**Lecture Slides: SOLID Principles & Advanced C# Concepts**

**Slide 1: Title Slide**

* **Title**: SOLID Principles & Advanced C# Concepts
* **Subtitle**: Enhancing Software Design and C# Programming
* **Date**: [Insert Date]
* **Instructor**: [Your Name]

**Section 1: SOLID Principles**

**Slide 2: Introduction to SOLID**

* **Definition**: SOLID is an acronym for five design principles aimed at making software designs more understandable, flexible, and maintainable.
* **Importance**: Following these principles helps reduce code smells, improve testability, and facilitate easier refactoring.

**Slide 3: Single Responsibility Principle (SRP)**

* **Definition**: A class should have only one reason to change, meaning it should have only one job or responsibility.
* **Example**:

csharp

public class Report {

public void GenerateReport() { /\* Generate report logic \*/ }

}

public class ReportPrinter {

public void Print(Report report) { /\* Print logic \*/ }

}

* + **Explanation**: The Report class handles report generation, while the ReportPrinter class handles printing. This separation makes the code easier to maintain.
* **Activity**: Review a sample codebase to identify classes that violate SRP and refactor them.

**Slide 4: Open/Closed Principle (OCP)**

* **Definition**: Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
* **Example**:

csharp

public abstract class Shape {

public abstract double Area();

}

public class Circle : Shape {

public double Radius { get; set; }

public override double Area() => Math.PI \* Radius \* Radius;

}

public class Rectangle : Shape {

public double Width { get; set; }

public double Height { get; set; }

public override double Area() => Width \* Height;

}

* + **Explanation**: You can add new shapes (like Triangle) without modifying existing classes. This reduces the risk of introducing bugs.
* **Activity**: Extend an existing class hierarchy by adding new features without modifying existing classes.

**Slide 5: Liskov Substitution Principle (LSP)**

* **Definition**: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
* **Example**:

csharp

public class Bird {

public virtual void Fly() { /\* Flying logic \*/ }

}

public class Sparrow : Bird { /\* Inherits Fly \*/ }

public class Ostrich : Bird {

public override void Fly() {

throw new InvalidOperationException("Ostriches can't fly!");

}

}

* + **Explanation**: The Ostrich class violates LSP because replacing a Bird with an Ostrich breaks the expected behavior. Refactor to ensure all subclasses can be substituted without issues.
* **Activity**: Refactor a class hierarchy to ensure compliance with LSP.

**Slide 6: Interface Segregation Principle (ISP)**

* **Definition**: Clients should not be forced to depend on interfaces they do not use.
* **Example**:

csharp

public interface IBird {

void Fly();

void Swim(); // Not all birds swim

}

public interface IFlyable {

void Fly();

}

* + **Explanation**: Separate interfaces allow classes to implement only what they need, improving code clarity and reducing unnecessary dependencies.
* **Activity**: Refactor a large interface into smaller, more specific interfaces.

**Slide 7: Dependency Inversion Principle (DIP)**

* **Definition**: High-level modules should not depend on low-level modules; both should depend on abstractions.
* **Example**:

csharp

public interface IMessageService {

void Send(string message);

}

public class EmailService : IMessageService {

public void Send(string message) { /\* Email sending logic \*/ }

}

public class Notification {

private IMessageService \_service;

public Notification(IMessageService service) {

\_service = service;

}

}

* + **Explanation**: This decouples the Notification class from specific implementations of message services, allowing for easier testing and modification.
* **Activity**: Refactor a class to use dependency injection for its dependencies.

**Section 2: Advanced C# Concepts**

**Slide 8: Introduction to Advanced C#**

* Overview of advanced C# features that enhance programming efficiency and software design.

**Slide 9: Delegates**

* **Definition**: A type that represents references to methods with a specific parameter list and return type.
* **Example**:

csharp

public delegate void Notify(string message);

public class Process {

public Notify OnCompleted;

public void Execute() {

// Process logic

OnCompleted?.Invoke("Process completed!");

}

}

* + **Explanation**: Delegates allow methods to be passed as parameters, enabling event-driven programming.
* **Activity**: Create a simple process class that raises an event when completed.

**Slide 10: Events**

* **Definition**: A way for a class to provide notifications to other classes when something of interest occurs.
* **Example**:

csharp

public class Process {

public event Notify ProcessCompleted;

public void Execute() {

// Process logic

ProcessCompleted?.Invoke("Process completed!");

}

}

* + **Explanation**: Events use delegates to notify subscribers when they are raised.
* **Activity**: Implement an event in a class and subscribe to it in another class.

**Slide 11: Lambda Expressions**

* **Definition**: A concise way to represent anonymous methods using a syntax that is more readable.
* **Example**:

csharp

Func<int, int> square = x => x \* x;

var numbers = new List<int> { 1, 2, 3 };

var squares = numbers.Select(square);

* + **Explanation**: Lambda expressions simplify the syntax for delegates and are often used with LINQ.
* **Activity**: Use a lambda expression in a LINQ query to filter a list.

**Slide 12: LINQ (Language Integrated Query)**

* **Definition**: A set of methods for querying collections using a SQL-like syntax.
* **Example**:

csharp

var lowPriceProducts = products.Where(p => p.Price < 100).ToList();

* + **Explanation**: LINQ provides a powerful and expressive way to manipulate collections.
* **Activity**: Write a LINQ query to filter and sort a list of products.

**Slide 13: Asynchronous Programming (async/await)**

* **Definition**: Allows for non-blocking operations, enabling efficient handling of long-running tasks.
* **Example**:

csharp

public async Task<string> FetchDataAsync() {

using (var client = new HttpClient()) {

return await client.GetStringAsync("https://example.com");

}

}

* + **Explanation**: The async keyword allows the method to be asynchronous, and await pauses execution until the task completes.
* **Activity**: Create a simple async method and call it from a synchronous context.

**Section 3: Wrap-Up and Q&A**

**Slide 14: Summary**

* Review of SOLID Principles and Advanced C# Concepts.
* Importance of applying these principles and concepts in real-world applications.

**Slide 15: Q&A Session**

* Open the floor for questions and clarifications.

**Activities and Code Walkthroughs**

1. **Group Discussions**: Discuss real-world applications of each SOLID principle and advanced C# feature.
2. **Hands-On Coding**: Implement examples in small groups, focusing on applying SOLID principles to existing code.
3. **Code Reviews**: Share and review code snippets to identify adherence to SOLID principles and best practices in C#.
4. **Mini-Projects**: Create a small application that incorporates learned concepts, focusing on SOLID principles and advanced C# features.

This course outline provides a comprehensive guide to understanding both SOLID principles and advanced C# concepts, complete with clear examples and engaging activities to reinforce learning.

**Slide 1: Introduction to IEnumerable**

* **Definition**: IEnumerable<T> is an interface in C# that defines a method for iterating over a collection of a specified type. It is part of the System.Collections.Generic namespace.
* **Purpose**: Enables reading through a collection without exposing the underlying data structure.

**Slide 2: Basic Usage of IEnumerable**

* **Example**:

csharp

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

IEnumerable<int> enumerableNumbers = numbers;

foreach (int number in enumerableNumbers) {

Console.WriteLine(number);

}

* **Explanation**: In this example, List<int> is assigned to IEnumerable<int>, allowing iteration using a foreach loop.

**Slide 3: Advantages of Using IEnumerable**

* **Lazy Evaluation**:
  + Only retrieves items when needed, which can improve performance.
* **Flexibility**:
  + Can work with various collection types (arrays, lists, etc.).
* **Abstraction**:
  + Hides the collection's implementation details, allowing for easier code maintenance.

**Slide 4: Comparing IEnumerable with Other Collection Types**

| **Feature** | **IEnumerable<T>** | **IList<T>** | **List<T>** |
| --- | --- | --- | --- |
| **Read/Write** | Read-only | Read/Write | Read/Write |
| **Performance** | Lazy evaluation (deferred execution) | Immediate access to items | Immediate access to items |
| **Flexibility** | Can be used with LINQ and queries | More methods like Add, Remove | More methods like Add, Remove |
| **Implementation Hiding** | Yes (abstracts collection type) | No (exposes more details) | No (exposes more details) |

**Slide 5: IEnumerable in LINQ**

* **Example**:

csharp

var evenNumbers = numbers.Where(n => n % 2 == 0);

foreach (var num in evenNumbers) {

Console.WriteLine(num);

}

* **Explanation**: LINQ queries return IEnumerable<T>, allowing for deferred execution. The Where method filters the list, but the items are only evaluated when iterated over, thus improving efficiency.

**Slide 6: Practical Activity: IEnumerable vs. List**

* **Task**:
  + Create a list of products (using a List<Product>).
  + Use IEnumerable<Product> to filter products based on a condition (e.g., price).
  + Compare the performance and readability of using IEnumerable vs. List directly.

**Slide 7: Summary of IEnumerable**

* **Key Takeaways**:
  + IEnumerable<T> provides a unified way to iterate over collections.
  + It promotes lazy evaluation and flexibility, making it ideal for LINQ and other query operations.
  + Understanding its differences from other collection types is crucial for effective C# programming.