Programming Language - C#

Lambda Expressions

Agenda

- Introduction to Lambda Expressions
- Syntax of Lambda Expressions
- Lambda Expressions vs Anonymous Methods
- Using Lambda Expressions
- Real-Time Examples
- Advanced Usage
- Benefits and Limitations
- Best Practices

Introduction to Lambda Expressions

- Definition:
  - Lambda expressions are anonymous functions that can contain expressions or statements and are used to create delegates or expression tree types.
- Purpose:
  - They provide a concise way to represent anonymous methods using a clear and readable syntax.

Syntax of Lambda Expressions

- Basic Syntax:
  - (parameters) => expression
  - Example: (x, y) => x + y
- Single Parameter Syntax:
  - parameter => expression
  - Example: x => x * x
- Statement Block Syntax:
  - (parameters) => { statements }
  - Example: (x, y) => { return x + y; }

Lambda Expressions vs Anonymous Methods

- Anonymous Methods:
  - Func<int, int, int> add = delegate(int x, int y) { return x + y; };

- Lambda Expressions:
  - Func<int, int, int> add = (x, y) => x + y;

Using Lambda Expressions

- With Delegates
  - Func<int, int, int> add = (x, y) => x + y;
  - int result = add(3, 4); // result is 7
- With LINQ
  - int[] numbers = { 1, 2, 3, 4, 5 };
  - var evenNumbers = numbers.Where(n => n % 2 == 0).ToArray();
- With Events
  - button.Click += (sender, e) => MessageBox.Show("Button clicked!");

Examples

- Example 1: Filtering a List

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
List<int> evenNumbers = numbers.Where(n => n % 2 == 0).ToList();
```

- Example 2: Sorting a List of Objects

```
List<Person> people = new List<Person>
{
    new Person { Name = "Alice", Age = 30 },
    new Person { Name = "Bob", Age = 25 }
};
people.Sort((p1, p2) => p1.Age.CompareTo(p2.Age));
```

Advanced Usage

- Capturing Variables:

```
int factor = 2;
Func<int, int> multiplier = x => x * factor;
int result = multiplier(5); // result is 10
```

- Expression Trees:

```
Expression<Func<int, int, int>> addExpr = (x, y) => x + y;
Func<int, int, int> add = addExpr.Compile();
int result = add(3, 4); // result is 7
```

- Recursive Lambda Expressions:

```
Func<int, Func<int, int>> factorial = null;
factorial = x => x == 0 ? 1 : x * factorial(x - 1);
int result = factorial(5); // result is 120
```

Benefits and Limitations

- Benefits:
  - Conciseness:
    - Shortens the code and improves readability.
  - Flexibility:
    - Easily define inline functions.
  - Integration with LINQ:
    - Enhances the use of LINQ queries.
- Limitations:
  - Readability:
    - Overuse can lead to less readable code.
  - Debugging:
    - Can be harder to debug compared to named methods.

Best Practices

- Keep It Simple:
  - Use lambda expressions for simple operations.
- Use Meaningful Parameter Names:
  - Avoid single-letter parameter names unless in very simple expressions.
- Avoid Deep Nesting:
  - Don't nest lambda expressions deeply.
- Prefer Readability:
  - Use named methods if the lambda expression becomes too complex.
- Document Complex Expressions:
  - Provide comments for complex lambda expressions to improve maintainability.

Best Practices

- Versioning:
  - Handle versioning in serialized data to manage changes in object structure.
- Security:
  - Ensure secure handling of serialized data to prevent attacks like deserialization vulnerabilities.
- Performance:
  - Consider the performance impact of serialization, especially for large objects.
- Data Contracts:
  - Use data contracts for version-tolerant serialization.

Thank you

Innovative Services

Passionate Employees

Delighted Customers

© Copyright 2012 by Hexaware Technologies Limited

All rights reserved. All text included in this document is the exclusive property of Hexaware Technologies Ltd, and may not be copied, reproduced, or used in any way without the express permission in writing of Hexaware Technologies Ltd.