile-time polymorphism).

- **Method Overriding**:

```
•  public class Animal {
•      public virtual void Sound() {
•          Console.WriteLine("Some sound...");
•      }
•  }
•
•  public class Dog : Animal {
•      public override void Sound() {
•          Console.WriteLine("Bark");
•      }
•  }
•
•  public class Cat : Animal {
•      public override void Sound() {
•          Console.WriteLine("Meow");
•      }
•  }
```

In this case, both Dog and Cat override the Sound method from the Animal class, demonstrating runtime polymorphism.

EXERCISE

ISDM-NXT

1. **Create a Class and Objects**: Write a class called Person with properties Name and Age. Create a method Introduce() that displays the person's details. Instantiate objects of the Person class and call the Introduce() method for each object.

2. **Implement a Calculator**: Build a simple calculator application in C# that allows the user to add, subtract, multiply, and divide numbers. Use functions for each operation and implement input validation.

Case Study: Building a Library Management System

In this case study, you will build a simple **Library Management System** in C# that demonstrates the application of OOP concepts. The system should have the following features:

- **Book** class with properties like Title, Author, and ISBN.
- **Library** class that manages a collection of books, with methods to add, remove, and list books.
- **Member** class representing a library member with properties like Name, MemberId, and List<Book> for the books borrowed.

This case study will help you apply OOP principles like **Encapsulation**, **Abstraction**, **Inheritance**, and **Polymorphism** to solve a real-world problem.

ISDM-NxT

C# FOR ASP.NET

INTEGRATING C# WITH ASP.NET MVC

C# plays a central role in developing ASP.Net MVC applications. It is the language used for writing the logic that powers the Model, View, and Controller (MVC) components of an ASP.Net application. The integration of C# with ASP.Net MVC enables developers to create dynamic, data-driven web applications with seamless interaction between the front-end and back-end layers.

ASP.Net MVC is based on the MVC design pattern, which separates concerns into three distinct components:

- **Model**: Represents the data and business logic of the application.
- **View**: The user interface that presents the data to the user.
- **Controller**: The intermediary between the Model and the View, handling user input and determining what response to send.

C# is the primary programming language used to implement the Model and Controller components of an ASP.Net MVC application. The Model in C# represents the data structure and often maps to a database table or a business object. The Controller is responsible for handling requests from the user, interacting with the Model to retrieve or modify data, and then passing that data to the View for display.

Example: Integrating C# in MVC

Consider an application where users can view, add, and edit information about books. The Model might represent a Book class, the Controller will handle HTTP requests, and the View will display the book information.

Model (Book.cs):

```
public class Book

{

  public int Id { get; set; }

  public string Title { get; set; }
```

ISDM-NXT

```
  public string Author { get; set; }

  public decimal Price { get; set; }

}
```

Controller (BookController.cs):

```
public class BookController : Controller

{

  private List<Book> books = new List<Book>

  {

    new Book { Id = 1, Title = "C# Programming", Author = "John Doe", Price = 29.99m
},

    new Book { Id = 2, Title = "ASP.Net MVC", Author = "Jane Smith", Price = 39.99m }

  };



  public IActionResult Index()

  {

    return View(books);

  }



  [HttpGet]

  public IActionResult Edit(int id)

  {

    var book = books.FirstOrDefault(b => b.Id == id);

    return View(book);

  }
```

ISDM-NxT

```
  [HttpPost]

  public IActionResult Edit(Book book)

  {

    var existingBook = books.FirstOrDefault(b => b.Id == book.Id);

    if (existingBook != null)

    {

      existingBook.Title = book.Title;

      existingBook.Author = book.Author;

      existingBook.Price = book.Price;

    }

    return RedirectToAction("Index");

  }

}
```

In this example, C# is used to define the data structure (Book) and the logic for handling HTTP requests (BookController). The Controller is responsible for processing data from the Model and sending it to the View.

Integrating C# in Views

ASP.Net MVC uses the Razor view engine to combine C# code with HTML in the Views. Razor allows developers to embed C# code directly into HTML markup by using the @ symbol. This integration makes it easy to dynamically display data, such as the list of books, in a View.

View (Index.cshtml):

```
@model IEnumerable<Book>


<h1>Books List</h1>

<table>
```

ISDM-NXT

```
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>Price</th>
    <th>Action</th>
  </tr>
  @foreach (var book in Model)
  {
    <tr>
      <td>@book.Title</td>
      <td>@book.Author</td>
      <td>@book.Price</td>
      <td>
        <a href="@Url.Action("Edit", new { id = book.Id })">Edit</a>
      </td>
    </tr>
  }
</table>
```

ISDM-NXT

This example shows how C# integrates with the View to dynamically generate HTML content based on the data provided by the Controller.

HANDLING USER INPUT AND EVENTS IN C#

Handling user input and events is a core responsibility in any web application. In ASP.Net MVC, user input is often gathered through forms, which can send data to the server for processing. C# plays an integral role in processing this input, validating it, and triggering appropriate actions based on the user's interactions.

Handling User Input with Forms

In ASP.Net MVC, user input is usually captured via HTML forms. The forms can send data to the server using HTTP methods like GET or POST. The Controller processes the submitted data and determines the next step. ASP.Net MVC binds user input to model objects using model binding, where form fields are automatically mapped to properties of C# classes.

Example: Handling User Input

Consider a simple example where users submit a form to create a new book. The form sends data to the Controller, which processes it and updates the list of books.

View (Create.cshtml):

```
@model Book


<h1>Create New Book</h1>

<form asp-action="Create" method="post">

  <label for="Title">Title</label>

  <input type="text" id="Title" name="Title" required />


  <label for="Author">Author</label>

  <input type="text" id="Author" name="Author" required />


  <label for="Price">Price</label>

  <input type="text" id="Price" name="Price" required />


  <button type="submit">Create</button>

</form>
```

Controller (BookController.cs):

ISDM-NXT

```
[HttpPost]

public IActionResult Create(Book book)

{

  if (ModelState.IsValid)

  {

    books.Add(book);

    return RedirectToAction("Index");

  }

  return View(book);

}
```

In this example, the form sends a POST request to the Create action of the BookController. The Book object is automatically populated with the data from the form fields, and the Controller handles the input by adding the new book to the list.

Handling Events in C#

In ASP.Net MVC, events are usually triggered by user actions, such as clicking buttons or submitting forms. C# handles these events in the Controller, where different actions can be taken depending on the user's input. For example, when a user clicks the "Delete" button next to a book, an event handler in C# can remove that book from the database or in-memory list.

Example: Handling the Delete Event

```
[HttpPost]

public IActionResult Delete(int id)

{

  var book = books.FirstOrDefault(b => b.Id == id);

  if (book != null)

  {

    books.Remove(book);
```

ISDM-NXT

```
  }

  return RedirectToAction("Index");

}
```

In this example, when the user submits a POST request to delete a book, the Delete action in the Controller is triggered. The book is removed from the list, and the user is redirected back to the main page.

EXERCISES AND CASE STUDY

Exercise:

1. Create a form that allows users to input book details (Title, Author, and Price) and submit it to the server. Implement the Controller action to save the data and display the updated list of books.
2. Modify the existing application to include a "Delete" button that removes a book from the list when clicked.

Case Study:

Company ABC is developing a web application for managing a library of books. The application needs to allow users to add new books, edit existing ones, and delete them. The company decides to use ASP.Net MVC for the development process.

Your task is to design the application using C# for handling user input and events. Define the Model for books, the Controller actions for adding, editing, and deleting books, and the Views for displaying the data and handling user interaction. Implement the necessary forms and event handlers in the Controller.

ISDM-NXT

ERROR HANDLING AND DEBUGGING

EXCEPTION HANDLING IN C#

In C#, **exception handling** is a fundamental concept that allows developers to manage errors gracefully. Instead of letting errors crash a program, C# provides a structured mechanism to catch and handle exceptions using the try, catch, finally, and throw keywords. Proper exception handling helps ensure that an application continues to run smoothly even in the event of unexpected conditions.

What is an Exception?
An exception is an event that disrupts the normal flow of a program. It can occur due to various reasons, such as invalid user input, division by zero, file not found, or network issues. When an exception occurs, it typically terminates the program if not handled properly, leading to a poor user experience. However, with exception handling, the program can respond to errors, log them, and continue its execution or exit cleanly.

Using Try-Catch Block
The most common way to handle exceptions in C# is by using a try-catch block. The try block contains the code that might throw an exception, and the catch block defines how to handle the exception if it occurs.

```
try {

  int result = 10 / 0;  // Division by zero, will throw an exception

}

catch (DivideByZeroException ex) {

  Console.WriteLine("Error: " + ex.Message);  // Handles the exception

}
```

In this example, the try block attempts to divide 10 by 0, which will throw a DivideByZeroException. The catch block then catches the exception and prints the error message.

Multiple Catch Blocks
You can have multiple catch blocks to handle different types of exceptions separately. This allows you to provide more specific error messages based on the type of exception.

ISDM-NXT

```
try {
   string text = null;
   int length = text.Length;  // NullReferenceException
}
catch (NullReferenceException ex) {
   Console.WriteLine("Null reference error: " + ex.Message);
}
catch (Exception ex) {
   Console.WriteLine("General error: " + ex.Message);
}
```

Here, the NullReferenceException is caught and handled first, while the second catch block handles any other generic exceptions.

Finally Block

The finally block, if used, is executed regardless of whether an exception is thrown or not. It's typically used for cleanup operations, such as closing files, releasing resources, or disconnecting from a database.

```
try {
   FileStream fs = new FileStream("data.txt", FileMode.Open);
   // Perform file operations
}
catch (IOException ex) {
   Console.WriteLine("File error: " + ex.Message);
}
finally {
   // Cleanup resources
   Console.WriteLine("Cleaning up...");
```

ISDM-NXT

```
}
```

Throwing Exceptions

You can throw exceptions manually using the throw keyword. This is useful when you want to create custom exceptions or when a certain condition occurs that should trigger an error.

```
if (age < 18) {

   throw new ArgumentException("Age must be 18 or older");

}
```

In this example, if the age is less than 18, an ArgumentException is thrown with a custom message.

Proper exception handling in C# is critical to ensuring that your application remains stable and behaves predictably even when errors occur. By effectively using try-catch blocks, finally clauses, and custom exceptions, developers can create resilient applications that handle errors gracefully.

DEBUGGING TECHNIQUES IN VISUAL STUDIO

Debugging is an essential skill for every developer. It allows you to examine the behavior of your code, identify issues, and correct them efficiently. Visual Studio, one of the most popular integrated development environments (IDEs) for C#, provides powerful debugging tools that allow you to step through code, inspect variables, set breakpoints, and more.

Setting Breakpoints

A breakpoint is a marker in your code that tells Visual Studio to pause execution at a specific line, allowing you to inspect the state of your application at that point. To set a breakpoint, simply click in the margin next to the line number where you want to pause. When you run the application in **Debug** mode, the debugger will halt at the breakpoint.

```
int x = 10;

int y = 20;

int result = x + y;  // Set a breakpoint here

Console.WriteLine(result);
```

ISDM-NxT

Once the debugger hits the breakpoint, you can inspect the values of x, y, and result in the **Locals** or **Watch** window.

Stepping Through Code

When the debugger pauses at a breakpoint, you can step through your code line by line to observe how variables change and how the program behaves. Visual Studio provides several stepping options:

- **Step Over (F10)**: Executes the current line of code, and moves to the next line in the same method.
- **Step Into (F11)**: Steps into a function or method call to debug the logic inside it.
- **Step Out (Shift + F11)**: Exits the current method and goes back to the caller method.

Using these stepping techniques, you can navigate through your code to isolate the source of bugs or incorrect behavior.

Inspecting Variables

Visual Studio provides several ways to inspect the values of variables during debugging:

- **Locals Window**: Shows all the local variables in the current method, along with their values.
- **Watch Window**: Allows you to specify variables or expressions to monitor as the code executes.
- **Immediate Window**: Lets you interact with your code while it's paused in the debugger. You can evaluate expressions, change variable values, or execute commands on the fly.

For example, you can add a variable to the **Watch** window by right-clicking it in the code and selecting **Add Watch**. As you step through the code, the Watch window will update with the current value of the variable.

Using Call Stack

The **Call Stack** window shows the sequence of function calls that led to the current line of code. This is particularly helpful when debugging recursive functions or complex applications with deep call hierarchies. It allows you to understand the flow of execution and see where things went wrong.

ISDM-NxT

Exception Settings

Visual Studio also offers the ability to break execution when a specific exception is thrown. You can set up **Exception Settings** by going to **Debug > Windows > Exception Settings**. Here, you can specify which exceptions (e.g., NullReferenceException, DivideByZeroException) should break execution when thrown, even if they are handled in the code.

Using Debugging Windows

Visual Studio provides several specialized windows for effective debugging:

- **Autos Window**: Automatically displays variables used in the current and previous lines of code.
- **Locals Window**: Displays variables local to the current scope.
- **Watch Window**: Lets you manually add variables and expressions to track during the debugging session.

These tools, combined with breakpoints and stepping techniques, give you full visibility into the behavior of your program, allowing you to efficiently identify and fix issues.

Exercise

1. **Handling Exceptions**: Write a C# program that handles different types of exceptions, such as FileNotFoundException, DivideByZeroException, and NullReferenceException. Use try-catch blocks to handle these exceptions and display meaningful error messages.
2. **Debugging an Application**: Create a simple application with a few logical errors. Use breakpoints and the debugging tools in Visual Studio to locate and fix the errors.

ISDM-NXT

Case Study: Debugging a Bank Account Application

In this case study, you will build a **Bank Account** application with methods to deposit and withdraw money. The application should handle exceptions such as insufficient funds or invalid input. You will also use Visual Studio's debugging tools to inspect the flow of execution and troubleshoot any issues that arise.

Scenario:

- **Class**: BankAccount with methods Deposit and Withdraw.

- **Exception Handling**: Handle errors like invalid withdrawal amounts (negative values) or withdrawal exceeding the balance.
- **Debugging**: Use breakpoints to investigate the state of the BankAccount class before and after performing the transactions.

By working through this case study, you will gain practical experience in both exception handling and debugging techniques, ensuring your application runs smoothly and is error-free.

ISDM-NXT

Assignment: Develop a Simple CRUD Application Using ASP.Net Core and C#

In this assignment, you will develop a simple CRUD (Create, Read, Update, Delete) application using ASP.Net Core and C#. This application will manage a list of books, where you can add, view, edit, and delete books. We will use the MVC (Model-View-Controller) architecture for this project.

Prerequisites:

- Visual Studio or Visual Studio Code installed on your machine
- .NET Core SDK installed
- Basic knowledge of C#, ASP.Net Core, and MVC architecture

Step 1: Create a New ASP.Net Core Project

1. **Open Visual Studio or Visual Studio Code**.
2. **Create a new project**:
   - Select "ASP.NET Core Web Application".
   - Choose **"Web Application (Model-View-Controller)"**.
   - Name the project BookCrudApp.
3. **Configure the project**:
   - Select .NET Core and ASP.NET Core 5.0 or higher.
   - Click **Create**.

Step 2: Define the Model

ISDM-NXT

1. **Create a Model Class for the Book**:
   - In the Models folder, create a new class file named Book.cs.

```
using System;


namespace BookCrudApp.Models
{
  public class Book
  {
    public int Id { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public decimal Price { get; set; }
  }
}
```

The Book class has properties for Id, Title, Author, and Price.

Step 3: Create the Controller

1. **Create a Controller**:
   - In the Controllers folder, create a new class file named BookController.cs.

```
using BookCrudApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
```

ISDM-NXT

```
namespace BookCrudApp.Controllers
{
  public class BookController : Controller
  {
    private static List<Book> books = new List<Book>
    {
      new Book { Id = 1, Title = "C# Programming", Author = "John Doe", Price = 
29.99m },
      new Book { Id = 2, Title = "ASP.Net Core", Author = "Jane Smith", Price = 39.99m 
}
    };


    // GET: Book
    public IActionResult Index()
    {
      return View(books);
    }


    // GET: Book/Details/5
    public IActionResult Details(int id)
    {
      var book = books.FirstOrDefault(b => b.Id == id);
      if (book == null) return NotFound();
      return View(book);
```

ISDM-NxT

```
}

// GET: Book/Create
public IActionResult Create()
{
    return View();
}

// POST: Book/Create
[HttpPost]
public IActionResult Create(Book book)
{
    if (ModelState.IsValid)
    {
        book.Id = books.Max(b => b.Id) + 1;
        books.Add(book);
        return RedirectToAction(nameof(Index));
    }
    return View(book);
}

// GET: Book/Edit/5
public IActionResult Edit(int id)
{
```

ISDM-NXT

```
    var book = books.FirstOrDefault(b => b.Id == id);

    if (book == null) return NotFound();

    return View(book);

}


// POST: Book/Edit/5

[HttpPost]

public IActionResult Edit(int id, Book book)

{

    if (id != book.Id) return NotFound();


    var existingBook = books.FirstOrDefault(b => b.Id == id);

    if (existingBook == null) return NotFound();


    existingBook.Title = book.Title;

    existingBook.Author = book.Author;

    existingBook.Price = book.Price;

    return RedirectToAction(nameof(Index));

}


// GET: Book/Delete/5

public IActionResult Delete(int id)

{

    var book = books.FirstOrDefault(b => b.Id == id);
```

ISDM-NXT

```
            if (book == null) return NotFound();
            return View(book);
        }

        // POST: Book/Delete/5
        [HttpPost, ActionName("Delete")]
        public IActionResult DeleteConfirmed(int id)
        {
            var book = books.FirstOrDefault(b => b.Id == id);
            if (book != null)
            {
                books.Remove(book);
            }
            return RedirectToAction(nameof(Index));
        }
    }
}
```

The controller handles the following actions:

- **Index**: Displays the list of books.
- **Details**: Displays details of a selected book.
- **Create**: Adds a new book to the list.
- **Edit**: Updates the details of an existing book.
- **Delete**: Deletes a book from the list.

ISDM-NXT

Step 4: Create Views for CRUD Operations

1. **Create Views for the Book Controller**:
    - In the Views/Book folder, create the following view files:

1.1. Index.cshtml

```
@model IEnumerable<BookCrudApp.Models.Book>


<h1>Books List</h1>

<a href="@Url.Action("Create")">Create New Book</a>

<table>

  <thead>

    <tr>

      <th>Title</th>

      <th>Author</th>

      <th>Price</th>

      <th>Actions</th>

    </tr>

  </thead>

  <tbody>

    @foreach (var book in Model)

    {

      <tr>

        <td>@book.Title</td>

        <td>@book.Author</td>

        <td>@book.Price</td>
```

ISDM-NXT

```
            <td>
              <a href="@Url.Action("Details", new { id = book.Id })">Details</a> |
              <a href="@Url.Action("Edit", new { id = book.Id })">Edit</a> |
              <a href="@Url.Action("Delete", new { id = book.Id })">Delete</a>
            </td>
          </tr>
      }
    </tbody>
</table>
```

1.2. Create.cshtml

```
@model BookCrudApp.Models.Book


<h1>Create New Book</h1>
<form method="post">
    <label for="Title">Title</label>
    <input type="text" id="Title" name="Title" value="@Model?.Title" required />


    <label for="Author">Author</label>
    <input type="text" id="Author" name="Author" value="@Model?.Author" required
/>


    <label for="Price">Price</label>
    <input type="number" step="0.01" id="Price" name="Price" value="@Model?.Price"
required />
```

ISDM-NXT

```
    <button type="submit">Create</button>
</form>
```

1.3. Edit.cshtml

```
@model BookCrudApp.Models.Book


<h1>Edit Book</h1>
<form method="post">
    <input type="hidden" id="Id" name="Id" value="@Model.Id" />


    <label for="Title">Title</label>
    <input type="text" id="Title" name="Title" value="@Model?.Title" required />


    <label for="Author">Author</label>
    <input type="text" id="Author" name="Author" value="@Model?.Author" required
/>


    <label for="Price">Price</label>
    <input type="number" step="0.01" id="Price" name="Price" value="@Model?.Price"
required />


    <button type="submit">Save</button>
</form>
```

1.4. Delete.cshtml

```
@model BookCrudApp.Models.Book
```

ISDM-NXT

```
<h1>Delete Book</h1>

<p>Are you sure you want to delete the book titled @Model.Title?</p>

<form method="post">

  <button type="submit">Yes, Delete</button>

</form>

<a href="@Url.Action("Index")">Cancel</a>
```

STEP 5: RUN THE APPLICATION

1. **Run the application**:
   - Press Ctrl+F5 to run the application without debugging.
   - Visit the /Book page to see the list of books.
   - You can now Create, Edit, and Delete books through the application interface.

STEP 6: CONCLUSION

You have successfully created a simple CRUD application using ASP.Net Core and C#. This application allows you to manage a list of books, providing functionalities to add, view, edit, and delete books. This project demonstrates the integration of C# with the ASP.Net MVC framework and how to handle user input and events in an MVC-based web application.

EXERCISE

1. Add validation to the Create and Edit forms (e.g., ensure that the price is a positive value).
2. Implement a search feature to allow users to search for books by title or author.

ISDM-NXT

3. Persist the book data in a database (e.g., using Entity Framework Core) instead of an in-memory list.

ISDM-NXT

ISDM-NXT