UNIDADE 11 – EXCEPTIONS

Nessa unidade veremos resumo bem objetivo de como usar o recurso das exceções.

As exceções são mecanismos primários para comunicar condições de erros. Elas possuem um grande poder e isso traz também grandes responsabilidades. Dessa forma, não devemos abusar deste recurso, mas saber usá-lo com bom senso.

SUMÁRIO

1 EXCEPTIONS

Como erros podem ocorrer durante a execução de uma aplicação, devemos definir como eles serão tratados. Tradicionalmente, códigos de erro são utilizados para lidar com falhas na execução de um programa. Nesta abordagem, os métodos devolveriam números inteiros para indicar o tipo de erro que ocorreu.

```
1  int Deposita(double valor)
2  {
3      if (valor < 0)
4      {
5          return 107; // código de erro para valor negativo
6      }
7      else
8      {
9          this.Saldo += valor;
10         return 0; // sucesso
11     }
12 }
```

Tabela 1: Utilizando códigos de erro

Utilizar códigos de erro exige uma vasta documentação dos métodos para explicar o que cada código significa. Alémdisso, esta abordagem "gasta" o retorno do método impossibilitando que outros tipos de dados sejam devolvidos. Em outras palavras, ou utilizamos o retorno para devolver códigos de erro ou para devolver algo pertinente a lógica natural do método. Não é possível fazer as duas coisas sem nenhum tipo de "gambiarra".

```
1  ??? GeraRelatorio()
2  {                          ; Program
3      if (...)
4      {
5          return 200; // código de erro tipo1
6      }
7      else
8      {
9          Relatorio relatorio = ...
10         return relatorio;
11     }
12 }
```

Tabela 2: Código de erro e retorno lógico

Observe que no código do método GeraRelatorio() seria necessário devolver dois tipos de dados incompatíveis: int e referências de objetos da classe Relatorio. Porém, não é possível definir dois tipos distintos como retorno de um método.

A linguagem C# tem uma abordagem própria para lidar com erros de execução. Na abordagem do C# não são utilizados códigos de erro ou os retornos lógicos dos métodos.

1.1 EXCEPTIONS E SYSTEM EXCEPTIONS

Na plataforma .NET, os erros de execução são definidos por classes que derivam direta ou indiretamente da classe **System.Exception**. Diversos erros já estão definidos na plataforma .NET. As classes que modelam os erros pré-definidos derivam da classe **System.SystemException**. A seguir uma tabela com as principais classes derivadas de **System.SystemException**

Exception	Descrição
DivideByZeroException	Erro gerado quando dividimos números inteiros por zero.
IndexOutOfRangeException	Erro gerado quando acessamos posições inexistentes de um array
NullReferenceException	Erro gerado quando utilizamos referências nulas
InvalidCastException	Erro gerado quando realizamos um casting incompatível

1.2 LANÇANDO ERROS

Para lançar um erro, devemos criar um objeto de qualquer classe que deriva de Exception para representar o erro que foi identificado.

Depois de criar uma exception podemos "lançar" a referência dela utilizando o comando **throw**. Observe o exemplo utilizando a classe System.ArgumentException que deriva indiretamente da classe System.Exception.

```
1  if (valor < 0)
2  {
3      System.ArgumentException erro = new System.ArgumentException();
4      throw erro;
5  }
```

Tabela 3: Lançando uma System.ArgumentException

1.3 CAPTURANDO ERROS

Em um determinado trecho de código, para capturar uma exception devemos utilizar o comando try-cacth.

```
1  class Teste
2  {
3      static void Main()
4      {
5          Conta c = new Conta();
6
7          try
8          {
9              c.Deposita(100);
10         }
11         catch (System.ArgumentException e)
12         {
13             System.Console.WriteLine(" Houve um erro ao depositar ");
14         }
15     }
   }
```

Tabela 4: Teste.cs

Podemos encadear vários blocos catch para capturar exceptions de classes diferentes.

```
1  class Teste
2  {
3      static void Main()
4      {
5          Conta c = new Conta();
6
7          try
8          {
9              c.Deposita(100);
10         }
11         catch (System.ArgumentException e)
12         {
13             System.Console.WriteLine(" Houve uma System . ArgumentException ao depositar ");
14         }
15         catch (System.IO.FileNotFoundException e)
16         {
17             System.Console.WriteLine(" Houve um FileNotFoundException ao depositar ");
18         }
19     }
   }
```

Tabela 5: Teste.cs

1.4 FINALLY

Se um erro acontecer no bloco try ele é abortado. Consequentemente, nem sempre todas as linhas do bloco try serão executadas. Além disso, somente um bloco catch é executado quando ocorre um erro.

Em alguns casos, queremos executar um trecho de código independentemente se houver erros ou não. Para isso podemos, utilizar o bloco **finally**.

```
1  try
2  {
3      // código
4  }
5  catch (DivideByZeroException e)
6  {
7      Console.WriteLine(" Tratamento de divisão por zero ");
8  }
9  catch (System.NullReferenceException e)
10 {
11     Console.WriteLine(" Tratamento de referência nula ");
12 }
13 finally
14 {
15     // código que deve ser sempre executado
16 }
```

Tabela 6: Utilizando o bloco finally

1.5 EXERCÍCIOS DE FIXAÇÃO

1) Crie um projeto no chamado **Unidade 11** e uma pasta **ExercicioFixacao**.

2) Crie uma classe para modelar os funcionários do sistema do banco.

```
1  using System;
2
3  class Funcionario
4  {
5      private double salario;
6
7      public void AumentaSalario(double aumento)
8      {
9          if (aumento < 0)
10         {
11             ArgumentException erro = new ArgumentException();
12             throw erro;
13         }
14     }
15 }
```

Tabela 7: Funcionario.cs

3) Agora teste a classe Funcionario.

```
1  class TestaFuncionario
2  {
3      private static void Main()
4      {
5          Funcionario f = new Funcionario();
6          f.AumentaSalario(-1000);
7      }
   }
```

Tabela 8: TestaFuncionario.cs

Execute e observe o erro no console.

4) Altere o teste para capturar o erro.

```
1  class TestaFuncionario
2  {
3      static void Main()
4      {
5          Funcionario f = new Funcionario();
6
7          try
8          {
9              f.AumentaSalario(-1000);
10         }
11         catch (ArgumentException e)
12         {
13             Console.WriteLine(" Houve uma ArgumentException ao aumentar o salário");
14         }
15     }
   }
```

Tabela 9: TestaFuncionario.cs

1.6 EXERCÍCIOS COMPLEMENTARES

1) Crie uma pasta **ExerciciosComplementares** na **Unidade 11**.

2) Copie alguma aplicação que você já fez e customize uma exceptions e faça o tratamento.

Bons Estudos!