

Atividades e Exemplos em Programação com C#



Status

Lendo - In progress

▼ Quinta-feira Aula 04 - Colecoes listas e linq

O conteúdo sobre Arrays e Coleções em C# explica estruturas de dados essenciais que ajudam a organizar e manipular grupos de elementos. Vou resumir e explicar os principais pontos:

1. Arrays

Os arrays são estruturas de dados de tamanho fixo, usados quando se sabe a quantidade de elementos que serão armazenados. Eles permitem acesso direto aos elementos por meio de índices.

Exemplo:

```
int[] numeros = new int[5]; // Um array de inteiros com 5
```

Arrays têm a limitação de não permitir adicionar ou remover elementos após sua criação.

2. List<T>

A `List<T>` é uma coleção mais flexível que permite aumentar ou diminuir o número de elementos dinamicamente. Ela oferece métodos para adicionar, remover e buscar elementos.

Exemplo:

```
List<string> frutas = new List<string>();  
frutas.Add("Maçã"); // Adiciona um elemento  
frutas.Remove("Banana"); // Remove um elemento
```

3. Dictionary<TKey, TValue>

O `Dictionary<TKey, TValue>` armazena dados em pares de chave-valor, permitindo acessar rapidamente o valor associado a uma chave. As chaves precisam ser únicas.

Exemplo:

```
Dictionary<int, string> pessoas = new Dictionary<int, string>();  
pessoas.Add(1, "Alice");
```

4. HashSet<T>

O `HashSet<T>` é uma coleção que não permite duplicatas. Ele é útil quando é importante garantir que todos os elementos sejam únicos.

Exemplo:

```
HashSet<string> nomes = new HashSet<string>();  
nomes.Add("Alice"); // Elementos duplicados não serão adicionados
```

5. Queue<T>

A `Queue<T>` segue o princípio FIFO (First In, First Out), onde o primeiro elemento a entrar é o primeiro a sair. É usada em cenários como filas de espera.

Exemplo:

```
Queue<string> fila = new Queue<string>();  
fila.Enqueue("Cliente 1"); // Adiciona no final  
fila.Dequeue(); // Remove do início
```

6. Stack<T>

A `Stack<T>` funciona de maneira oposta à fila, seguindo o princípio LIFO (Last In, First Out), onde o último elemento a entrar é o primeiro a sair. É útil em cenários como controle de histórico.

Exemplo:

```
Stack<string> pilha = new Stack<string>();  
pilha.Push("Item 1"); // Adiciona no topo  
pilha.Pop(); // Remove do topo
```

Comparação Geral

- **Array**: Tamanho Fixo, ordenado, permite duplicatas.
 - **List<T>**: Dinâmico, ordenado, permite duplicatas.
 - **Dictionary<TKey, TValue>**: Chave-valor, não ordenado, chaves únicas.
 - **HashSet<T>**: Elementos únicos, não ordenado.
 - **Queue<T>**: FIFO, não ordenado.
 - **Stack<T>**: LIFO, não ordenado.
-

1. O que é LINQ?

LINQ integra consultas dentro do código C#, facilitando a leitura e manutenção, além de aproveitar a verificação de tipos em tempo de compilação, evitando erros comuns.

2. Tipos de LINQ

- **LINQ to Objects**: Para coleções em memória (arrays, listas).
 - **LINQ to SQL**: Para consultas em bancos de dados SQL.
 - **LINQ to XML**: Para consultas em arquivos XML.
 - **LINQ to Entities**: Para usar com o Entity Framework.
-

Métodos Básicos de LINQ

3.1. **Select**

O método **Select** projeta os dados, ou seja, transforma cada item de uma coleção em um novo formato.

Exemplo:

```
List<int> numeros = new List<int> { 1, 2, 3, 4, 5 };
var quadrados = numeros.Select(n => n * n);

foreach (var quadrado in quadrados)
{
    Console.WriteLine(quadrado); // Saída: 1, 4, 9, 16, 25
}
```

Explicação desse código passo a passo para que tudo fique mais claro, especialmente o uso do

n

1.

Criação da Lista

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

Aqui, você está criando uma lista chamada **numeros** que contém cinco números inteiros: **{ 1, 2, 3, 4, 5 }**. É simplesmente uma coleção de números.

2. Método **Select**

```
var quadrados = numeros.Select(n => n * n);
```

Agora, vamos analisar esta linha:

`numeros.Select(n => n * n)` está aplicando o método `Select` à lista `numeros`. O `Select` é usado para transformar (ou projetar) os itens da lista.

`n` representa **cada elemento** da lista `numeros` conforme o LINQ itera sobre ela. Então, na primeira iteração, `n` será 1, depois 2, depois 3, e assim por diante.

`n => n * n` é uma **expressão lambda**, que define a operação que será aplicada a cada elemento da lista. O `n` antes do `=>` é o valor de entrada (neste caso, cada número da lista), e o `n * n` após o `=>` é o que vai ser retornado (ou seja, o quadrado de `n`).

Resumindo:

- O LINQ vai pegar cada número em `numeros`, atribuí-lo a `n` e, em seguida, calcular `n * n` (o quadrado daquele número).
- A lista resultante (`quadrados`) terá os valores: `{ 1, 4, 9, 16, 25 }`.

3.

Laço `foreach`

```
foreach (var quadrado in quadrados)
{
    Console.WriteLine(quadrado); // Saída: 1, 4, 9, 16, 25
}
```

Nesta parte:

- O laço `foreach` vai iterar sobre a coleção `quadrados`, que contém os números quadrados gerados pelo `Select`.
- Para cada elemento da lista `quadrados`, a variável `quadrado` vai armazenar o valor atual (ex.: primeiro 1, depois 4, e assim por diante), e o `Console.WriteLine(quadrado)` irá imprimir esse valor.

Resumo:

- `n => n * n` é uma expressão que pega cada elemento `n` da lista original e o multiplica por ele mesmo, criando uma nova lista com os quadrados dos números.
- O `foreach` percorre essa nova lista e imprime cada quadrado.

▼ 3.2. `Where`

O método

`Where` filtra os elementos de uma coleção, retornando apenas aqueles que atendem a uma condição.

Exemplo

```
List<int> numeros = new List<int> { 1, 2, 3, 4, 5, 6 };
var numerosPares = numeros.Where(n => n % 2 == 0);

foreach (var numero in numerosPares)
{
    Console.WriteLine(numero); // Saída: 2, 4, 6
}
```

▼ 3.3. `OrderBy`

`OrderBy` ordena os elementos de uma coleção em ordem crescente com base em um critério.

Exemplo

```
List<Pessoa> pessoas = new List<Pessoa>
{
    new Pessoa { Nome = "Carlos", Idade = 30 },
    new Pessoa { Nome = "Ana", Idade = 25 },
    new Pessoa { Nome = "Bruno", Idade = 35 }
}
```

```
};

var pessoasOrdenadas = pessoas.OrderBy(p => p.Idade);

foreach (var pessoa in pessoasOrdenadas)
{
    Console.WriteLine($"{pessoa.Nome}: {pessoa.Idade} a
}
```

▼ 3.4. `GroupBy`

O método

`GroupBy` agrupa elementos de uma coleção com base em uma chave, permitindo operações de agregação.

Exemplo

```
List<Pessoa> pessoas = new List<Pessoa>
{
    new Pessoa { Nome = "Carlos", Cidade = "São Paulo" },
    new Pessoa { Nome = "Ana", Cidade = "Rio de Janeiro" },
    new Pessoa { Nome = "Bruno", Cidade = "São Paulo" }
};

var gruposPorCidade = pessoas.GroupBy(p => p.Cidade);

foreach (var grupo in gruposPorCidade)
{
    Console.WriteLine($"Cidade: {grupo.Key}");
    foreach (var pessoa in grupo)
    {
        Console.WriteLine($" - {pessoa.Nome}");
    }
}
```

Resumo

- **LINQ** facilita consultas dentro do código C# sem precisar de SQL ou outras linguagens externas.
 - Os principais métodos, como `Select`, `Where`, `OrderBy` e `GroupBy`, ajudam a filtrar, projetar, ordenar e agrupar dados.
 - É muito útil em várias situações, desde trabalhar com listas e arrays até consultar bancos de dados ou XML.
-

Comparando Má e Boa Prática: Consultas com SQL e LINQ em C#, Java e PHP

Vamos explorar um exemplo de **má prática** onde consultas SQL são feitas diretamente dentro do código, o que pode gerar problemas de legibilidade, manutenção, e segurança (**como vulnerabilidade a injeção de SQL**). Em seguida, veremos a **boa prática** utilizando **LINQ em C#** e boas práticas equivalentes em **Java** e **PHP**.

Má Prática: Consultas SQL Diretas

Exemplo Ruim em **Java**

```
import java.sql.*;

public class ExemploRuimJava {
    public static void main(String[] args) {
        try {
            Connection conn = DriverManager.getConnection(
                Statement stmt = conn.createStatement();
                String sql = "SELECT * FROM clientes WHERE ida
                ResultSet rs = stmt.executeQuery(sql);
```



```

        while (rs.next()) {
            System.out.println("Nome: " + rs.getString(1));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

Exemplo Ruim em **PHP**

```

<?php
$servername = "localhost";
$username = "usuario";
$password = "senha";
$dbname = "meubanco";

$conn = new mysqli($servername, $username, $password, $dbname);

$sql = "SELECT * FROM clientes WHERE idade > 30";
$result = $conn->query($sql);

if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        echo "Nome: " . $row["nome"] . " - Idade: " . $row["idade"] . "<br>";
    }
} else {
    echo "Nenhum resultado encontrado";
}
$conn->close();
?>

```

Exemplo Ruim em

C# (Sem LINQ)

```
using System;
using System.Data.SqlClient;

class ExemploRuimCSharp {
    static void Main(string[] args) {
        SqlConnection conn = new SqlConnection("Data Source=...;Initial Catalog=...;User ID=...;Password=...");
        conn.Open();

        string sql = "SELECT * FROM clientes WHERE idade > 30";
        SqlCommand cmd = new SqlCommand(sql, conn);
        SqlDataReader reader = cmd.ExecuteReader();

        while (reader.Read()) {
            Console.WriteLine("Nome: " + reader["nome"] + " Idade: " + reader["idade"]);
        }

        conn.Close();
    }
}
```

Problemas com a Má Prática:

1. **Segurança:** Consultas SQL diretas estão vulneráveis a **injeção de SQL** se não forem tratadas corretamente.
2. **Legibilidade:** O código mistura a lógica de negócio (filtrar clientes com idade > 30) com consultas SQL.
3. **Manutenção Difícil:** Alterar a consulta exige mudanças no código, o que dificulta a manutenção e testes.
4. **Reutilização de Código:** Não há um bom isolamento de lógica de consulta; você precisaria copiar e colar SQL.

Boa Prática: Usando LINQ em C# e Boas Práticas em Java e PHP

Exemplo Correto em
C# com **LINQ**

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Cliente
{
    public string Nome { get; set; }
    public int Idade { get; set; }
}

class ExemploCorretoCSharp {
    static void Main(string[] args) {
        // Simulando uma lista de clientes
        List<Cliente> clientes = new List<Cliente>
        {
            new Cliente { Nome = "Ana", Idade = 25 },
            new Cliente { Nome = "Carlos", Idade = 35 },
            new Cliente { Nome = "Bruno", Idade = 40 }
        };

        // Usando LINQ para filtrar clientes com idade > 30
        var clientesFiltrados = clientes.Where(c => c.Idade > 30);

        foreach (var cliente in clientesFiltrados) {
            Console.WriteLine("Nome: " + cliente.Nome + ", Idade: " + cliente.Idade);
        }
    }
}
```

Exemplo Correto em
Java com **JPA** (Hibernate)

```
import javax.persistence.*;
import java.util.List;

public class ExemploCorretoJava {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("persistence-unit");
        EntityManager em = emf.createEntityManager();

        // Usando JPQL (JPA Query Language) para fazer a consulta
        String jpql = "SELECT c FROM Cliente c WHERE c.id = 1";
        List<Cliente> clientes = em.createQuery(jpql, Cliente.class).getResultList();

        for (Cliente cliente : clientes) {
            System.out.println("Nome: " + cliente.getNome());
        }

        em.close();
        emf.close();
    }
}
```

Exemplo Correto em
PHP com **PDO**

```
<?php
$servername = "localhost";
$username = "usuario";
$password = "senha";
$dbname = "meubanco";

$conn = new PDO("mysql:host=$servername;dbname=$dbname", $username, $password);
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

```
$stmt = $conn->prepare("SELECT nome, idade FROM clientes W
$stmt->bindParam(':idade', $idade);
$idade = 30;
$stmt->execute();

$result = $stmt->fetchAll();

foreach ($result as $row) {
    echo "Nome: " . $row["nome"] . " - Idade: " . $row["id.
}
?>
```

Por que usar boas práticas é importante?

1. **Segurança:** Boas práticas como usar **LINQ em C#, JPA em Java, e PDO em PHP** ajudam a proteger o código contra **injeção de SQL**, uma das vulnerabilidades mais comuns em software.
2. **Legibilidade e Manutenção:** Separar a lógica da aplicação da lógica de acesso a dados torna o código mais legível e fácil de manter. Por exemplo, com LINQ em C#, as consultas são feitas diretamente sobre objetos, tornando o código mais compreensível.
3. **Isolamento e Testabilidade:** Com LINQ ou ORM (Object-Relational Mapping) como JPA e Hibernate, a lógica de consultas pode ser isolada, facilitando **testes unitários** sem dependência direta de um banco de dados. Em vez de testar SQL, você pode testar a lógica de filtragem diretamente sobre coleções de objetos.

Como testar essas consultas corretamente?

- **Teste Unitário (C#):** Como o LINQ trabalha com coleções em memória, é fácil testar as consultas sem acessar um banco de dados real. Você pode criar listas de clientes fictícios e testar os resultados das consultas **LINQ**.
- **Teste Unitário (Java):** Usando JPA com frameworks como **JUnit** e **Mockito**, é possível testar as regras de negócio separadamente da consulta ao banco de dados, utilizando um banco de dados em memória (como H2).
- **Teste Unitário (PHP):** Usando **PHPUnit**, você pode testar as consultas e o tratamento de dados de maneira isolada, mockando as conexões de banco de dados.

Exemplo de Teste Unitário com LINQ (C#):

```
using System;
using System.Collections.Generic;
using System.Linq;
using Xunit; // Biblioteca de testes

public class Cliente
{
    public string Nome { get; set; }
    public int Idade { get; set; }
}

public class ClienteTests
{
    [Fact]
    public void TestFiltragemClientesPorIdade()
    {
        // Arrange
        List<Cliente> clientes = new List<Cliente>
        {
```

```

        new Cliente { Nome = "Ana", Idade = 25 },
        new Cliente { Nome = "Carlos", Idade = 35 }
    };

    // Act
    var clientesFiltrados = clientes.Where(c => c.Idade > 25);

    // Assert
    Assert.Single(clientesFiltrados); // Deve haver um
    Assert.Equal("Carlos", clientesFiltrados[0].Nome);
}
}

```

Neste teste, você simula uma lista de clientes e testa se a filtragem por idade funciona corretamente.

Conclusão

- Usar **LINQ** em C# ou práticas equivalentes em outras linguagens (como ORM em Java e PDO em PHP) melhora a **segurança, manutenção e legibilidade** do código.
- Consultas SQL diretas podem levar a códigos difíceis de manter e vulneráveis.
- Testes unitários tornam o código mais confiável, permitindo verificar a lógica sem acessar bancos de dados reais.