C# 6 – tips and tricks

[Digite o resumo do documento aqui. Em geral, um resumo é um apanhado geral do conteúdo do documento. Digite o resumo do documento aqui. Em geral, um resumo é um apanhado geral do conteúdo do documento.]

[Digite o subtítulo do documento]

Sumário

[Inicializando Propriedades Automáticas 2](#_Toc440987034)

[Propriedades “Read Only” 3](#_Toc440987035)

[Expression Bodied Methods 3](#_Toc440987036)

[Inicialização de Dicionários 3](#_Toc440987037)

[Null Condicional 4](#_Toc440987038)

[Acesso Simplificado a Membros Estáticos 4](#_Toc440987039)

[Filtros para Exception 5](#_Toc440987040)

[Operador nameof 6](#_Toc440987041)

# Inicializando Propriedades Automáticas

Propriedades automáticas não são novas, elas apareceram no C# 3. Veja o código abaixo e observe que ele apresenta três propriedades automáticas:

public class AutoPropertyInitialier {

public string FirstName { get; set; }

public string LastName { get; set; }

public string FullName { get; set; }

public AutoPropertyInitialier() {

FirstName = "José";

LastName = "Silva";

FullName = FirstName + " " + LastName;

}

}

Como você pode ver, as propriedades FirstName e LastName (inclusive a FullName) são propriedades automáticas, sendo que FullName é inicializada no construtor.

Com os recursos do C# 6 essa construção pode ser modificada para algo bem diferente, veja:

public class AutoPropertyInitialier {

public string FirstName { get; set; } = "José";

public string LastName { get; set; } = "Silva";

public string FullName { get; set; }

public AutoPropertyInitialier() {

FullName = FirstName + " " + LastName;

}

}

No caso da propriedade FullName ela ainda precisa ser inicializada dentro do construtor, pois somente após a inicialização da classe (criação da instância) o assessor “this” estará disponível.

## Propriedades “Read Only”

Também é possível criar propriedades de somente leitura (readonly). Anteriormente, faríamos dessa forma:

public class AutoPropertyInitialier {

private readonly List<string> \_values;

public List<string> Values {

get { return \_values; }

}

public AutoPropertyInitialier() {

\_values = new List<string>();

}

}

Como a nova abordagem, podemos encurtar o caminho:

public class AutoPropertyInitialier {

public List<string> Values { get; } = new List<string>();

}

Bem mais simples, não?

# Expression Bodied Methods

Esse é uma coisa muito simples (e acho que um tanto inútil). É possível criar métodos como se fossem expressões lambda.

public void DoSomething() => Console.WriteLine("Hello World!"); // Método sem parâmetros

public void DoSomethingWithParameter(string parameter) => Console.WriteLine(parameter); // Método com parâmetros

public string ThatProperty => "Property"; // Propriedade

# Inicialização de Dicionários

Um nova forma de inicializar dicionários,

var dictionary = new Dictionary<string, string> {

["ABC"] = "123",

["DEF"] = "123",

["GHI"] = "123"

};

# Null Condicional

Acredito que essa seja uma das **features** mais interessantes do C# 6. É possível acessar propriedades dentro de um objeto, mesmo ele sendo **null**, se ter de recorrer a verificação explícita de **null**. Veja um exemplo:

public class Student {

public string Name { get; set; }

public DateTime Birthday { get; set; }

public Address Address { get; set; }

}

public class Address {

public string Street { get; set; }

public int ZipCode { get; set; }

}

var student = new Student();

Console.WriteLine("Endereço: {0}", student?.Address?.Street);

Neste caso, mesmo a propriedade “Address”, dentro de “Student”, sendo nula, não ocorre um erro de “NullReferenceException”. A verificação e retorno estão implícitas.

# Acesso Simplificado a Membros Estáticos

Essa **feature** não é tão importante, mas pode ajudar a escrever um pouco menos. Na declaração de “using”, é possível deixar especificado qual classe você está utilizando (um pouco parecido com o que o Java faz). Feito isso, você tem acesso direto aos métodos estáticos presentes nessa classe. Um exemplo básico é o membro **Console**. Veja o exemplo:

using static System.Console;

namespace MyApplication {

public class Program {

public static void Main(string[] args) {

WriteLine("Hello World!");

ReadKey();

}

}

}

# Filtros para Exception

Agora é possível filtrar exceções pelas variáveis de contexto. Isso pode se tornar muito prático quando é necessário lançar uma exceção específica dado o valor de uma variável. Veja:

using System;

namespace MyApplication {

public class Program {

public static void Main(string[] args) {

var i = 1;

try {

throw new Exception("ERROR");

} catch (Exception ex) when (i == 1) {

Console.WriteLine("i == 1");

} catch (Exception ex) when (i == 2) {

Console.WriteLine("i == 2");

} catch (Exception ex) when (i == 3) {

Console.WriteLine("i == 3");

}

Console.ReadKey();

}

}

}

# Operador nameof

O operador **nameof** aceita o elemento do código e retorna uma string literal com o nome do elemento fornecido. Exemplo:

using System;

namespace MyApplication {

public class Program {

public static void Main(string[] args) {

Console.WriteLine(nameof(Student));

Console.WriteLine(nameof(Student.Roll));

Console.WriteLine(nameof(Student.Name));

Console.WriteLine(nameof(Student.Address));

Console.ReadKey();

}

public class Student {

public int Roll { get; set; }

public string Name { get; set; }

public Address Address { get; set; }

}

public class Address {

public int ZipCode { get; set; }

public string Street { get; set; }

}

}

}